Introduction to Web Services Development

CS311 – Handouts

Syed Nauman Ali Shah

Contents

Module 1 (Object Oriented Paradigm)

Objective:

In this lecture we will discuss Object Orientation in detail. It will be followed by discussion on Data Modeling with reference to Abstraction, Inheritance, Encapsulation and Information Hiding, what are the advantages and they are used in classes and their examples.

What is an Object Orientation?

In real world, object oriented approach focuses on objects that represents abstract or concrete things.

What is a Model?

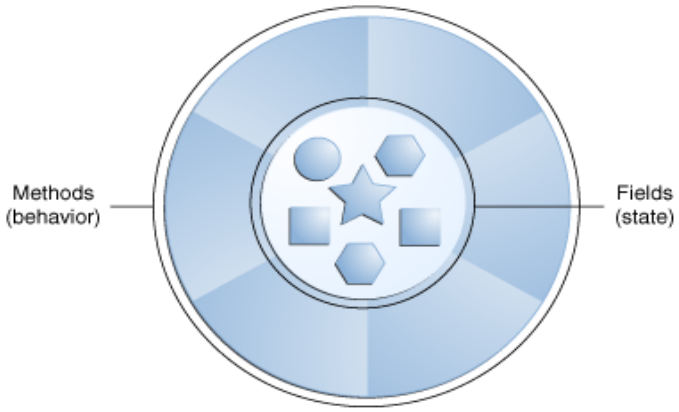
Modeling is an approach that is used at the beginning of software. The reasons to model a system are:

Communication: Model diagrams before implementation can be more understandable and can allow users to give developers feedback on the appropriate structure of the system.

Abstraction: The goal of the software methodology is to first what functionality is used and then how to take this abstract description and refine it into an implementable design.

What is an Object?

For understanding an object orientation, you need to understand object first. Objects are key to understand *object-oriented*. Look around yourself and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle. Object is something tangible. Object has two characteristics; state and behavior. State is a noun, attribute, a well-defined behavior and behavior is a certain operation. For example dog which is a tangible object, an object which can be touched has a state of colour, name, and breed and has behavior of barking, fetching.



Object

"Object" can be a combination of variables and functions. You may also notice that some objects, in turn, will also contain other objects. Relationships of identified objects are constructed (such as relating a person's age to a specific person). For example there are several objects like Ali, House, Car, and Tree. Relationships are built between these objects like; Ali lives in the house and Ali drives a car. In these two sentences, Ali, house and car are objects which interact with each other and form a certain relation.
Time is another example in which Hours, minutes and seconds are the objects and the operations performed on time like set Hours, set minutes and set seconds are the behavior and operations performed on the object.

Advantages:
In real world, we are surrounded by objects. Modeling picks up each thing/object in the real world which is involved in the requirement. It makes the requirement simple and easily understandable by representing simple diagrams.

What is Abstraction?

Abstraction is the process of taking out characteristics from something in order to reduce it into a set of essential characteristics. Through the process of abstraction, a programmer hides all the irrelevant data about an object in order to reduce complexity and increase efficiency.

Examples:

Ali is a PhD student and teaches BS students. Below table shows the attributes of Ali as a student and employee. When Ali is at university, then he is a **"Student"**. When he is at work, he is an **"Employee"**.

Attributes:

Name	Employee ID
Student Roll No	Designation
Year of Study	Salary
CGPA	Age

Ali has different relationships in different roles. So it boils down to what in what context we are looking at an entity/object. So if I am modelling a ***Payroll System***, I will look at Ali as an ***Employee*** **(**employee ID, name, designation, salary and age) shown in the table below.

Attributes:

Employee ID
Name
Designation
Salary
Age

If am modelling a ***Course Enrollment System***, then I will consider Ali's aspects and characteristics as a ***Student*** **(**student roll number, name, year of study, cgpa, age) shown in the table below.

Attributes:

Name
Student Roll No
Year of Study
CGPA
Age

Abstract class and abstract method.

Abstraction means putting all the variables and methods in a class which are necessary.
Abstraction is the common thing.

Example:

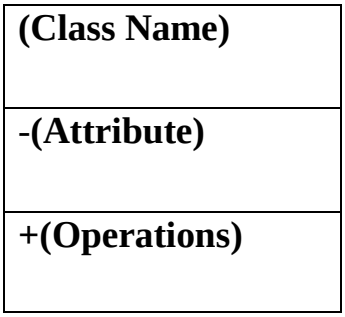
If somebody in your collage tell you to fill application form, you will fill your details like name, address, data of birth, which semester, percentage you have got etc.
If some doctor gives you an application to fill the details, you will fill the details like name, address, date of birth, blood group, height and weight.

See in the above example what is the common thing?

Age, name, address so you can create the class which consist of common thing that is called abstract class.
That class is not complete and it can inherit by other class.

Graphical Representation of Classes

The class is represented as a rectangle, divided in 3 boxes one under another. The name of the class is at the top. Next, there are the attributes of the class. At the very bottom are the operations or methods. The plus/minus signs indicate whether an attribute / operation is visible (+ means public) or not visible (- means private). Protected members are marked with #.



Normal Form

Suppressed Form

Abstraction in Programming:

A Car has Engine, wheels and many other parts. When we write all the properties of the Car, Engine, and wheel in a single class, it would look this way:

```
public class Car {

    int price;
    String name;
    String color;

    int engineCapacity;
    int engineHorsePower;

    String wheelName;
    int wheelPrice;

    void move(){
    //move forward
    }

    void rotate(){
      //Wheels method
    }

    void internalCombustion(){
      //Engine Method
    }

}
```

In the above example, the attributes of wheel and engine are added to the Car type. This will not create any kind of issues programming. But it becomes more complex when it comes to maintenance of the application.

Abstraction has three advantages:

- By using abstraction, we can separate the things that can be grouped to another type.
- Frequently changing properties and methods can be grouped to a separate type so that the main type need not undergo changes.
- Simplifies the representation of the domain models.

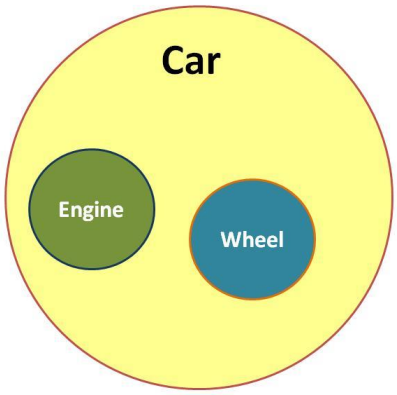
Applying the abstraction with composition, the above example can be modified as given below:

```
public class Car {

    Engine engine = new En
gine();
    Wheel wheel = new Whee
l();

    int price;
    String name;
    String color;



    void move(){
    //move forward
    }


}
```

```
public class Engine {
  int engineCapacity;
  int engineHorsePower;


  void internalCombustion(
){
    //Engine Method
  }

}
```

```
public class Wheel {
  String wheelName;
  int wheelPrice;

  void rotate(){
    //Wheels method
  }

}
```


Engine and Wheel are referred from the Car type. Whenever an instance of Car is created, both Engine and Wheel will be available for the Car and when there are changes to these Types (Engine and Wheel), changes will only be confined to these classes and will not affect the Car class.



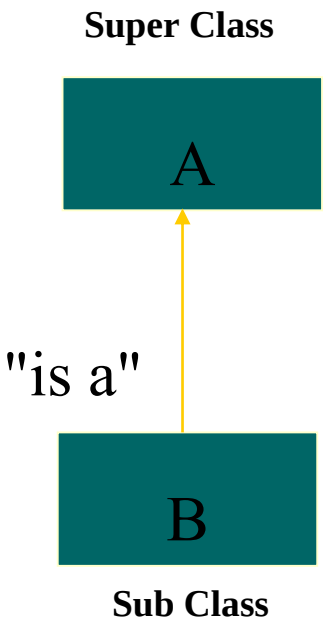
What is Inheritance?

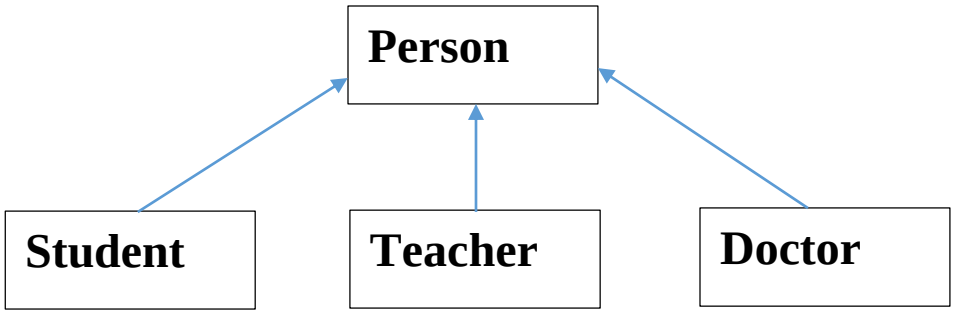
Inheritance enables new objects to take on the properties of existing objects. A class that is used as the basis for inheritance is called a *superclass* or *base class.* A class that inherits from a superclass is called a *subclass* or *derived class*. The terms *parent class* and *child class* are also acceptable terms to use respectively. The parent class is called base class and the child class is called derived class. A child inherits characteristics from its parent while adding additional characteristics of its own.

Subclasses and superclasses can be understood in terms of the **is a** relationship. A subclass **is a** more specific instance of a superclass. For example, orange is a fruit, parrot is a bird. An orange is a fruit; so it is okay to write an Orange class is a subclass of a Fruit class.

Examples:

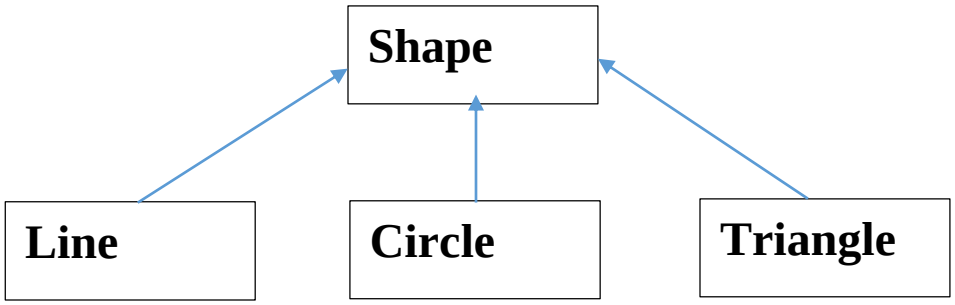
If a class B inherits from class A then it contains all the characteristics (information structure and behaviour) of class A as shown in figure below.





In the example above, Class Student, Teacher and Doctor inherits from Class Person. Each child class contain “is a” relation with parent class. Student “is a” person, Teacher “is a” person and Doctor “is a” person. Each child contains all the characteristics of parent class.

Similarly:

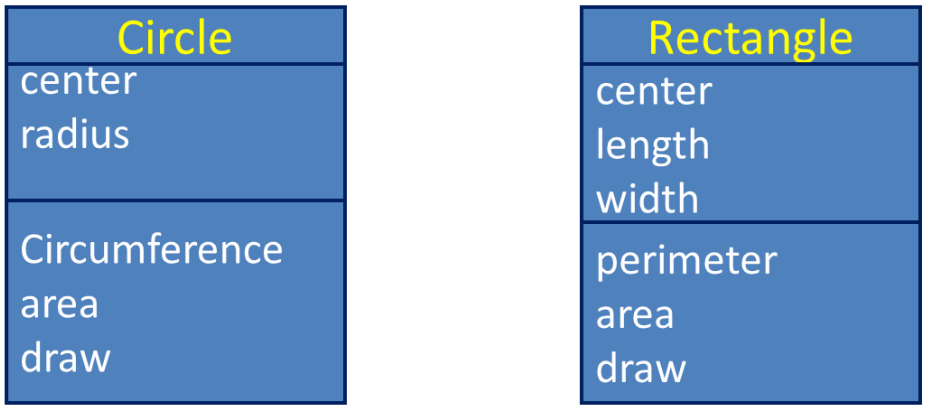


In the example above, Class Line, Circle and Triangle inherits from Class Shape. Each child class contain “is a” relation with parent class. Line “is a” Shape, Circle “is a” shape and Triangle “is a” Shape. Each child contains all the characteristics of parent class.

Generalization:

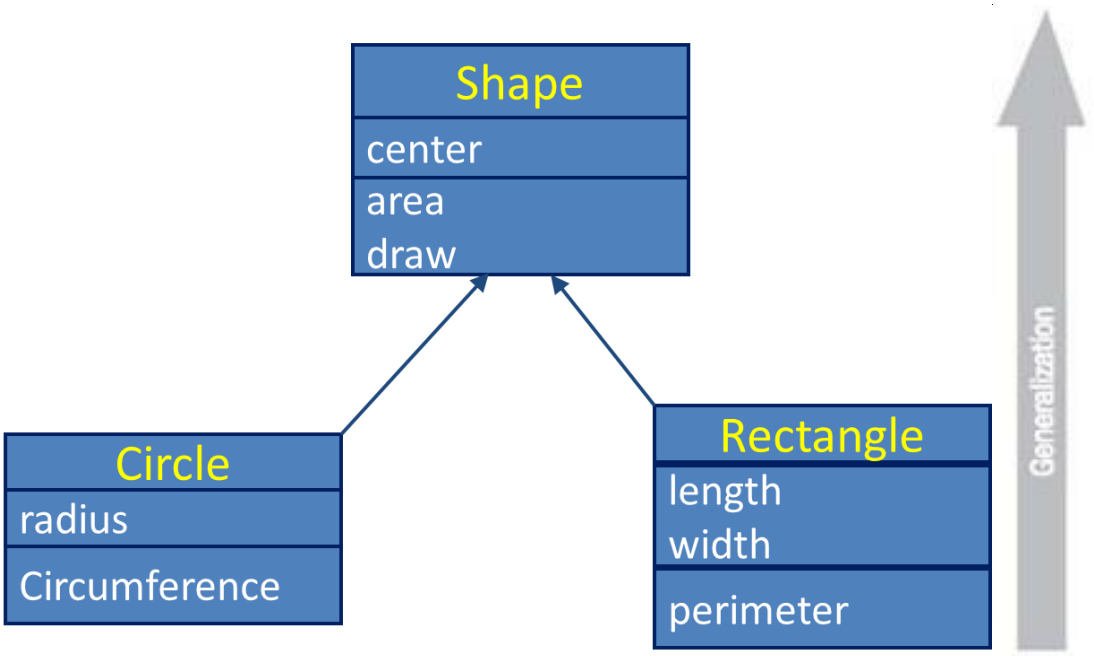
Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

The classes Circle and Rectangle partially share the same attributes. From a domain perspective, the two classes are also very similar.



During generalization, the shared characteristics are combined and used to create a new superclass Shape. Circle and Rectangle become subclasses of the class Shape.

The shared attributes are only listed in the superclass, but also apply to the two subclasses, even though they are not listed there.

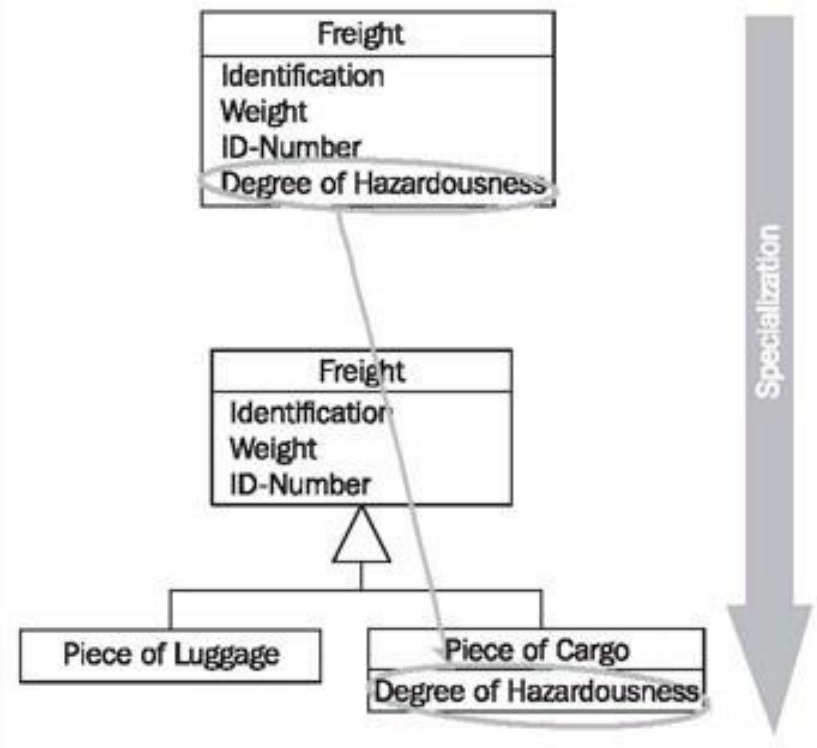


Example of Generalization

Specialization:

In contrast to generalization, specialization means creating new subclasses from an existing class. If it turns out that certain attributes, associations, or methods only apply to some of the objects of the class, a subclass can be created. The most inclusive class in a generalization/specialization is called the superclass and is generally located at the top of the diagram. The more specific classes are called subclasses and are generally placed below the superclass.

The class Freight has the attribute Degree of Hazardousness, which is needed only for cargo, but not for passenger luggage. Obviously, here two similar but different domain concepts are combined into one class. Through specialization the two special cases of freights are formed: Piece of Cargo and Piece of Luggage. The attribute Degree of Hazardousness is placed where it belongs—in Piece of Cargo. The attributes of the class Freight also apply to the two subclasses Piece of Cargo and Piece of Luggage:



Example of specialization

Specialization & Extension:

As the name suggests, class extension is concerned with adding something to a class. We can add both variables and operations.

These considerations lead us to the following definition of *class extension*.

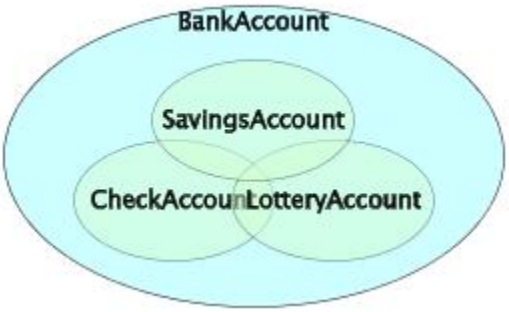
If class B is an *extension* of class A then

- B may add new variables and operations to A
- Operations and variables in A are also present in B
- B-objects are not necessarily conceptually related to A-objects

Here are the three classes given below that specialize the class BankAccount.

A specialization hierarchy of bank accounts

Figure below shows the extensions of the bank account classes. The specialized bank accounts overlap in such a way that there can exist a bank account which is both a CheckAccount, a SavingsAccount, and a LotteryAccount. An overlapping is the prerequisite for multiple specialization.

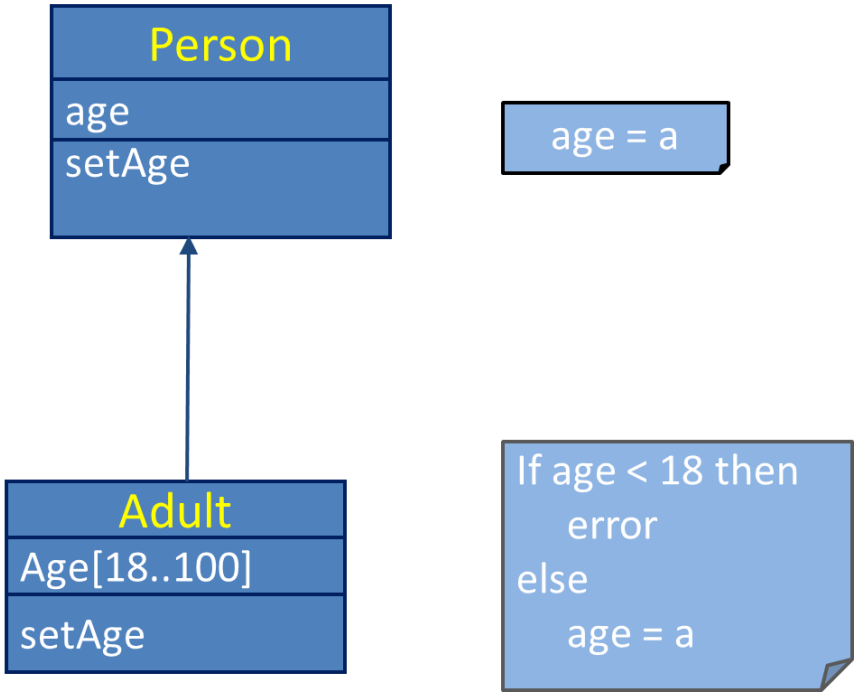


Possible extensions of the bank account classes

Specialization & Restriction:

Specialization means that derived class is behaviorally incompatible with the base class, behaviorally incompatible means that base class can't always be replaced by the derived class.

For example, new subclasses from an existing class. Through specialization, Person class is formed: Age is an attribute which is common for every person but in subclass, a condition is defined in which age of an adult should be above 18 otherwise, error will occur. So the subclass was restricted to adult age which should be above 18.



Advantages:

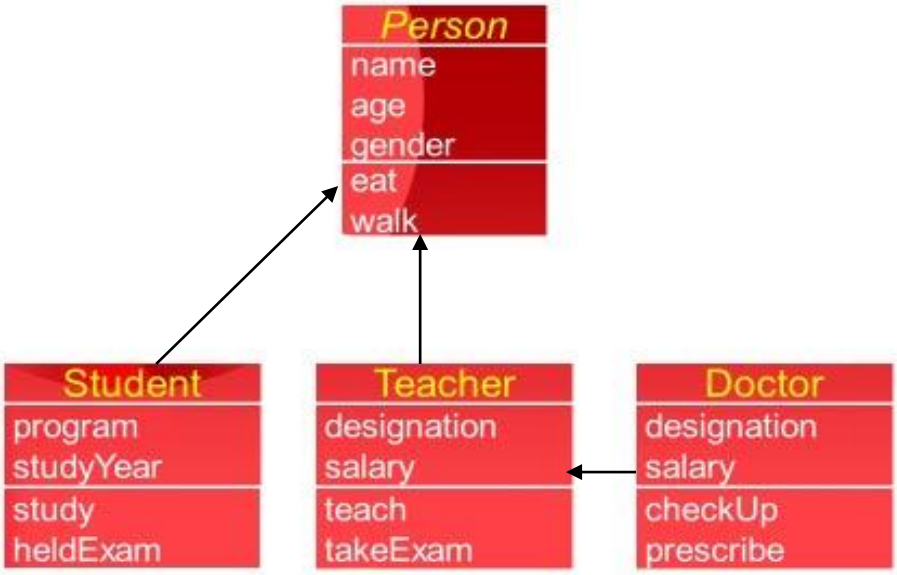
Main purpose of inheritance is reuse:

We can easily add new classes by inheriting from existing classes and can select an existing class closer to the desired functionality. We can create a new class and inherit it from the selected class. We can add to and/or modify the inherited functionality.

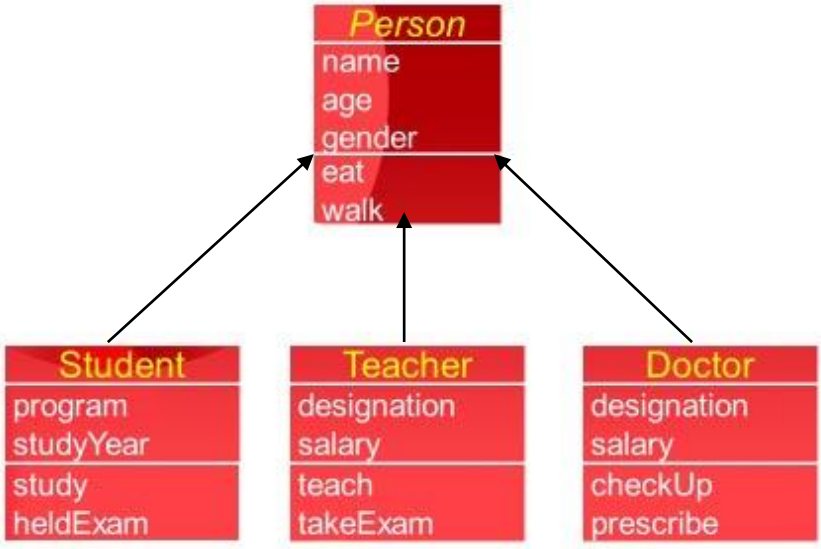
The classes Student and Teacher partially share the same attributes. From a domain perspective, the two classes are also very similar.

During generalization, the shared characteristics are combined and used to create a new superclass Person. Student and Teacher become subclasses of the class Person. The shared attributes are only listed in the superclass like name, age and gender, but also apply to the two subclasses, even though they are not listed there.

A new class of doctor has common attributes of class Teacher so the teacher class eventually become the superclass of Doctor. As Teacher is a subclass of Person class so the Doctor class will indirectly have the properties of Person class as well as shown in figure below.



If the inheritance of Doctor Class is changed from Teacher class to Person class, there will be no change in Teacher class. Doctor class will still be the same but only the functions of Teacher class will be excluded.



What is Encapsulation?

An object has to provide its users only with the essential information for manipulation, without the internal details. A Secretary using a Laptop only knows about its screen, keyboard and mouse. Everything else is hidden internally under the cover. She does not know about the inner workings of Laptop, because she doesn't need to. Therefore parts of the properties and methods remain hidden to her.

The person writing the class has to decide what should be hidden and what not. When we program, we must define as private every method or field which other classes should not be able to access.

If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding (not data security).

Thus encapsulation is said to be providing "access control" through which we can control which parts of the program can access the members of any class and thus prevent misuse. Various access controls are public, private and protected.

Real world Example of Encapsulation:-

Let's take example of Mobile Phone and Mobile Phone Manufacturer

Suppose you are a Mobile Phone Manufacturer and you designed and developed a Mobile Phone design(class), now by using machinery you are manufacturing a Mobile Phone(object) for selling, when you sell your Mobile Phone the user only learn how to use the Mobile Phone but not that how this Mobile Phone works.

Another example is the TV operation. It is encapsulated with cover and we can operate with remote and no need to open TV and change the channel.

Here everything is in private except remote so that anyone can access not to operate and change the things in TV.

Implementation:

Hide the data for security such as making the variables as private, and expose the property to access the private data which would be public.

So, when you access the property you can validate the data and set it.

Example:

```
class Demo
{
  private int _mark;

  public int Mark
  {
    get { return _mark; }
    set { if (_mark > 0) _mark = value; else _mark = 0; }
  }
 }
```

Advantages:

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

What is Information Hiding?

Information hiding concept restricts direct exposure of data. All information should not be accessible to all persons. Private information should only be accessible to its owner.

By Information Hiding we mean "Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world."

Your name and other personal information is stored in your brain we can't access this information directly. For getting this information we need to ask you about it and it will be up to you how much details you would like to share with us.

An email server may have account information of millions of people but it will share only our account information with us if we request it to send anyone else accounts information our request will be refused.

A phone SIM card may store several phone numbers but we can't read the numbers directly from the SIM card rather phone-set reads this information for us and if the owner of this phone has not allowed others to see the numbers saved in this phone we will not be able to see those phone numbers using phone.

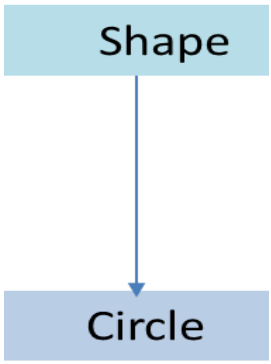
Advantages of Information Hiding

- Information Hiding makes easier for everyone to understand object oriented model.
- It is a barrier against change propagation. As implementation of functions is limited to our class and we have only given the name of functions to user along with description of parameters so if we change implementation of function it doesn't affect the object oriented model.

Types of Inheritance:

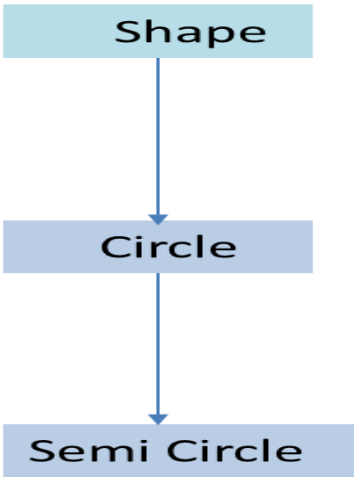
Single Inheritance:

It is the inheritance hierarchy wherein one derived class inherits from one base class. The below flow diagram shows that class Circle extends only one class which is Shape. Here Shape is a parent class of Circle and Circle would be a child class of Shape.



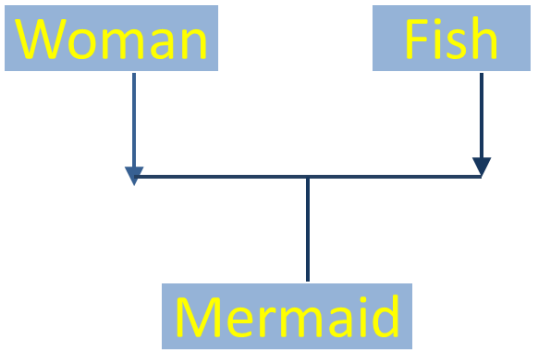
Multilevel Inheritance:

It is the inheritance hierarchy wherein subclass acts as a base class for other classes. In Multilevel inheritance there is a concept of grandparent class. As you can see in below flow diagram Semicircle is subclass or child class of Circle and Circle is a child class of Shape. So in this case class Semicircle is implicitly inheriting the properties and method of class Shape along with Circle that's what is called multilevel inheritance.

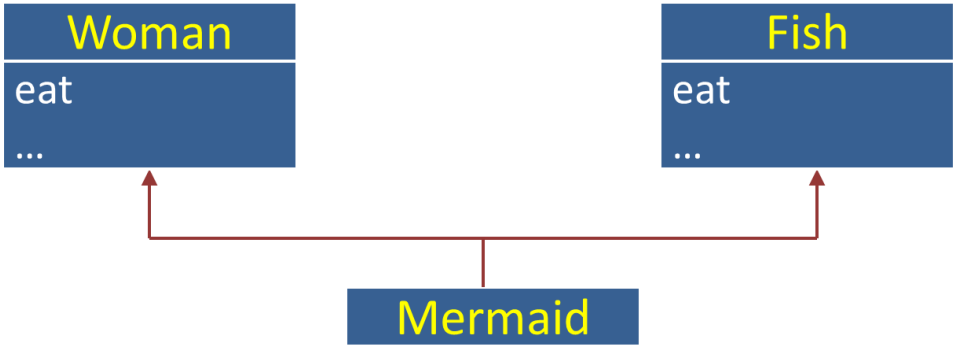


Multiple Inheritance:

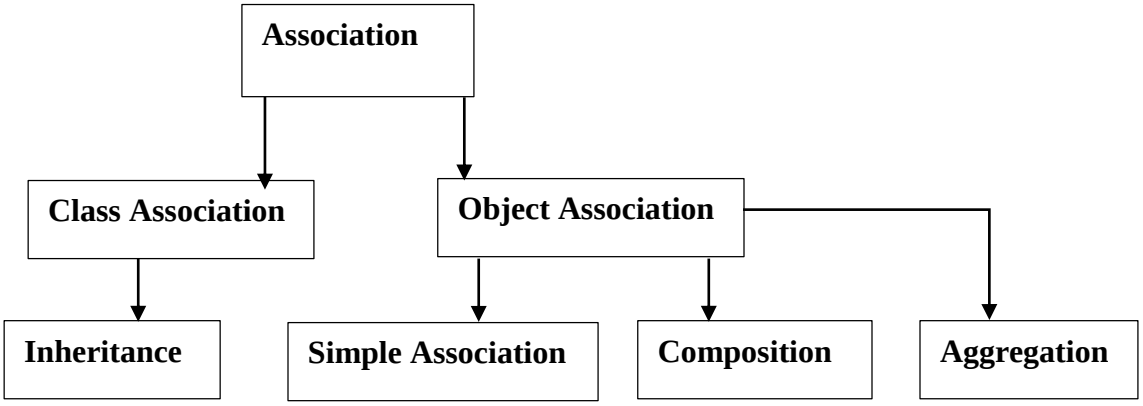
It is the inheritance hierarchy wherein one derived class inherits from multiple base class (es). The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes. In figure below, Mermaid is child class of class Woman and Fish. Mermaid will have all the properties of both Woman and Fish class.



There are problems in multiple inheritance. Both woman and Fish have the attribute of eat with their aspects so duplicate attribute of eat in Mermaid will cause complexity in choosing which property will Mermaid will follow as shown in figure below.



Links & Relationship:



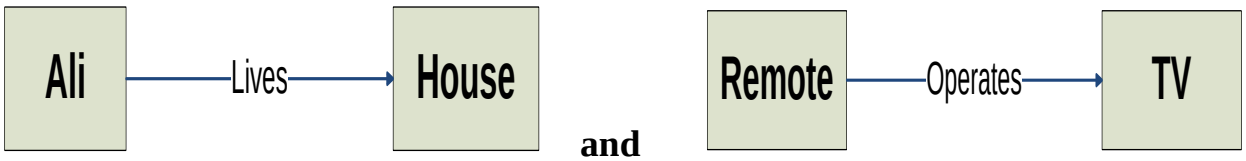
Simple Association

Association is a relationship between two objects. In other words, Two objects may depend on each other but don't interact directly (weak association). Association defines the multiplicity between objects. For example, the project management system involves various general relationships, including manage, lead, execute, input, and output between projects, managers, teams, work products, requirements, and systems. Consider, for example, how a project manager leads a team.

One Way Association:

Associations are generally assumed to be bi-directional i.e. a message can pass in both directions between objects. However in implementation this doesn't have to be the Case as shown in the example bellow:

Single directional arrow shows the message conveying from one object to other. For example: Ali lies in house but house lives in Ali will be incorrect. Moreover, Remote operates TV and not TV operates Remote so these both are one way Associations.

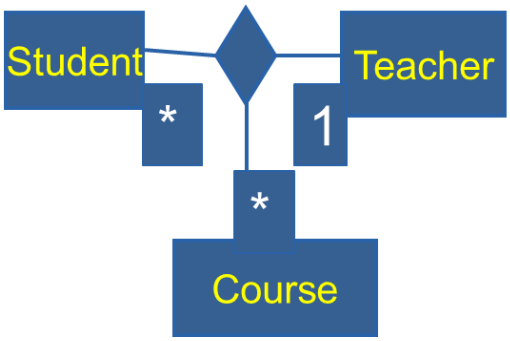


Two Way Association:

This type of association is bi-directional i.e. a message can pass in both directions between objects. It is usually donated by a single straight line or an arrow on both sides of the objects. For example, Employee works for company and the company employs many employees.

Ternary Association

Any association may be drawn as a diamond with a solid line for exactly 3 associations end connecting the diamond. For example, Teacher can teaches many courses, Many Students can enroll many courses, each teacher can teaches many students.



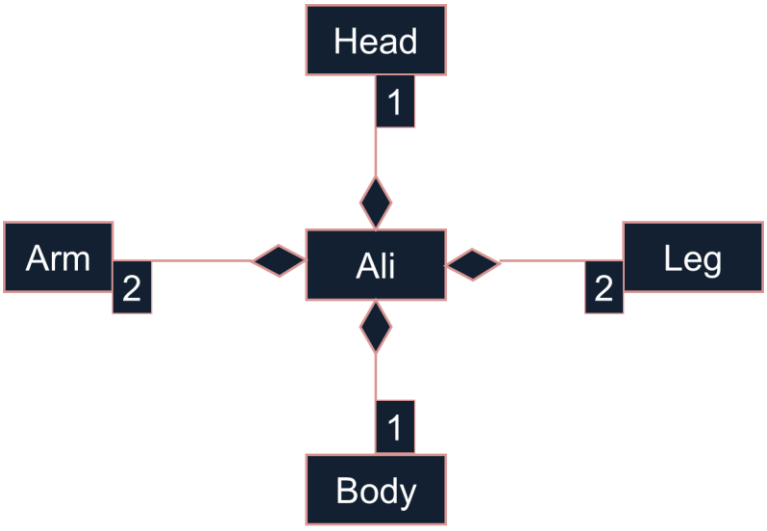
N-ary association with more than two ends can only be drawn this way.

Composition:

Composition is specialized form of Aggregation. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted. Let's take again an example of relationship between House and Rooms. House can contain multiple rooms - there is no independent life of room and any room cannot belong to two different houses. If we delete the house - room will automatically be deleted. Let's take another example relationship between Questions and Options. Single questions can have multiple options

and option cannot belong to multiple questions. If we delete questions options will automatically be deleted.

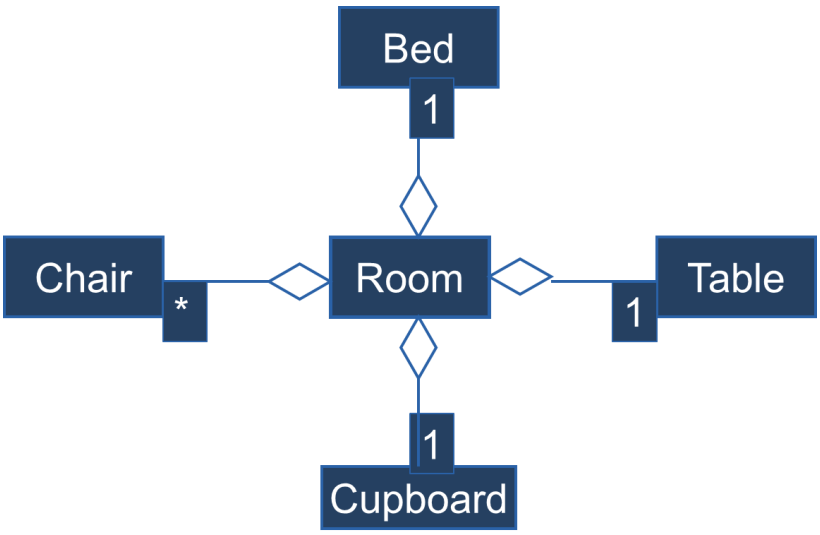
This is a strong type of relation because composed objects become the part of the composer. For example, Ali is made of 1 Head, 2 Arm, 2 Leg and 1 Body. These four classes cannot exist independently. If Ali is deleted, all four classes will automatically be removed because these classes are dependent on Ali.



Aggregation:

Aggregation is a specialized form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object. Take an example of Department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department teacher object will not be destroyed. We can think about it as a "has-a" relationship. Aggregation is weaker relationship, because aggregate object is not a part of the container, they can exist independently. Aggregation is represented by a line with unfilled-diamond head towards the container. Direction between them specified which object contains the other object.

In the example below, Room has 1 bed, 1 table, 1 cupboard and many chairs. Furniture is not the intrinsic part of the composer. They are weak aggregation and can exist independently. Furniture can shift to other room so they can exist independent of a particular room.



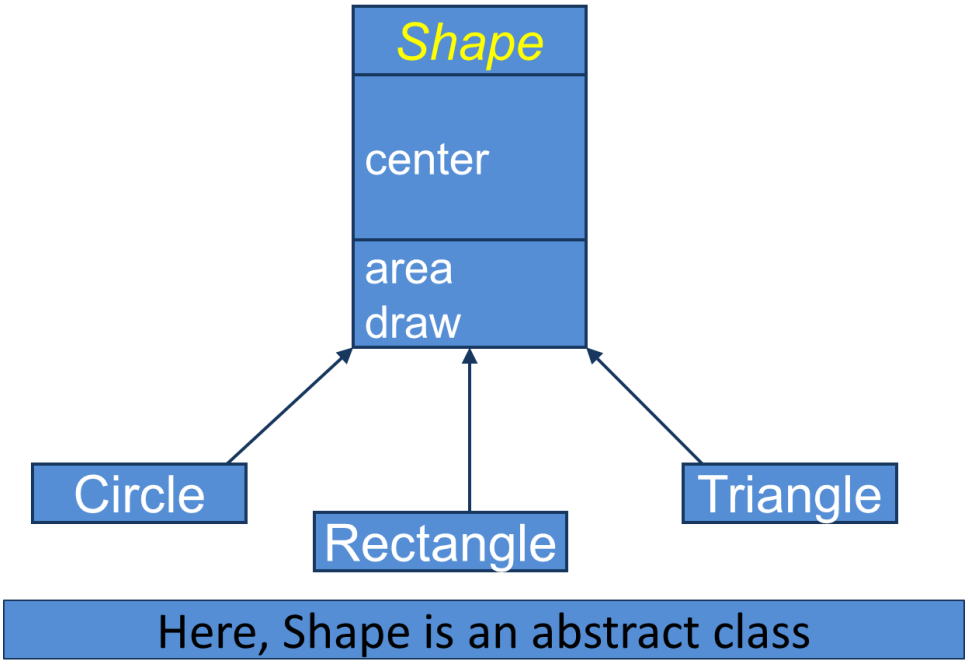
Take another example of plant and garden. They are weak aggregation and can exist independently. Plant is not the intrinsic part of the garden and can exist independently. They can be planted to other garden and it is not dependent to the particular garden.

Abstract Class:

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. An abstract method is a method that is declared without an implementation. If a class includes abstract methods, then the class itself must be declared abstract. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

For example, you can draw circles, rectangles, triangles and many other shapes. These objects all have certain states (for example: center, orientation, line color, fill color) and behaviors (for example: area, rotate, draw) in common. Some of these states and behaviors are the same for all shapes (for example: center, area, and draw). All shapes must be able to draw themselves; they

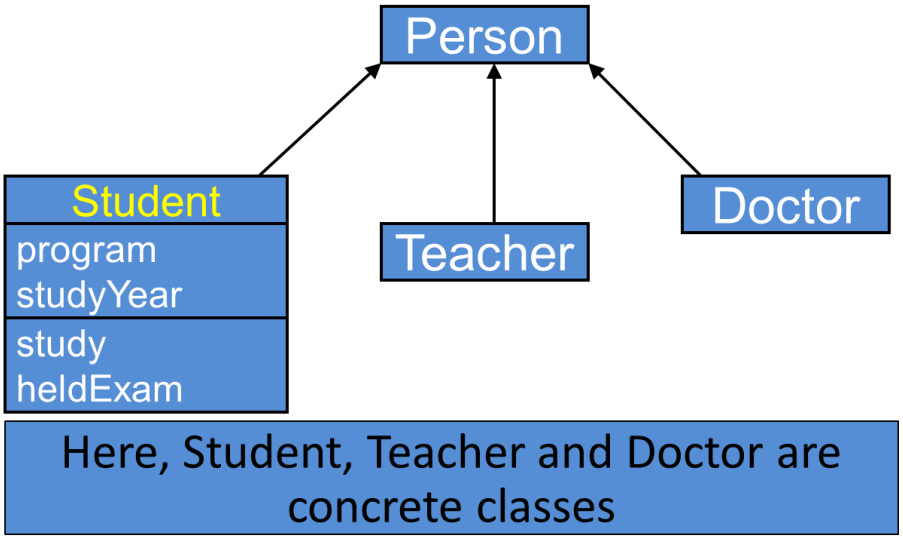
just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, Shape) as shown in the following figure. Classes Rectangle, triangle and Circle Inherit from Shape.



Concrete Class:

A concrete class has concrete methods, i.e., with code and other functionality. This class may extend an abstract class or implements an interface. The derived class is expected to provide implementations for the methods that are not implemented in the base class. A derived class that implements all the missing functionality is called a concrete class.

In example below, Student, Teacher and Doctor are concrete classes and Person is the Abstract class.



Interface (Java):

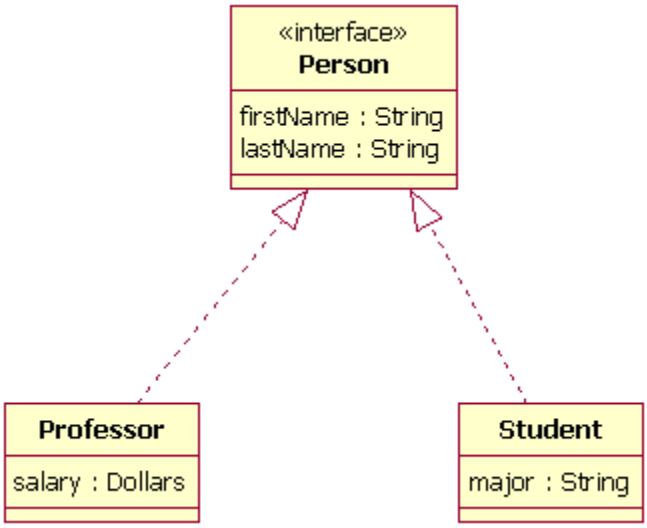
An interface is an elegant way of defining the public services that a class will provide without being bogged down by implementation details.

An interface is used to define the public services of a class. The interface provides no implementation details. Think of an interface as a business contract between two parties. The class implementing the interface agrees to provide the services defined in that interface to other classes. The other classes calling on the public services agree to abide by the semantics of the interface.

In the diagram below, both the Professor and Student classes implement the Person interface and do not inherit from it. We know this for two reasons:

1) The Person object is defined as an interface — it has the "«interface»" text in the object's name area, and we see that the Professor and Student objects are class objects because they are labeled according to the rules for drawing a class object.

2) We know inheritance is not being shown here, because the line with the arrow is dotted and not solid. As shown in Figure, a dotted line with a closed, unfilled arrow means realization (or implementation); a solid arrow line with a closed, unfilled arrow means inheritance.



Overloading:

Overloading occurs when two or more methods in one class have the same method name but different parameters.

An appropriate example would be a Print(object O) method. In this case one might like the method to be different when printing, for example, text or pictures. The two different methods may be overloaded as Print(text_object T); Print(image_object P). If we write the overloaded print methods for all objects our program will "print", we should not have to worry about the type of the object, and the correct function call again, the call is always: Print(something).

Rules:

The overloaded function must differ by data types...

The same function name is used for various instances of function call.

Programming Example:

In the example below variables are added with two datatypes, int and double. Through datatype int, out will be in real numbers and through double, the output will be in floating numbers as shown below. The method is same of addition in both case but from different datatypes, the output is different.

```
#include
#include
void add(int x, int y);
void add(double x, double y);
int main()
{
clrscr();
add(10,20);
add(10.4,20.4);
return(0);
}
void add(int x, int y)
{
cout
}
void add(double x,double y)
{
cout
}
```

Run output

30

30.8

Overriding:

Overriding means having two methods with the same method name and parameter. One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

One of the simplest example – Here Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the method eat().

```
class Human{
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    obj.eat();
  }
}
```

Output:

Boy is eating.

Here is another example of overriding. The dog variable is declared to be a Dog. During compile time, the compiler checks if the Dog class has the bark() method. As long as the Dog class has the bark() method, the code compilers. At run-time, a Hound is created and assigned to dog. Dog is referring to the object of Hound, so it calls the bark() method of Hound.

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}

public class OverridingTest{
    public static void main(String [] args){
        Dog dog = new Hound();
        dog.bark();
    }
```

Output:

Bowl

The main advantage of overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class(base class).

Polymorphism:

"Poly" means "many" and "morph" means "form". Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms.

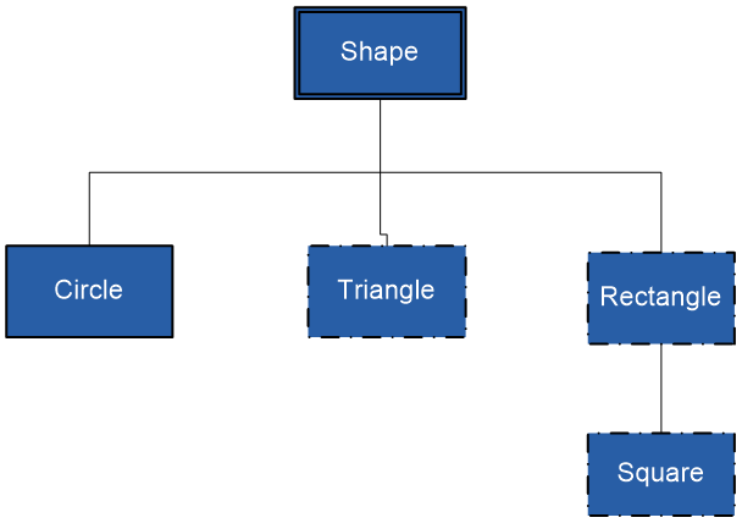
There are two types of polymorphism one is compile time polymorphism and the other is run time polymorphism. Compile time polymorphism is functions and operators overloading. Runtime time polymorphism is done using inheritance and virtual functions. Here are some ways how we implement polymorphism in Object Oriented programming languages.

Compile time polymorphism -> Operator Overloading, Function Overloading

Run time polymorphism -> Interface and abstract methods, Virtual member functions.

An example would be:

A "Shape" class can be a part of an inheritance hierarchy where derived classes are "Circle", “Triangle” and "Rectangle". Derived from "Rectangle" could be "Square",



Now, using such an example, it is true that any object below in a hierarchy is also something that is directly up in the hierarchy. Hence, Square “is a" Rectangle, and Rectangle "is a" Shape. Also, Square "is a" Shape.

Each of these classes will have different underlying data. A point shape needs only two co-ordinates (assuming it's in a two-dimensional space of course). A circle needs a center and

radius. A square or rectangle needs two co-ordinates for the top left and bottom right corners (and possibly) a rotation.
And, by making the class responsible for its code as well as its data, you can achieve polymorphism. In this example, every class would have its own Draw() function and the client code could simply do: shape.Draw().
Hence, using pointers of Base classes (higher in an inheritance hierarchy) can be assigned to objects of derived classes and can be used in a unified manner with the use of virtual functions. Hence, Polymorphism.

(The plus "+" operator example used above would not be correct, as that is actually an overloaded operator (in the case the last poster presumed) and not precisely polymorphism).

Dynamic Binding:

Late binding means the binding occurs at runtime, based on the type of the object. Late binding is also called dynamic binding or runtime binding. When a language implements late binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function. In the case of a compiled language, the compiler doesn't know the actual object type, but it inserts code that finds out and calls the correct function body.

An example of polymorphism is the technique by which a reference that is used to invoke a method can actually invoke different methods at different times depending on what it is referring to at the time. This can be illustrated by the following example.

In the for loop, the statement all[i].toString() will either invoke the definition in Student or MScStudent depending on what type of object the polymorphic reference all[i] is pointing to at the time.

```
public class TestPoly2
{
   public static void main(String [] args)
{
       Student [] all= new Student[3];
       all[0]= new Student("kate");
       all[1]= new MScStudent("mike");
       all[2]= new Student("Jane");

       for (int i=0;i<3;i++)
           System.out.println(all[i].toString());
   }
}
```

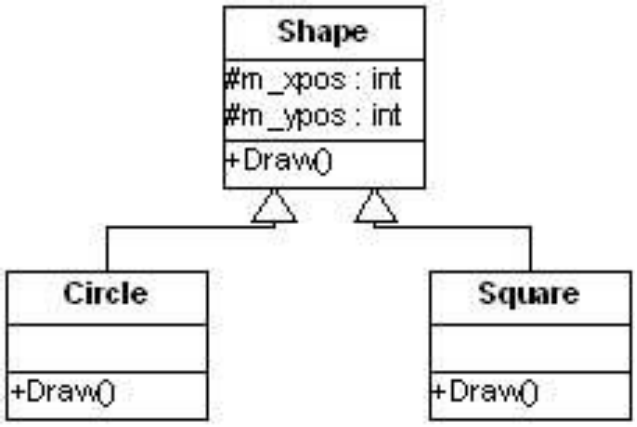
Up-casting:

In up-casting we convert a smaller datatype into a larger datatypes. Since converting a smaller dataytype into a larger datatype does not cause any information loss therefore up-casting is implicit (i.e. it occurs automatically and we do not have to write any extra piece of code)

In primitive datatypes such as int, float, double upcasting occurs when we convert a smaller data type (in terms of size (bytes)) into a large data type. For example; converting int (4 bytes) to long (8 bytes) or float (4 bytes) to double(8 bytes). This has been shown in the following code fragment.

```
int i = 4;
double d ;
d = i;
```


The diagram below is our polymorphism example.



Consider the following code:

Shape s = new Circle(100, 100);

We have cast Circle to the type Shape. This is possible, because Circle has been derived from Shape and you expect all methods and properties of Shape to exist in Circle. Executing the Draw method by doing s.Draw() gives the following output:

Drawing a CIRCLE at 100,100

If we had declared the Draw method in Circle as follows, public new void Draw() the output would have been:

Drawing a SHAPE at 100,100

As we have already mentioned, marking the method with new, tells the compiler that we are not overriding the base class implementation of the method.

So why is this called up-casting?

Consider the diagram above. From Circle, we are moving up the object hierarchy to the type Shape, so we are casting our object "upwards" to its parent type.

Up-casting is implicit and is safe. By mentioning “safe”, we can happily cast Circle to Shape and expect all the properties and methods of Shape to be available.

Down-Casting:

In down-casting we convert a larger datatype into a smaller datatype. Since converting a larger dataytype into a smaller datatype may cause an information loss therefore down-casting is explicit (i.e. it does not occur automatically and we have to tell compiler that we want to downcast otherwise it won't let us compile the code).

Since double is a bigger type, conversion is automatic but when we convert from double to int there can be a loss of information and therefore explicit casting is required (i,e, we are telling the compiler that we know what we are going to do, so do it.)

```
double d = 4;
int i ;
i =(int)d;
```

To help us better understand down-casting, we are going to add a new method to our Circle class. This will be a simple method called FillCircle.

```
public void FillCircle()
{
   Console.WriteLine("Filling CIRCLE at {0},{1}", m_xpos, m_ypos);
}
```

Using the example from up-casting, we know that we are able to write the following:

```
Shape s = new Circle(100, 100);
```

We are then free to call the Draw method. Having added the FillCircle method to our Circle class we are not able to call this method by doing the following:

```
s.FillCircle ();
```

Because we have cast Circle to the type Shape, we are only able to use methods found in Shape, that is, Circle has inherited all the properties and methods of Shape. If we want to call FillCircle,

we need to down-cast our type to Circle. In down-casting, we are simply moving down the object hierarchy, from Shape down to Circle. The code for doing this is quite simple:

```
Circle c;
c = (Circle)s;
```

Simply, we are declaring c as the type Circle and explicitly casting s to this type. We are now able to call the FillCircle method by doing the following:

```
c.FillCircle();
```

This gives us the following output:

```
Drawing a CIRCLE at 100,100

Filling CIRCLE at 100,100
```

We could also write ((Circle)s).FillCircle() reducing the lines of code needed to down-cast our type and call the required method.

Disadvantages:

Down-casting is potentially unsafe, because you could attempt to use a method that the derived class does not actually implement. With this in mind, down-casting is always explicit, that is, we are always specifying the type we are down-casting to.

Module 2: Basics of Java Programming

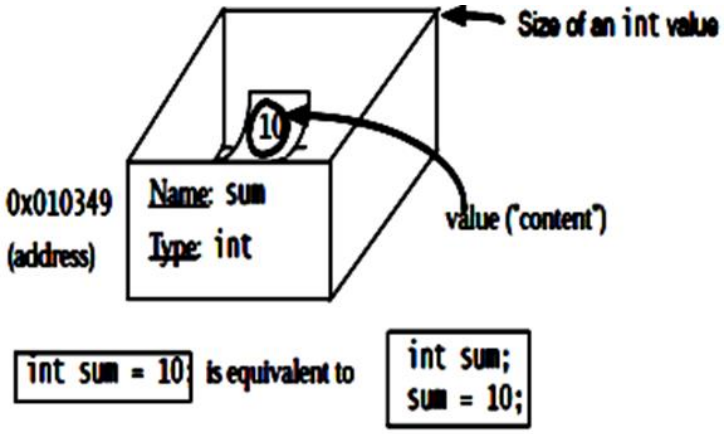
There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

Local variables:

Local variables are declared in methods, constructors, or blocks.

- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.



Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . ObjectReference.VariableName.

Class/static variables:

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name ClassName.VariableName.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms.

- A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- An implementation its implementation is known as JRE (Java Runtime Environment).
- Runtime Instance Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

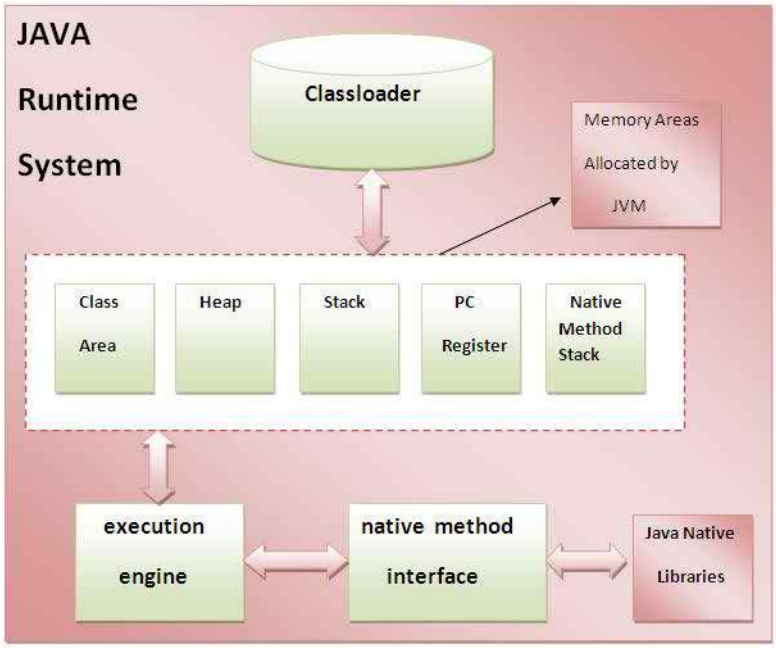
JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.

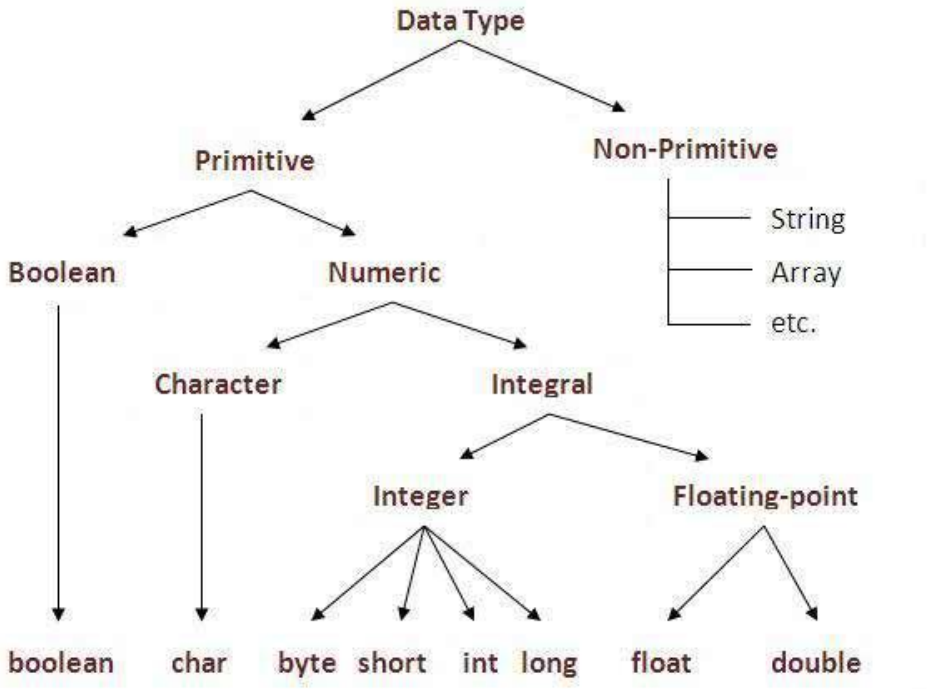
Jvm Internal



Data Types in Java

In java, there are two types of data types

- primitive data types
- non-primitive data types



Java - Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator and Example
+ (Addition)
Adds values on either side of the operator
Example: A + B will give 30

- (Subtraction)
Subtracts right hand operand from left hand operand
Example: A - B will give -10

* (Multiplication)
Multiplies values on either side of the operator
Example: A * B will give 200

/ (Division)
Divides left hand operand by right hand operand
Example: B / A will give 2

% (Modulus)
Divides left hand operand by right hand operand and returns remainder
Example: B % A will give 0

++ (Increment)
Increases the value of operand by 1
Example: B++ gives 21

-- (Decrement)

Decreases the value of operand by 1

Example: B-- gives 19

Module 3: (Program Control Flow)

Control Flow:

The statements inside your source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

Selection Statements:

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true.

For example, the Bicycle class could allow the brakes to decrease the bicycle's speed only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes()
{
   // the "if" clause: bicycle must be moving
   if (isMoving)
{
       // the "then" clause: decrease current speed
       currentSpeed--;
   }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes()
{
   if (isMoving)
{
       currentSpeed--;
   }
 else
{
       System.err.println("The bicycle has already stopped!");
   }
}
```

Loop:

A loop is a sequence of instruction s that is continually repeated until a certain condition is reached.

For Loop:

A for loop iterates over elements of a sequence. A variable is created to represent the object in the sequence. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment)
{
   statement(s)
}
```

For example,

```
x = [100,200,300]
for i in x:
    print i
```

This will output:

100

200

300

The for loop loops over each of the elements of a list or iterator, assigning the current element to the variable name given. In the example above, each of the elements in x is assigned to i.

While Loop:

The loop construct, found in nearly all procedural languages, that executes one or more instructions (the "loop body") repeatedly so long as some condition evaluates to true. In contrast to a repeat loop, the loop body will not be executed at all if the condition is false on entry to the while.

For example, in C, a while loop is written

while () ;

Where is any expression and is any statement, including a compound statement within braces "..".

Its syntax can be expressed as:

```
while (expression)
{
    statement(s)
}
```


For example:

```
x= 5
while x > 0:
    print x
    x = x - 1
```

Will output:

5

4

3

2

1

While loops can also have an 'else' clause, which is a block of statements that is executed (once) when the while statement evaluates to false. The break statement inside the while loop will not direct the program flow to the else clause. For example:

```
x = 5
y = x
while y > 0:
    print y
    y = y - 1
else:
    print x
```

T

5

4

3

2

1

5

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while
(expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
   public static void main(String[] args){
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count < 11);
   }
}
```


Nested Statement:

```
#include <stdio.h>
 int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   /* check the boolean condition */
   if( a == 100 )
{
      /* if condition is true then check the following */
      if( b == 200 )
{
         /* if condition is true then print the following */
         printf("Value of a is 100 and b is 200\n" );
      }
   }
    printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );
    return 0;
}
```

If variable “a” is equal to 100 then it will proceed to next statement. It will then check the next condition, if “b” is equal to 200 then it will print the statement and the output is given below.

When the above code is compiled and executed, it produces the following result −

Value of a is 100 and b is 200
Exact value of a is: 100
Exact value of b is: 200

Program code (containing a logical error):

```
if (gas_tank < 0.75)          // 1st if-statement
    if (gas_tank < 0.25)      // 2nd if-statement
        System.out.println("Low on gas!");
else
    System.out.println("At least 3/4 tank. Go on!");
```

If gas tank is less than 3/4 full then the system will check the next statement. Then check if it is less than 1/4 full, it will then print "Low on gas!", otherwise it will print "At least 3/4 tank. Go on!"

A Local Block:

Statements which are enclosed in left brace ({) and the right brace (}) forms a local Block. Local Block can have any number of statements. Branching Statements or Conditional Statements, Loop Control Statements such as if, else, switch, for, while forms a local block. These Statements contain braces, so the portion of code between two braces would be considered a local block. Variables declared in a local block have local scope i.e they can be accessed within the block only.


```
#include<stdio.h>
int x = 40 ;        // Scope(Life) : Whole Program
int main()
{
int x = 10 ;        // Scope(Life) : In main
  {
  x = 30 ;
  printf("%d",x);
  }
printf("%d",x);
}
```

Output:

30 10

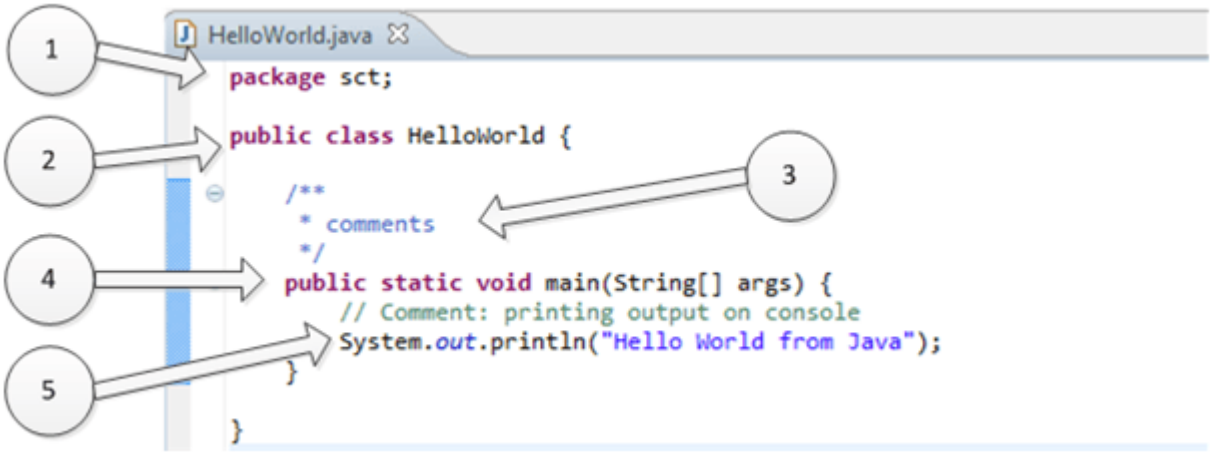
Statement terminator:

Semicolon (;) is used to terminate a statement in Java. A compound statement does not need a semicolon to terminate it. A semicolon alone denotes the empty statement that does nothing. The semicolon terminates a statement. Although by convention you should always end your statements with semicolons, they are not strictly required in ActionScript. The interpreter attempts to infer the end of a statement if the semicolon is omitted. For example:

```
// These are preferred
   var x = 4;
   var y = 5;
   // But these are also legal
   var x = 4
   var y = 5
```


Java Program Structure

Let's use example of HelloWorld Java program to understand structure and features of class. This program is written on few lines, and its only task is to print "Hello World from Java" on the screen. Refer the following picture.



1. "package sct":

It is package declaration statement. The package statement defines a name space in which classes are stored. Package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in program. It must be first line of your program or you can omit it.

2. "public class HelloWorld":

This line has various aspects of java programming.

a. public: This is access modifier keyword which tells compiler access to class.

b. class: This keyword used to declare class. Name of class (HelloWorld) followed by this keyword.

3. Comments section:

We can write comments in java in two ways.

a. Line comments: It start with two forward slashes (//) and continue to the end of the current line. Line comments do not require an ending symbol.
b. Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/).Block comments can also extend across as many lines as needed.

4. "public static void main (String [ ] args)":

Its method (Function) named main with string array as argument.
a. public : Access Modifier
b. static: static is reserved keyword which means that a method is accessible and usable even though no objects of the class exist.
c. void: This keyword declares nothing would be returned from method. Method can return any primitive or object.
d. Method content inside curly braces. { } asdfla;sd

5. System.out.println("Hello World from Java") :

a. System:It is name of Java utility class.
b. out:It is an object which belongs to System class.
c. println:It is utility method name which is used to send any String to console.
d. "Hello World from Java":It is String literal set as argument to println method.

Download and Install JDK:

To develop and run any java program you need to install JDK in your system.
After selecting the Windows platform and clicking the Download button you'll see a Login for Download screen. Click Save File on the pop-up screen (the file name depends on the version of JDK). Once download completes you can start installation.

Sentinel Controlled Loop:

The type of loop where the number of execution of the loop is unknown, is termed by sentinel controlled loop. The value of the control variable differs within a limitation and the execution can be terminated at any moment as the value of the variable is not controlled by the loop. The control variable in this case is termed by sentinel variable.

Example: The following do....while loop is an example of sentinel controlled loop.

```
= = = = = =
do
{
printf(“Input a number.\n”);
scanf("%d", &num);
}
while(num>0);
= = = = = =
```

In the above example, the loop will be executed till the entered value of the variable num is not 0 or less than 0. This is a sentinel controlled loop and here the variable num is a sentinel variable.

Module 4: (Using Objects)

Object Model:

A collection of objects or classes through which a program can examine and manipulate some specific parts of its world. Such an interface is said to be the object model of the represented service or system.
For example, the Document Object Model (DOM) is a collection of objects that represent a page in a web browser, used by script programs to examine and dynamically change the page. There is a Microsoft Excel object model for controlling Microsoft Excel from another program, and the ASCOM Telescope Driver is an object model for controlling an astronomical telescope.

Object Reference

Objects can be accessed via object references. To invoke a method in an object, the object reference and method name are given, together with any arguments.

Important Aspects of Object Model:

1. Abstraction:

Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics. It provides a generalized view of your classes or object by providing relevant information.

Example:

Suppose you have 3 mobile phones as following:-

Nokia 1400 (Features:- Calling, SMS)
Nokia 2700 (Features:- Calling, SMS, FM Radio, MP3, Camera)
Black Berry (Features:-Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)

Abstract information which are Necessary and Common Information for the object "Mobile Phone" is make a call to any number and can send SMS." The abstract class for object mobile phone is as follows:

```
abstract class MobilePhone{  public void Calling(); public void SendSMS(); }
   public class Nokia1400 : MobilePhone {
   }
 public class Nokia2700 : MobilePhone{
      public void FMRadio();
```

2. Encapsulation:

Encapsulation is a way to obtain "information hiding" so, following your example, you don't "need to know the internal working of the mobile phone to operate" with it. You have an interface to use the device behavior without knowing implementation details. Encapsulation is like your bag in which you can keep your pen, book etc. It means this is the property of encapsulating members and functions.

Example:

TV operation

It is encapsulated with cover and we can operate with remote and no need to open TV and change the channel.

Realization of the Object Model:

1. Classes:

Class is a blueprint of an object that contains variables for storing data and functions to performing operations on these data. Class will not occupy any memory space and hence it is only logical representation of data.

The class is represented as a rectangle, divided in 3 boxes one under another. The name of the class is at the top. Next, there are the attributes of the class. At the very bottom are the operations or methods. The plus/minus signs indicate whether an attribute / operation is visible (+ means public) or not visible (- means private). Protected members are marked with #.

(Class Name)
-(Attribute)
+(Operations)

Normal Form

To create a class, you simply use the keyword "class" followed by the class name:

```
Class Employee
{

}
```

2. Object:

Objects are the basic run-time entities in an object oriented system. They may represent a person, a place or any item that the program has to handle.

"Object is a Software bundle of related variable and methods."
"Object is an instance of a class"

Class will not occupy any memory space. To work with the data represented by the class a variable must be created, which is called as an object.
When an object is created by using the keyword **"new"**, then memory will be allocated for the class in heap memory area, which is called as an instance and its starting address will be stored in the object in stack memory area.

Example:

```
class Employee
{

}
```

Syntax to create an object of class Employee:-

```
Employee objEmp = new Employee();
```

Class Declaration:

The class declaration component declares the name of the class along with other attributes such as the class's superclass, and whether the class is public, final, or abstract.

The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

Class declaration must contain the class keyword and the name of the class that is being defined. Thus the simplest class declaration looks like this:

```
class NameOfClass
{
      . . .
   }
```

Class declarations can include these components, in order:

1. Modifiers such as *public*, *private,* and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.

3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

Object Reference:

An "object reference" is a variable that acts as a handle/pointer/marker to a given object so that its methods and fields maybe manipulated.

We can assign value of reference variable to another reference variable. Reference Variable is used to store the address of the variable. By assigning Reference we will not create distinct copies of Objects but all reference variables are referring to same Object.

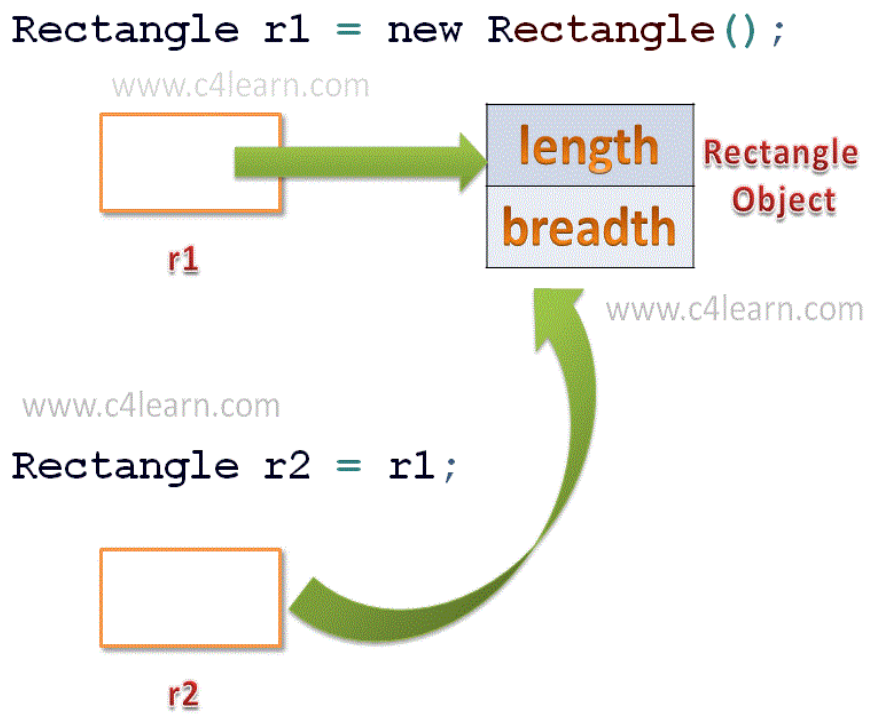
Consider This Example –

```
Rectangle r1 = new Rectangle();
Rectangle r2 = r1;
```

r1 is reference variable which contain the address of Actual Rectangle Object.

r2 is another reference variable

r2 is initialized with r1 means – "r1 and r2" both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.



[468×60]

String:

String is a contiguous sequence of symbols or values, such as a character string or a binary digit string. It represents text as a series of Unicode characters. A string is generally understood as a data type and is often implemented as an array of bytes that stores a sequence of elements, typically characters, using some character encoding.

Java uses the 16-bit Unicode character set that contains the characters from the ISOLatin-1 and the 7-bit ASCII character sets.

To standardize the storing of alphanumeric characters, the American Standard Code for Information Interchange (ASCII) was created. It defined a unique binary 7-bits number for each

storable character to support the numbers from 0-9, the upper/lower case English alphabet (a-z, A-Z), and some special characters like ! $ + - ( ) @ < > .

Since ASCII used one byte (7 bits for the character, and one of bit for transmission parity control), it could only represent 128 different characters. In addition 32 of these characters were reserved for other control purposes.
Charsets are named by strings composed of the following characters:

The uppercase letters 'A' through 'Z' ('\u0041' through '\u005a'),
The lowercase letters 'a' through 'z' ('\u0061' through '\u007a'),
The digits '0' through '9' ('\u0030' through '\u0039'),
The dash character '-' ('\u002d', HYPHEN-MINUS),
The plus character '+' ('\u002b', PLUS SIGN),
The period character '.' ('\u002e', FULL STOP),
The colon character ':' ('\u003a', COLON), and
The underscore character '_' ('\u005f', LOW LINE).

String:

String is traditionally a sequence of characters, some kind of variable. A string is generally understood as a data type and is often implemented as an array of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding.

String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change. The original String object will be deallocated, once there is no more references, and subsequently garbage-collected.

Because String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas). For example,

```
// inefficient codes
```

```
String str = "Hello";
for (int i = 1; i < 1000; ++i) {
  str = str + i;
}
```

Creating a String object:

String can be created in number of ways, here are a few ways of creating string object.

1. String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
String str1 = "Hello";
```

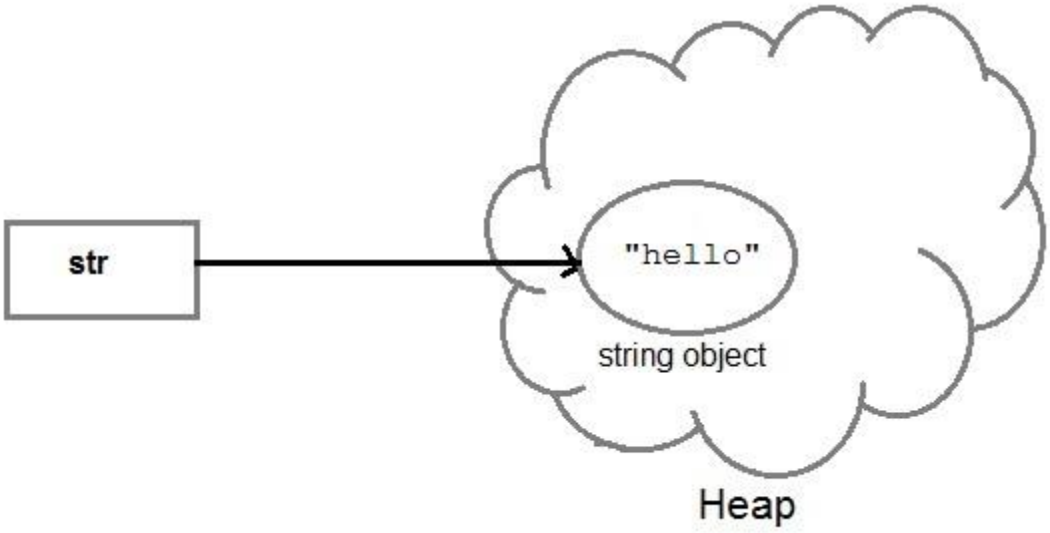
2. String str2 = new String(str1);
3. Using + operator (Concatenation)

```
String str4 = str1 + str2;
or,
String str5 = "hello"+"Java";
```

String object and How they are stored:

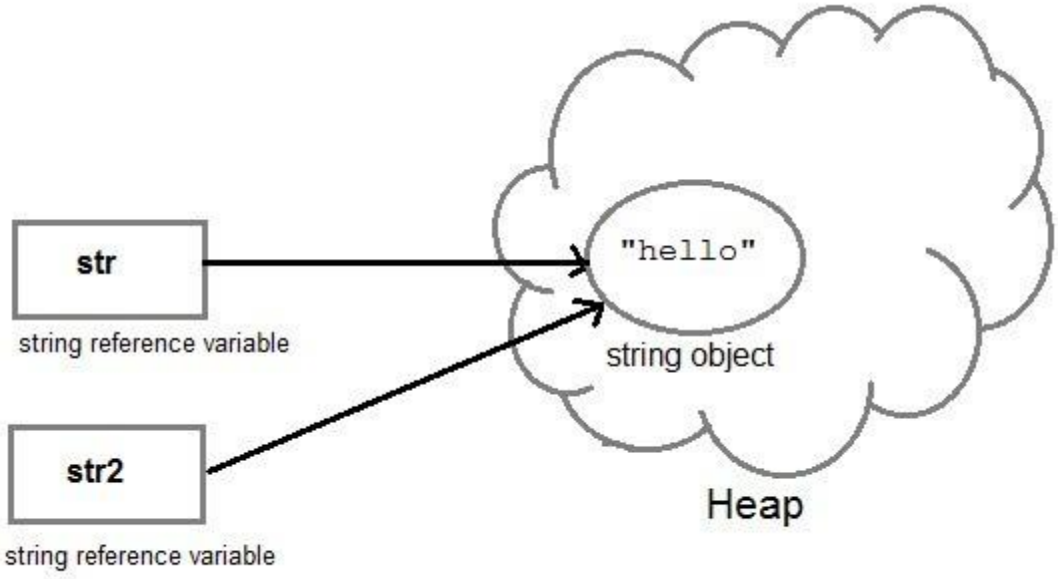
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

String str= "Hello";



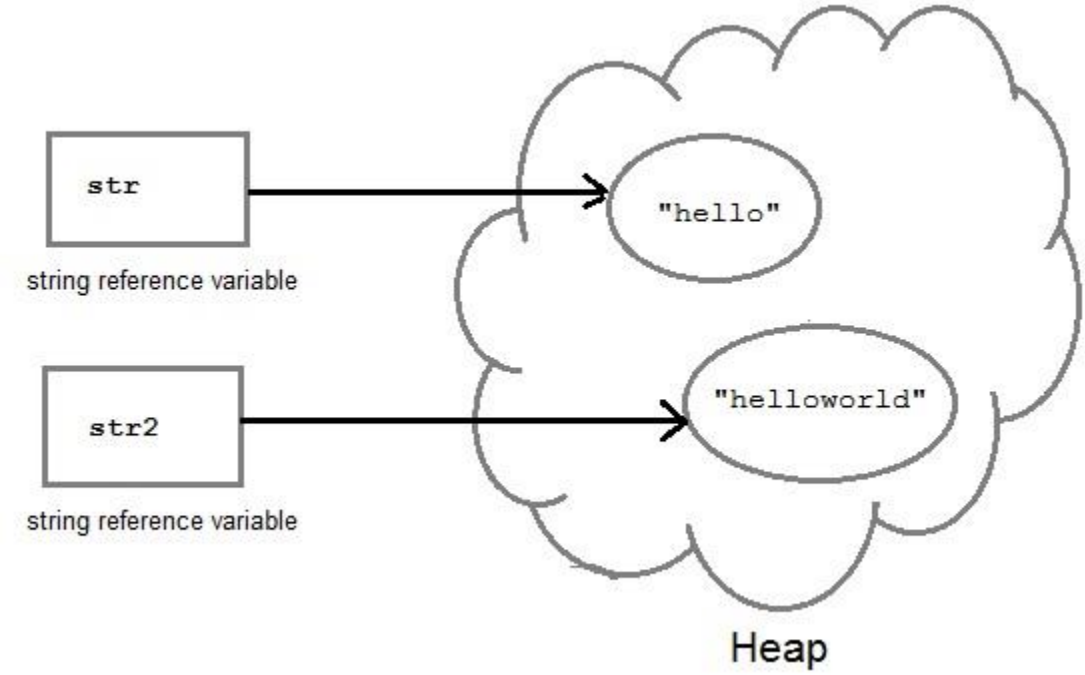
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

String str2=str;



But if we change the new string, its reference gets modified.

str2=str2.concat("world");



String Length:

Returns the length of the string, in terms of bytes. This is the number of actual bytes that conform the contents of the string, which is not necessarily equal to its storage capacity.

Example

Return the number of characters in a string:

```
var str = "Hello World!";
var n = str.length;
```

The result of n will be:

12

charAt(int index):

The method charAt(int index) returns the character at the specified index. The index value should lie between 0 and length()-1. For e.g. s.charAt(0) would return the first character of the string “s”. It throws IndexOutOfBoundsException if the index is less than zero or greater than equal to the length of the string (index<0|| index>=length()).

Concatenation:

This method concatenates the string str at the end of the current string.
For e.g. s1.contact("Hello"); would concatenate the String “Hello” at the end of the String s1.
This method can be called multiple times in a single statement like this

1. **Using concat() method**

```
String s1="Beginners";
s1= s1.contact("Book").contact(".").concat("com");
```

2. **Using + operator**

```
string str = "Rahul";
string str1 = "Dravid";
string str2 = str + str1;
string st = "Rahul"+"Dravid";
```

String Comparison:

This means that if you call the equals() method to compare 2 String objects, then as long as the actual sequence of characters is equal, both objects are considered equal. The == operator checks if the two strings are exactly the same object.

String comparison can be done in 3 ways.

1. Using equals() method
2. Using == operator
3. By CompareTo() method

1. Using equals() method

Equals() method compares two strings for equality. Its general syntax is,

boolean equals (Object str)

It compares the content of the strings. It will return true if string matches, else returns false.

```
String s = "Hell";
String s1 = "Hello";
String s2 = "Hello";
s1.equals(s2);    //true
s.equals(s1) ;   //false
```

2. Using == operator

== operator compares two object references to check whether they refer to same instance. This also, will return true on successful match.

```
String s1 = "Java";
String s2 = "Java";
String s3 = new string ("Java");
test(Sl == s2)     //true
test(s1 == s3)      //false
```

3. By compareTo() method

compareTo() method compares values and returns an int which tells if the string compared is less than, equal to or greater than th other string. Its general syntax is,

int compareTo(String str)

To use this function you must implement the Comparable Interface. compareTo() is the only function in Comparable Interface.

```
String s1 = "Abhi";
String s2 = "Viraaj";
String s3 = "Abhi";
s1.compareTo(S2);     //return -1 because s1 < s2
s1.compareTo(S3);     //return 0 because s1 == s3
s2.compareTo(s1);     //return 1 because s2 > s1
```

Comparing Object references

The two operators that can be used with object references are comparing for equality (==) and inequality (!=). These operators compare two values to see if they refer to the same object.

If the objects have the same value, and not whether two objects are a reference to the same object. For example,
(Star == Singer)
This is true only if name is a reference to the same object that "Singer" refers to. This will be false if the String in name was read from other object, even though name really does have exactly those characters in it.

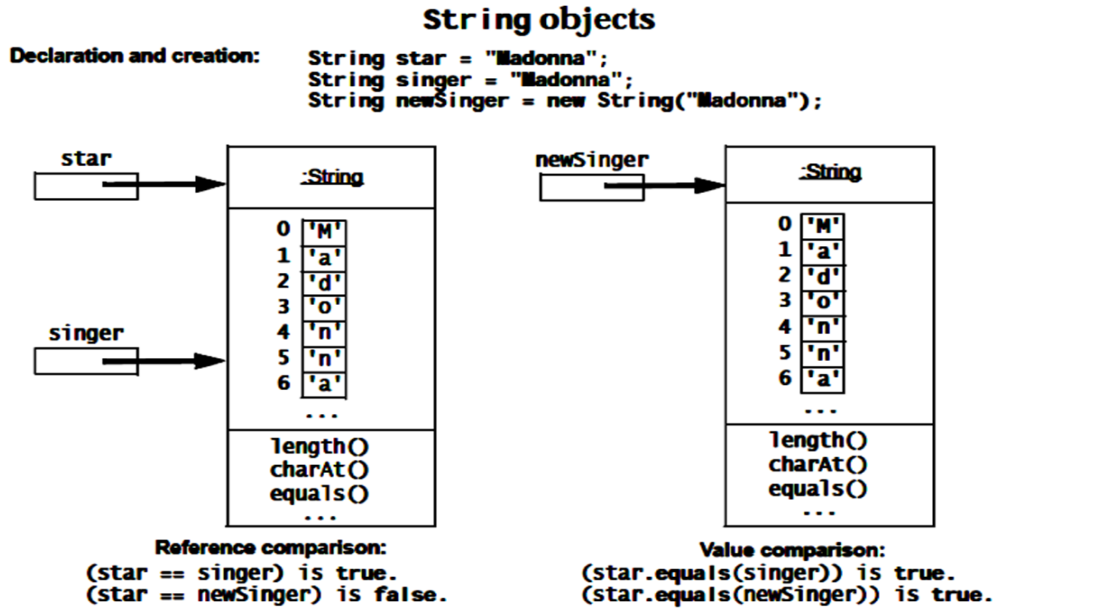
Comparing Object values

The equals() method returns a boolean value. The previous example can be fixed by writing:

```
if (Star.equals("Singer"))  // Compares values, not references.
```

(Star.equals("newSinger")) // Compares values, not references.

Because the equals() method makes a == test first, it can be fairly fast when the objects are identical. It only compares the values if the two references are not identical.



More on Control Structures

Module 5: (Primitive values as objects)

A wrapper **class** is simply a **class** that encapsulates a single, immutable value.

Integer class wraps up an **int** value
Float **class** wraps up a **float** value.

Primitive values as objects

Primitive datatypes	Corresponding wrapper classes
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

```
Integer iRef = 10; // (1) Automatic boxing, the value 10 is boxed as an Integer.
int j = iRef;      // (2) Automatic unboxing, j is assigned the value 10
                   //     from the Integer object referred by iRef.

Integer iRef = new Integer(10);   // (1) Explicit boxing.
int j = iRef.intValue();          // (2) Explicit unboxing.

Double d = j; // Compile-time error. An int value cannot be boxed as a Double.
Double d = (double) j; // OK. A double value can be boxed as a Double.
```

Autoboxing and Unboxing

Autoboxing is the automatic conversion between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing.

Passed as a parameter to a method that expects a value of the corresponding primitive type.

Assigned to a variable of the corresponding primitive type.

Here is the simplest example of autoboxing and unboxing:

Integer iRef = 10; //Automatic boxing

Int j = iRef; // Automatic Unboxing

Extending Assignment Operator:

+= assigns the result of the addition.

-= assigns the result of the subtraction.

*= assigns the result of the multiplication

/= assigns the result of the division.

%= assigns the remainder of the division.

&= assigns the result of the logical AND.

|= assigns the result of the logical OR.

^= assigns the result of the logical XOR.

<<= assigns the result of the signed left bit shift.

>>= assigns the result of the signed right bit shift.

>>>= assigns the result of the unsigned right bit shift.

Examples:

To assign the result of an addition operation to a variable:

```
//add 2 to the value of number
number = number + 2;
```

Using the assignment operator "+=" to do the same thing:

```
//add 2 to the value of number
number += 2;
```

Unary increment & decrement operators:

Increment (++) and decrement (--) operators easily add 1 to, or subtract 1 from, a variable. For example, using increment operators, you can add 1 to a variable named a like this:

```
a++;
```


An expression that uses an increment or decrement operator is a statement itself. That's because the increment or decrement operator is also a type of assignment operator because it changes the value of the variable it applies to.

An increment or decrement can also be used as operator in an assignment statement:

```
int a = 5;
int b = a--;    // both a and b are set to 4
```

Increment and decrement operators can be placed before (prefix) or after (postfix) the variable they apply to. If an increment or decrement operator is placed before its variable, the operator is applied before the rest of the expression is evaluated. If the operator is placed after the variable, the operator is applied after the expression is evaluated.

For example:

```
int a = 5;
int b = 3;
int c = a * b++;   // c is set to 15
int d = a * ++b;   // d is set to 20
```

Counter-controlled loops: for loop

The general loop construction is mostly used for counter-controlled loops where the number of iterations is known beforehand.

```
for (initialization ; condition ; increment)
  statement
```


The initialization names the loops control variable and provides its initial value, condition determines whether the loop should continue executing and increment modifies the control variable's value so that the loop-continuation condition eventually becomes false.

All three expressions in a for statement are optional. If the condition is omitted, loop-continuation condition is always true, thus creating an infinite loop. The initialization expression may be omitted if the program initializes the control variable before the loop. The increment expression may be omitted if the program calculates the increment with statements in the loop's body or if no increment is needed.

Example:

```
public class ForCounter
{
  public static void main( String[] args )
  {
    // for statement header includes initialization,
    // loop-continuation condition and increment
    for ( int counter = 1; counter <= 10; counter++ )
      System.out.printf( "%d  ", counter );

    System.out.println(); // output a newline
  } // end main
} // end class ForCounter
```

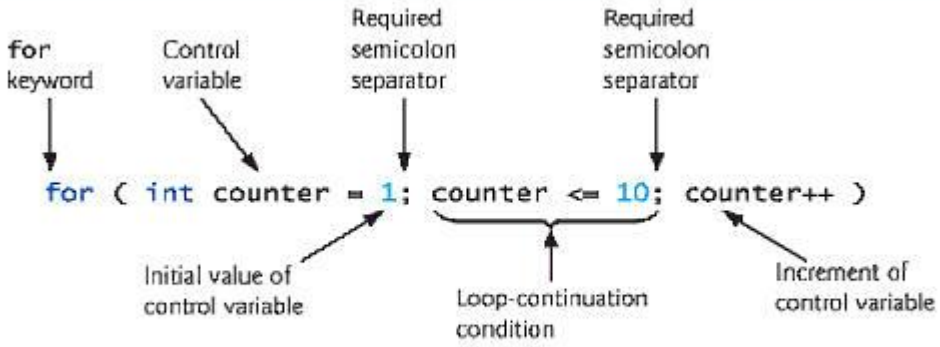



Fig. 5.3 | `for` statement header components.

A common logic error with counter-controlled repetition is an off-by-one error.
For example, if the above condition is changed to counter < 10, the loop would iterate only nine times.

Nested Loop:

Nested repetition is when a control structure is placed inside of the body or main part of another control structure. Put any construct inside of any other construct. However, you may not have them partially overlap each other.

Example: Print the multiplication tables from 1 to 10.

Example: Print the multiplication tables from 1 to 10.

```
for (int i = 1; i <= 10; ++i) {
    System.out.println("Multiplication table for " + i);
    for (int j = 1; j <= 10; ++j) {
        System.out.println(i + "\tx\t" + j + "\tis\t" + (i*j)); // (1)
    }
}
```

Module 6: (Arrays)

Arrays:

An array is an object containing a list of elements of the same data type.

We can create an array by:

- Declaring an array reference variable to store the address of an array object.
- Creating an array object using the new operator and assigning the address of the array to the array reference variable.

Here is a statement that declares an array reference variable named dailySales:

```
double[ ] dailySales;
```

The brackets after the key word double indicate that the variable is an array reference variable. This variable can hold the address of an array of values of type double. We say the data type of dailySales is double array reference.

The second statement of the segment below creates an array object that can store seven values of type double and assigns the address of the array object to the reference variable named dailySales:

```
double[ ] dailySales;

dailySales = new double[7];
```

The operand of the new operator is the data type of the individual array elements and a bracketed value that is the array size declarator. The array size declarator specifies the number of elements in the array.

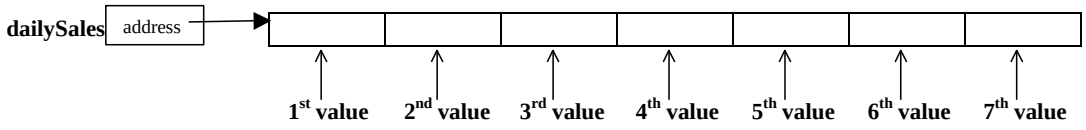
It is possible to declare an array reference variable and create the array object it references in a single statement.

Here is an example:

double[ ] dailySales = new double[7];

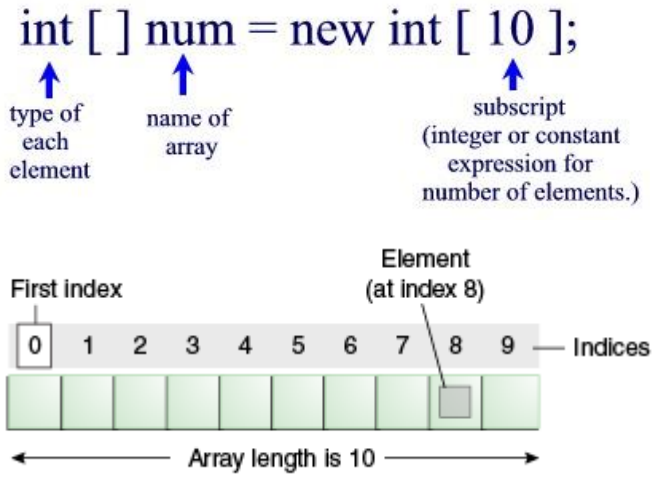
The statement below creates a reference variable named dailySales and an array object that can store seven values of type double as illustrated below:

double[ ] dailySales = new double[7];



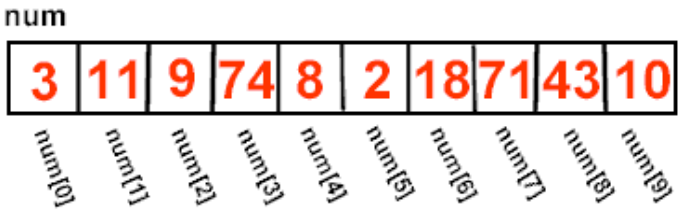
Initializing an Array:

Declare an array of 10 integer values.



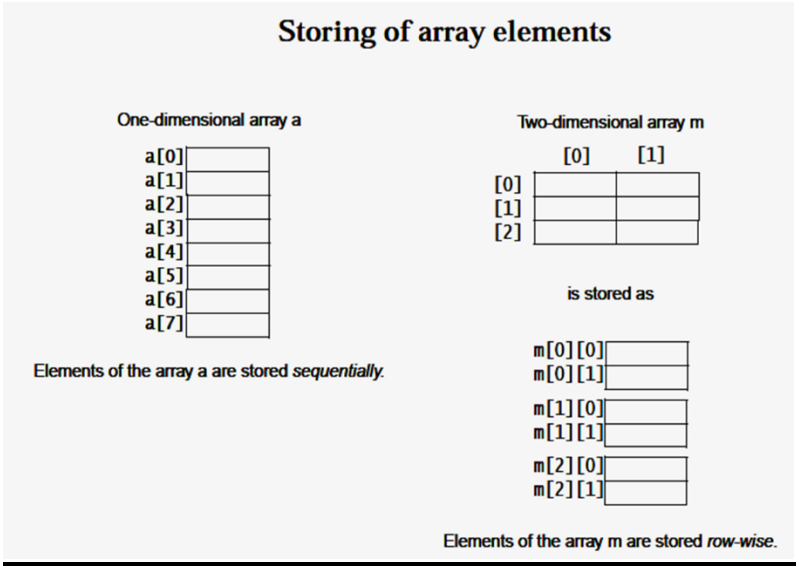
This declaration declares an array named num that contains 10 integers. When the compiler encounters this declaration, it immediately sets aside enough memory to hold all 10 elements.

The square brackets ([ ]) after the "type" indicate that num is going to be an array of type int rather than a single instance of an int. Since the new operator creates (defines) the array, it must know the type and size of the array. The new operator locates a block of memory large enough to contain the array and associates the array name, num, with this memory block.



A program can access each of the array elements (the individual cells) by referring to the name of the array followed by the subscript denoting the element (cell). For example, the third element is denoted num[2].

Storing Array Elements:



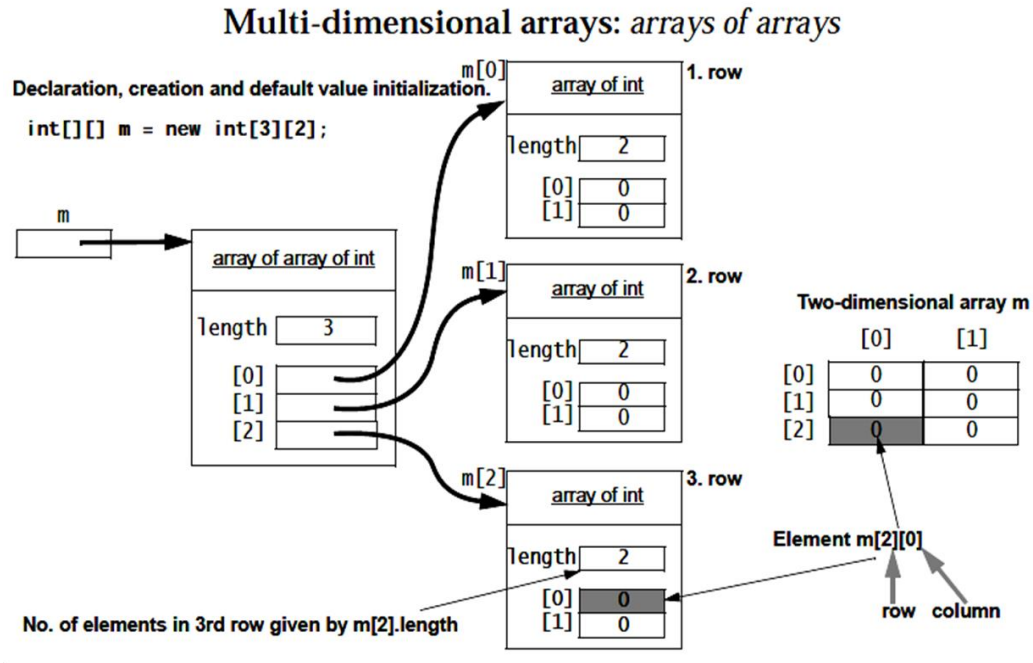
One dimensional arrays are stored in the memory in exactly the way you see them in your head. In a line. The array gets stored in exactly that way in the memory.

Two-dimensional arrays are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on.

Multi-Dimensional Array:

Arrays can have more than one dimension, these arrays-of-arrays are called multidimensional arrays.

Multi-Dimensional array reserves space (construct) for each array that will store the values; because each array is processed separately, it is possible to set different sizes and create a zig-zag, this is precisely the logic computer would use to understand any n-dimensional arrays, i.e. each time as a single row of elements, where each element is itself an array of n-1 dimensions. Keep doing till 0-dimensional array, or a single element.



Iterating over an array.

One-dimensional array

To walk through every element of a one-dimensional array, we use a for loop, that is:

```
int[] myArray = new int[10];
for (int i = 0; i < myArray.length; i++) {
```

```
  myArray[i] = 0;
}
```

Two-dimensional array

For a two-dimensional array, in order to reference every element, we must use two nested loops. This gives us a counter variable for every column and every row in the matrix.

```
int cols = 10;
int rows = 10;
int[][] myArray = new int[cols][rows];

// Two nested loops allow us to visit every spot in a 2D array.
// For every column I, visit every row J.
for (int i = 0; i < cols; i++) {
  for (int j = 0; j < rows; j++) {
    myArray[i][j] = 0;
  }
}
```

Multidimensional Array:

Simply use two nested for loops. To get the sizes of the dimensions, use GetLength():

```
for (int i = 0; i < arrayOfMessages.GetLength(0); i++)
{
    for (int j = 0; j < arrayOfMessages.GetLength(1); j++)
    {
        string s = arrayOfMessages[i, j];
        Console.WriteLine(s);
    }
}
```

This assumes you actually have string[,]. By using GetLowerBound() and GetUpperBound() the get the bounds for each dimension.

To generate a random number we can use the class java.util.Random. Such number are called pseudo-random numbers, as they are not really random.
If you wanted a random double between 0.0 and 10.0 here is how you would generate it.

```
// Generate random doubles 0.0 <= d < 10.0
import java.util.Random;
// ...
// This time, to get different results each run,
Random wheel = new Random();
//...
for ( int i=0; i<100; i++ ){
  // generate a number between 0.0 <= x < 1.0, then scale
  double d = wheel.nextDouble() * 10.0d;
  out.println( d );
  }
```


Module 7: (Classes)

Class:

Class are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

Classes defined in the following way:

```
class MyClass {
   // field, constructor, and
   // method declarations
}
```

This is a class declaration. The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

Parameter Passing:

The terms "arguments" and "parameters" are used interchangeably; they mean the same thing. We use the term formal parameters to refer to the parameters in the definition of the method. In the example that follows, x and y are the formal parameters.

We use the term actual parameters to refer to the variables we use in the method call. In the following example, length and width are actual parameters.

```
   // Method definition
```

```
public int mult(int x, int y)
{
  return x * y;
}

// Where the method mult is used
int length = 10;
int width = 5;
int area = mult(length, width);
```

Pass-by-value means that when you call a method, a copy of each actual parameter (argument) is passed. Copy inside the method is changed, but this will have no effect on the actual parameter.

Passing Arrays:

Arrays are references. This means that when we pass an arrays as a parameter, we are passing its handle or reference. So, we can change the contents of the array inside the method.

```
public static void tryArray(char[] b)
{
  b[0] = 'x';
  b[1] = 'y';
  b[2] = 'z';
}
```

When the following code is executed, the array a does indeed have the new values in the array.

```
char[] a = {'a', 'b', 'c'};
tryArray(a);
System.out.println("a[0] = " + a[0] + ", a[1] = " + a[1] +
                   ", a[2] =" + a[2]);
```

The print statements produces "a[0] = x, a[1] = y, a[2] = z".

Constructors:

Constructors are the methods which are used to initialize objects. Constructor method has the same name as that of class, they are called or invoked when an object of class is created and can't be called explicitly.

Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

New Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
   gear = 1;
   cadence = 10;
   speed = 0;
}
```


Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

In general, class declarations can include these components, in order:

- Modifiers such as public, private, and a number of others that you will encounter later.
- The class name, with the initial letter capitalized by convention.
- The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- The class body, surrounded by braces, {}.

Object Creation and Referencing:

Object is something that has a state in memory.

When an object is created by using new operator, the space in memory is reserved and the structure to hold variables is created. This would definitely like to create a handler (reference) that would keep track of Meta information (like the memory address where the object memory is reserved etc).

A class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the CreateObjectDemo program creates an object and assigns it to a variable:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the Point class, and the second and third lines each create an object of the Rectangle class.

Each of these statements has three parts (discussed in detail below):

- Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.
- Instantiation: The new keyword is a Java operator that creates the object.
- Initialization: The new operator is followed by a call to a constructor, which initializes the new object.

Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.
Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

Point originOne = new Point(23, 94);

The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

int height = new Rectangle().height;

Initializing an Object

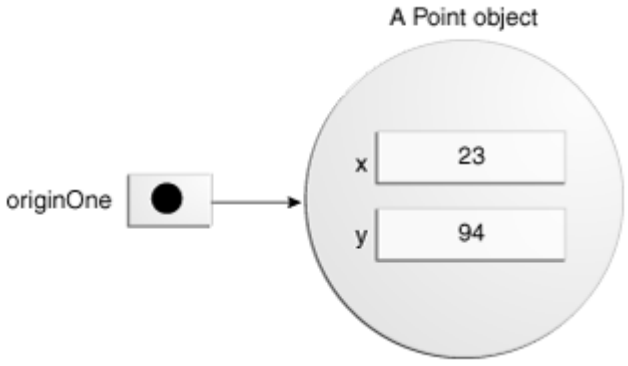
The code for the Point class:

```
public class Point {
   public int x = 0;
   public int y = 0;
   //constructor
   public Point(int a, int b) {
      x = a;
      y = b;
   }
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

Point originOne = new Point(23, 94);

The result of executing this statement can be illustrated in the figure:



originOne now points to a Point object.

The code for the Rectangle class, which contains four constructors:

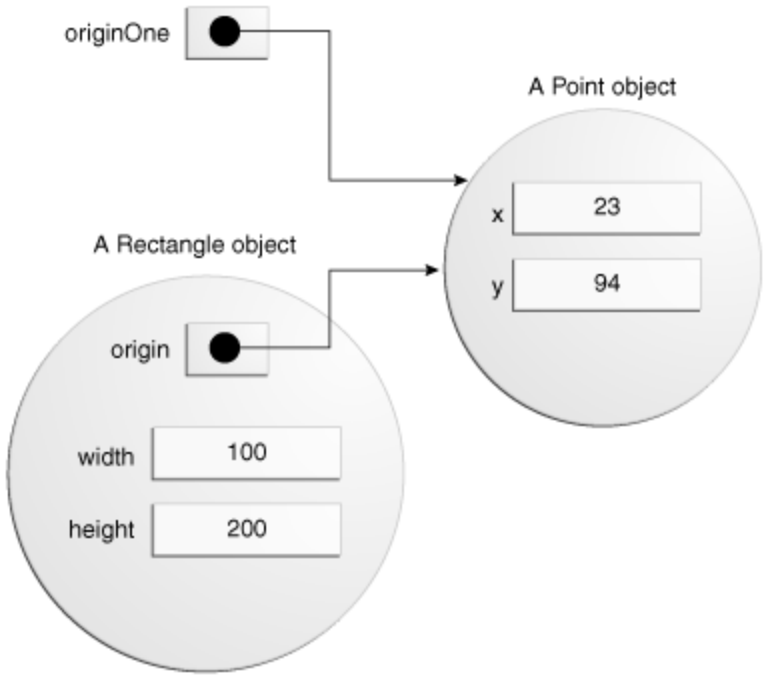
```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
```

```
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```

Each constructor lets you provide initial values for the rectangle's origin, width, and height, using both primitive and reference types. If a class has multiple constructors, they must have different signatures.

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. There are two references to the same Point object—an object can have multiple references to it.



The following line of code calls the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

Rectangle rectTwo = new Rectangle(50, 100);

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

Rectangle rect = new Rectangle();

All classes have at least one constructor. If a class does not explicitly declare any, the compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

Declaring a Variable to Refer to an Object

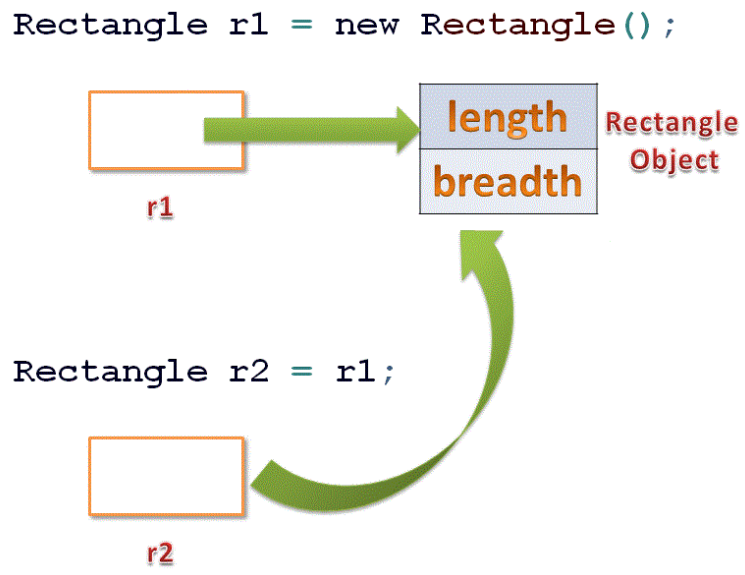
1. Create Distinct Objects.
2. Allocate Memory
3. Create duplicate Copy

Consider This Example –

Rectangle r1 = new Rectangle();
Rectangle r2 = r1;

r1 is reference variable which contain the address of Actual Rectangle Object.
r2 is another reference variable

r2 is initialized with r1 means – “r1 and r2” both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.




```
class Rectangle {
  double length;
  double breadth;
}

class RectangleDemo {
  public static void main(String args[]) {

  Rectangle r1 = new Rectangle();
  Rectangle r2 = r1;

  r1.length = 10;
  r2.length = 20;

  System.out.println("Value of R1's Length : " + r1.length);
  System.out.println("Value of R2's Length : " + r2.length);
```

```
  }
}
```

Output:

```
Value of R1's Length: 20.0
Value of R2's Length: 20.0
```

When variable is declared as field (static or instance variable inside class), then initialization of that variable is optional. In other words, while declaring field variable you may or may not initialize to its value.

Following table shows variables types and their default values

data type	Default value
boolean	false
char	\u0000
int,short,byte / long	0 / 0L
float /double	0.0f / 0.0d
any reference type	null

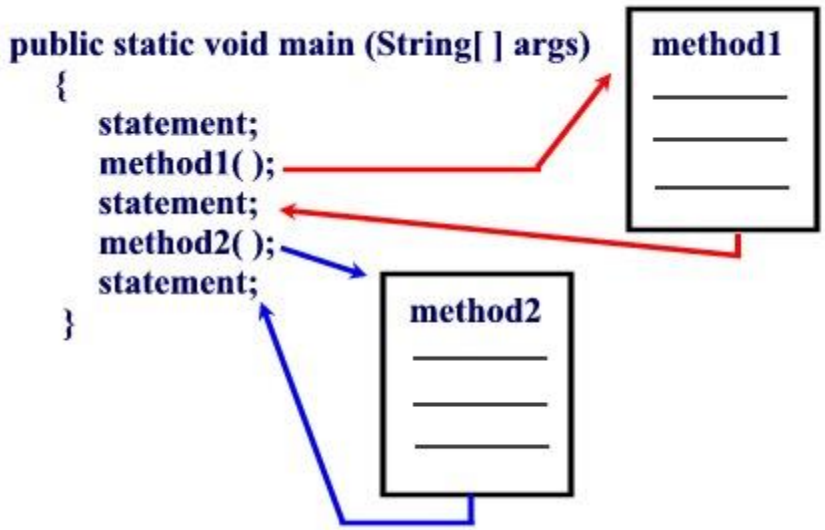
Char primitive default value is \u0000, which means blank/space character.

When you have the variable of reference types then it gets the null as default value.

When local/block variable is declared, they didn't get the default values. They must assigned some value before accessing it otherwise compiler will throw an error.

Declaration of Behaviors: methods

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.
Each method has its own name. When that name is encountered in a program, the execution of the program branches to the body of that method. When the method is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.



Example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
   //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

- Modifiers—such as public, private, and others.
- The return type—the data type of the value returned by the method, or void if the method does not return a value.
- The method name—the rules for field names apply to method names as well, but the convention is a little different.
- The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
- The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Access Modifiers

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Example:

The following class uses private access control:

```
public class Logger {
  private String format;
  public String getFormat() {
    return this.format;
  }
  public void setFormat(String format) {
    this.format = format;
  }
}
```

Here, the format variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly. So to make this variable available to the outside world, we defined two public methods: getFormat(), which returns the value of format, and setFormat(String), which sets its value.

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

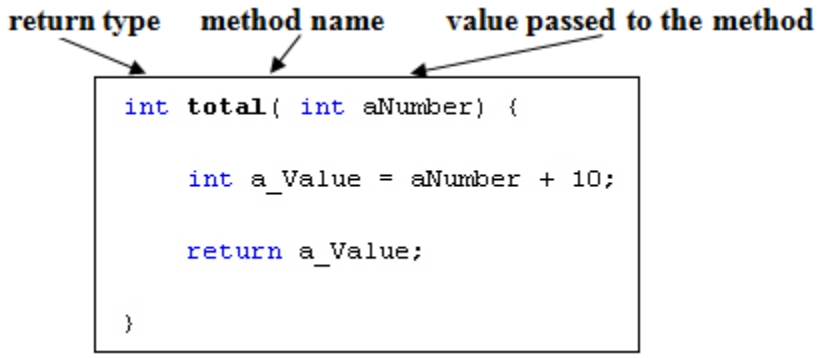
Example:

The following function uses public access control:

```
public static void main(String[] arguments) {
  // ...
}
```


The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Return Value



The method's return type is an int type in the code above. After the method type, you need a space followed by the name of method. In between a pair of round brackets we've method variable called aNumber, and that it will be an integer.

To separate this method from any other code, a pair of curly brackets are needed. The return value can't be a string if you started the method with int total.

A method that doesn't return any value at all can be set up with the word void. In which case, it doesn't need the return keyword. Here's a method that doesn't return a value:

```
void print_text() {

    System.out.println( "Some Text Here" );

}
```

Parameter Names

A parameter is the technical term for the value between the round brackets of your method headers.
The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor.
A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to shadow the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

```
public class Circle {
   private int x, y, radius;
   public void setOrigin(int x, int y) {
      ...
   }
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, not to the field.

Method Body:

Method definition for setWidth():

```
public void setWidth( int newWidth )
{
  width = newWidth ;
}
```

There's only one assignment statement. The purpose of this method is to set the internal instance variable, width, to the parameter, newWidth.

The parameter is being provided to us from an object user. If the object user wants the width to be 10, then the parameter variable newWidth is set to 10.

However, parameter variables disappear after the method called. Instance variables do not. So if we want to record this information in the object, we must copy the value from the parameter variable to the instance variable.

Thus, the assignment statement:

```
width = newWidth ;
```

Return a Value

Let's implement getWidth().

Method definition for getWidth():

```
public int getWidth()
{
  return width ;
}
```

These are the following steps for returning a value:

- Evaluates the (return) expression to a value
- Exits the method, providing the return value

If there are any statements after the return statement, it never gets run, assuming the return statement ran.

```
// Method definition for getWidth()
// println() statement never runs
public int getWidth()
```

```
{
  return width ;
  System.out.println( "NEVER GETS HERE" ) ;
}
```

For example, this code never prints the message "NEVER GETS HERE". That's because running the return statement exits the method with the return value. There's no reason to put code after a return statement that definitely runs, since they have no purpose.

Local variables:

A local variable in Java is a variable that's declared within the body of a method. Then you can use the variable only within that method. Other methods in the class aren't even aware that the variable exists.

Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.

Access modifiers cannot be used for local variables.

Local variables are visible only within the declared method, constructor or block.

Local variables are implemented at stack level internally.

There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to this method only.

```
public class Test{
```

```
  public void pupAge(){
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String args[]){
    Test test = new Test();
    test.pupAge();
  }
}
```

This would produce the following result:

Puppy age is: 7

Field Variables:

Variables outside of methods that all the methods in class can see. These are known as Field variables (or Instance variables). You set them up in exactly the same way as any other variable.

```
package exams;

public class StudentResults {

      String Full_Name;
      String Exam_Name;
      String Exam_Score;
      String Exam_Grade;

}
```

We're setting up four string variables (four string fields). As the names of the fields suggest, the string will hold a person's name, an exam name, a score, and a grade. These four fields will be available to all the methods that we write in this class, and won't be local to any one method. They are said to have global scope.

Difference between a field variable and a local variable

Local Variables:

- Local variable's scope is within the block in which they were defined.

Example:

```
if(x > 10) {
   String local = "Local value";
}
```

- They are alive as long as the block is executed.
- They can not have static access modifier

Field Variables:

- The life span is more than the local variables.
- The are alive as long as the instance of that class is active.
- They can have only 'static' access modifier.
- If I wanted to use it outside of the object, and it was not public, I would have to use getters and/or setters.

Example:

```
public class Point {
   private int xValue; // xValue is a field

   public void showX() {
      System.out.println("X is: " + xValue);
```

```
   }
}
```

Mutator:

A mutator method is a method used to control changes to a variable. write-operations that change the state of the object. They are also widely known as setter methods. Often a setter is accompanied by a getter (also known as an accessor), which returns the value of the private member variable. They are usually declared as public. e.g. switchOn().

Selectors:

Selectors read-operations that have access to the object state, but they do not change the state. They are also usually declared as public. e.g. isOn() is a selector.

Utility Methods:

The operations used by other methods in the class to implement behavior of the class. They are usually declared as private and are not part of the contract of the class, but the implementation.

Method Call

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- Return statement is executed.
- Reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Consider an example:

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example:

int result = sum(6, 9);

Example:

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber{

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }

  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

This would produce the following result:

Minimum value = 6

Method with Parameter:

An example will help set the stage.

```
// Method definition
public int mult(int x, int y)
{
  return x * y;
}

// Where the method mult is used
int length = 10;
int width = 5;
int area = mult(length, width);
```

We use the term actual parameters to refer to variables in the method call, in this case length and width. They are called ``actual" because they determine the actual values that are sent to the method.

You may have heard the term "argument" used, or just "parameter" (without specifying actual or formal). You can usually tell by the context which sort of parameter is being referred to.

Pass a primitive data type by reference

When passing Java objects, you're passing an object reference, which makes it possible to modify the object's member variables. If you want to pass a primitive data type by reference, you need to wrap it in an object.

The easiest of all is to pass it as an array (or even a Vector). Your array only needs to contain a single element, but wrapping it in an array means it can be changed by a function. Here's a simple example of it in action.

```
public static void increment(int[] array, int amount)
{
  array[0] = array[0] + amount;
}

public static void main(String args[])
{
  int[] myInt = { 1 };

  increment (myInt, 5);
```

```
   System.out.println ("Array contents : " + myInt[0]);
}
```

If you're modifying the contents of parameters passed to a method, you really should try to avoid this behavior. It increases the complexity of code, and really should be avoided. The preferred way is to return a value from a method, rather than modifying parameter values directly.

Parameter Passing

Parameter passing methods are the ways in which parameters are transferred between methods when one method calls another. Java provides only one parameter passing method--*pass-by-value*.

Passing-by-value.

Copies of argument values are sent to the method, where the copy is manipulated and in certain cases, one value may be returned. While the copied values may change in the method, the original values in main did not change (unless purposely reassigned after the method).

The situation, when working with arrays, is somewhat different. If we were to make copies of arrays to be sent to methods, we could potentially be copying very large amounts of data.

Not very efficient!

Passing an array mimics a concept called "pass-by-reference", meaning that when an array is passed as an argument, its memory address location (its "reference") is used. In this way, the contents of an array can be changed inside of a method, since we are dealing directly with the actual array and not with a copy of the array.

```
int [ ] num = {1, 2, 3};
testingArray(num);   //Method call
System.out.println("num[0] = " + num[0] + "\n num[1] = " + num[1] + "\n num[2] =" + num[2]);
. . .
```



```
//Method for testing
public static void testingArray(int[ ] value)
{
   value[0] = 4;
   value[1] = 5;
   value[2] = 6;
}
```

Output:
num[0] = 4
num[1] = 5
num[2] = 6
(The values in the array have been changed.
Notice that nothing was "returned".)

Formal parameter — the identifier used in a method to stand for the value that is passed into the method by a caller. Parameters in a subroutine definition are called formal parameters or dummy parameters.
For example, amount is a formal parameter of processDeposit

Actual parameter — the actual value that is passed into the method by a caller. The value of the actual parameter can be assigned to the formal parameter. Parameters are passed to a subroutine when it is called. When a subroutine is called, the actual parameters in the subroutine call statement are evaluated and the values are assigned to the formal parameters in the subroutine's definition. Then the body of the subroutine is executed.

For example, the 200 used when processDeposit is called is an actual parameter.
Actual parameters are often called arguments

When a method is called, the formal parameter is temporarily "bound" to the actual parameter. The method uses the formal parameter to stand for the actual value that the caller wants to be used.

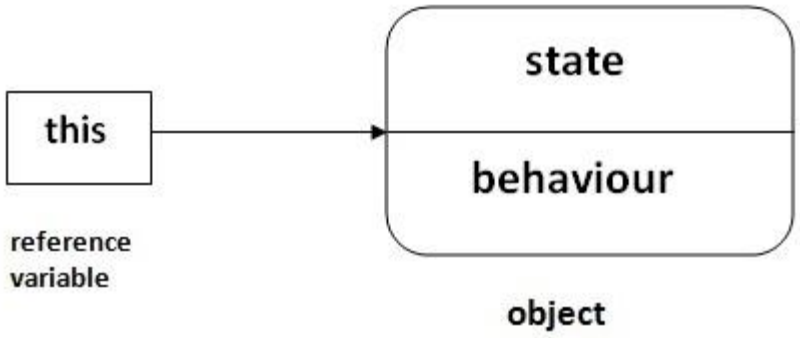
For example, here the processDeposit method uses the formal parameter amount to stand for the actual value used in the procedure call:

balance = balance + amount ;

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



Static Members in Class

Static variable's value is same for all the object (or instances) of the class or in other words you can say that all instances (objects) of the same class share a single copy of static variables.

Understand this with an example:

class VariableDemo

```
{
  static int count=0;
  public void increment()
  {
     count++;
  }
  public static void main(String args[])
  {
     VariableDemo obj1=new VariableDemo();
     VariableDemo obj2=new VariableDemo();
     obj1.increment();
     obj2.increment();
     System.out.println("Obj1: count is="+obj1.count);
     System.out.println("Obj2: count is="+obj2.count);
  }
}
```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

As you can see in the above example that both the objects of class, are sharing a same copy of static variable that's why they displayed the same value of count.

- It is a method which belongs to the class and not to the object(instance)
- A static method can access only static data. It cannot access non-static data (instance variables)
- A static method can call only other static methods and cannot call a non-static method from it.
- A static method can be accessed directly by the class name and doesn't need any object

Syntax: <class-name>.<method-name>

- A static method cannot refer to “this” or “super” keywords in anyway

In the code below we have declared a class named Vehicle, a class field member named vehicleType and a method named getVehicleType(), both declared as static.

```
1 public class Vehicle {
2
3    private static String vehicleType;
4
5    public static String getVehicleType(){
6        return vehicleType;
7    }
8
9 }
```

The static modifier allows us to access the variable vehicleType and the method getVehicleType() using the class name itself, as follows:

Vehicle.vehicleType
Vehicle.getVehicleType()

Static Methods and Variables:

Variables can be declared with the “static” keyword.

Example: static int y = 0;

When a variable is declared with the keyword "static", it's called a "class variable". All instances share the same copy of the variable. A class variable can be accessed directly with the class, without the need to create an instance.

Static methods and variables can only be used with outer classes. Inner classes have no static methods or variables. A static method or variable doesn't require an instance of the class in order to run.
Before an object of a class is created, all static member variables in a class are initialized, and all static initialization code blocks are executed. These items are handled in the order in which they appear in the class.

A static method is used as a utility method, and it never depends on the value of an instance member variable. Because a static method is only associated with a class, it can't access the instance member variable values of its class.

Field Variables:

Variables outside of methods that all the methods in your class can see. These are known as Field variables (or Instance variables). You set them up in exactly the same way as any other variable. Field variables are local to an object, i.e. each object has its own copy of all the field variables.

Instance methods

Methods and variables that are not declared as static are known as instance methods and instance variables. To refer to instance methods and variables, you must instantiate the class first means you should create an object of that class first.
They're associated with a particular object.
They have no definition modifier.
They're created with every object instantiated from the class in which they're declared.

Instance methods and member variables are used by an instance of a class, that is, by an object. An instance member variable is declared inside a class, but not within a method. Instance methods usually use instance member variables to affect the behavior of the method.

Suppose that you want to have a class that collects two-dimensional points and plots them on a graph. The following skeleton class uses member variables to hold the list of points and an inner class to manage the two-dimensional list of points.

```
01 public class Plotter {
02
03    // This inner class manages the points
04    class Point {
05       Double x;
06       Double y;
07
08       Point(Double x, Double y) {
09          this.x = x;
10          this.y = y;
11       }
12       Double getXCoordinate() {
13          return x;
14       }
15
16       Double getYCoordinate() {
17          return y;
18       }
19    }
20
21    List<Point> points = new List<Point>();
22
23    public void plot(Double x, Double y) {
24       points.add(new Point(x, y));
25    }
26
27    // The following method takes the list of points and does something with them
28    public void render() {
29    }
30 }
```


Scope:

Scope is where in the source code a variable can be used directly with its simple name, without indicating where it is declared.

Class Level Scope:

The scope of a variable is the part of the program over which the variable name can be referenced.

You cannot refer to a variable before its declaration.

You can declare variables in several different places:

- In a class body as class fields. Variables declared here are referred to as class-level variables.
- As parameters of a method or constructor.
- In a method's body or a constructor's body.
- Within a statement block, such as inside a while or for block.

Variable scope refers to the accessibility of a variable.

Inside a Block:

The variables defined in a block are only accessible from within the block. The scope of the variable is the block in which it is defined. For example, consider the following for statement.

```
public class MainClass {
  public static void main(String[] args) {
    for (int x = 0; x < 5; x++) {
        System.out.println(x);
    }
  }

}
```

Lifetime:

Life time of a variable declaration is the period the variables exists in the memory during execution.

Different kinds of variables have different lifetimes. Parameter variables exist when the method call is made, and lasts until the method call is complete. These parameter variables hold boxes to values or handles. If the box holds handles (i.e., the parameter variable is an object variable), then when the box disappears, the handle may still stay around. That's usually because you have some variable around that holds the handle, and was passed to the method as an argument.

Lifetime for Local Variables:

Local variables also have a similar lifetime. The boxes are created when the declaration appears, and lasts until you exit the method.

Lifetime for Instance Variables:

Instance variables have a much longer lifetime. They are created when the object is constructed, and goes away when the object disappears. The object disappears when it is no longer being used, which basically means that no variable has a handle to the object. When that happens, the garbage collector gets rid of the object.

Lifetime for Static Variables:

Static variables are called class variable and in way of scope they loaded when the class is loaded and unloaded when class is unloaded. For example a class variable like

private int classinVar;

is automatically initialized by its default value when class loaded, and same concept is with signout when you get signout then that class would go out of context with its static field.

During loading of the class at runtime, the static variables are created and initialized only once.

Lifetime for Field Variables:

Flied variables applies to all field variables in an object. Field variables exist as long as the object they belong to exists. The field variables are allocated and automatically initialized to default values when an object is created, if no explicit initialization is attempted by the program.

Constructors:

Constructor is a special type of method that is used to initialize the object.

Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating constructor

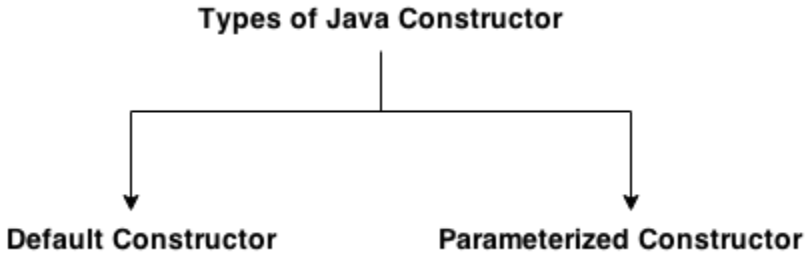
There are basically two rules defined for the constructor.

- Constructor name must be same as its class name
- Constructor must have no explicit return type

Types of constructors

There are two types of constructors:

- Default constructor (no-arg constructor)
- Parameterized constructor



Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
<class_name>(){}
```

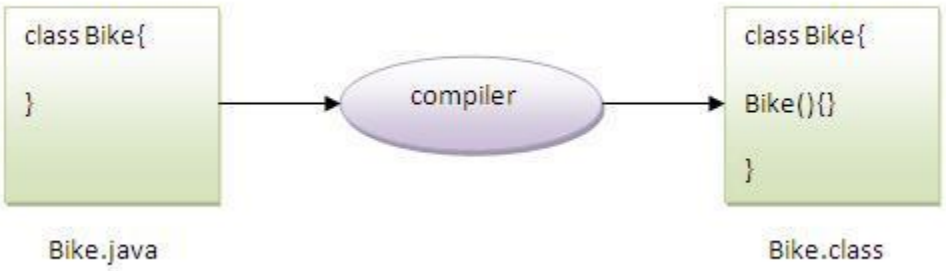
Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

Output:

Bike is created



Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
class Student3{
int id;
String name;

void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

Output:

```
0 null
0 null
```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Parameterized constructor

A constructor that have parameters is known as parameterized constructor.
Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Output:

111 Karan

222 Aryan

Module 8 (Object Communication)

Advantages of Abstraction:

- By using abstraction, we can separate the things that can be grouped to another type.
- Frequently changing properties and methods can be grouped to a separate type so that the main type need not undergo changes.
- Simplifies the representation of the domain models.
- Abstraction makes the application extendable in much easier way. It makes refactoring much easier.
- When developing with higher level of abstraction, you communicate the behavior and less the implementation.
- Abstraction helps designers of all kinds in all fields focus on a set of relevant fundamentals. It does not mean the details go away; instead they are layered and handled at an appropriate level of focus. This approach isolates aspects of a complex design.

Abstraction Principles:

A good abstraction is when the programmers use abstractions whenever suitable in order to avoid duplication (usually of code).

It is used merely for helping the programmer comprehend & modify programs faster to suit different scenarios.

Abstraction is really the process of pulling out common pieces of functionality into re-usable components (be it abstract classes, parent classes, interfaces, etc.)

A good practice is to determine which methods will be useful to have in order to manipulate the fields of a Class.

Abstraction is to represent an object often comes before its implementation.

Structured Programming:

Methodology:

- **Block:**

Block is a section of code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block-structured programming language. Blocks are fundamental to structured programming, where control structures are formed from blocks.

- **Choice (conditional execution)**

There are three major structures related to the conditional execution - the if statement, the if-else statement, and the switch-case statement.

If Condition:

The if statement allows the program to execute a block of code, only if a certain condition is met. The general structure for an if statement is given by:

```
1 if ( conditional statement ) {
    //conditionally executed
2 code
3 }//end if statement
```

The conditional expression in the if statement can be any statement that evaluates to a Boolean true or false.

If-else condition:

else statement will be executed when a certain block of code in the event that the conditional in an if statement is false. There is no alternate behavior when the expression evaluates to false. The general form of an if-else statement is simply an extension of that for the if statement:

```
1
        if ( conditional statement ) {
2
           //conditionally executed code
3
        }//end if statement
4
        else {
5
           //alternate code if <conditional_exp<b></b>ression> is false
6
```

}//end else statement

Switch Condition:

Switch-case structures are used to execute alternate blocks of code based on the value of a single variable. The general of the switch-case expression is given by:

```
01 switch (<test_variable>) {
02    case <test_value_1>: {
03       //statements to execute if test_variable is equal to test_value_1
04       break;
05    }
06
07    //... other case: test_value structures
08
09    default: {
10       //statements to execute if test_variable is not equal to
11       //any of the specified values
12    }
13 }
```

- **Loop:**

Loops are used to repeat a block of code. There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

For Loop:

The For...Next construction performs the loop a set number of times. It uses a loop control variable, also called a counter, to keep track of the repetitions. You specify the starting and ending values for this counter, and you can optionally specify the amount by which it increases from one repetition to the next.

FOR - for loops are the most useful type. The syntax for a for loop is

```
for ( variable initialization; condition; variable update ) {
  Code to execute while the condition is true
}
```

While Loop:

The **While**...**End While** construction runs a set of statements as long as the condition specified in the **While** statement is **True**.

WHILE - WHILE loops are very simple. The basic structure is

```
While condition
   [ statements ]
   [ Continue While ]
   [ statements ]
   [ Exit While ]
   [ statements ]
End While
```

Do. While Loop:

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is

```
do {
} while ( condition );
```

The condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once. A while loop says "Loop while the condition is true, and

execute this block of code", a do..while loop says "Execute this block of code, and then continue to loop while the condition is true".

Advantages:

By using structured programming, the complexity is reduced. Also, using logical structures ensures that the flow of control is clear.

Results in methods that are simpler to understand and can be re-used many times, which saves time and reduces complexity, but also increases reliability. It is also easier to update or fix the program by replacing individual modules rather than larger amounts of code.

Duplicate Program Code:

There are three basic types of duplication that we can eliminate from our code that successfully build on each other.

Data
Type
Algorithm

Most developers tend to get stuck at the data level, but in this post, I will show you how to recognize type and algorithm duplication and refactor it out of your code.

Data duplication

The most basic type of duplication is that of data. It is also very easily recognizable.

Take a look at these methods:

```
public Position WalkNorth()
{
  var player = GetPlayer();
  player.Move("N");
  return player.NewPosition;
}

public Position WalkSouth()
{
  var player = GetPlayer();
  player.Move("S");
  return player.NewPosition;
}

public Position WalkEast()
{
  var player = GetPlayer();
  player.Move("E");
  return player.NewPosition;
}

public Position WalkWest()
{
  var player = GetPlayer();
  player.Move("W");
  return player.NewPosition;
}
```

Pretty easy to see here what needs to be refactored.

Most developers don't need any help to realize that you should probably refactor this code to a method like the following:

```
public Position Walk(string direction)
{
  var player = GetPlayer();
  player.Move(direction);
  return player.NewPosition;
}
```

Communication and cooperation:

Object communication and interaction

It is through interactions among objects that programmers get the behavior their programs were designed for. Software objects communicate and interact with each other by calling (or invoking) each other's methods.

Calling an instance method:

Object A calls a method implemented by object B to have it to perform some behavior or return some value. This is also sometimes referred to as A *sending a message* to B. For example, when the play button is pressed on the CD player application, the button object may call the "`play`" method of the application object with the understanding that this means to play the current track.

Sometimes the called object's method needs additional information in order to perform its task. If your CD player application had multiple buttons, each of which played a different track, then the `play` method would also need the number of the track to play. This information would be passed along as an *argument* (or parameter) when calling the method.

There are three parts to calling a method:

1. The object you are calling that implements the method (e.g., the CD app object)
2. The name of the method to perform (e.g., `play`)
3. Any arguments needed by the called object (e.g., the CD track #)

The syntax for making a method call is to list the object to be called, followed by a period, then the method name. Any arguments needed are enclosed in parentheses after the method name.

A line of Java code to have the `cdApp` object play track 3 might look like:

```
cdApp.play(3);
```

It is also very common for an object to call one of its own methods as part of the implementation of another method. For example, suppose the CD player application was playing through a play list and it was time to play the next song in the list. The application object would need to call its own `play()` message, passing the number of the track to play next. Code to do this would look like:

```
this.play(3);
```

or more commonly,

```
play(3);
```

The special keyword "`this`" always refers to the object whose code is currently executing. In the second case there is no object specified.

Reference variable:

Reference Variable is used to store the address of the variable. In order to call a method on an object, we need a reference to the object. Assigning Reference will not create distinct copies of Objects but rather all reference variables are referring to same Object.

Rectangle r1 = new Rectangle();
Rectangle r2 = r1;

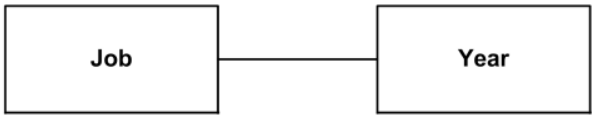
r1 is reference variable which contain the address of Actual Rectangle Object.

r2 is another reference variable

r2 is initialized with r1 means – "r1 and r2" both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.

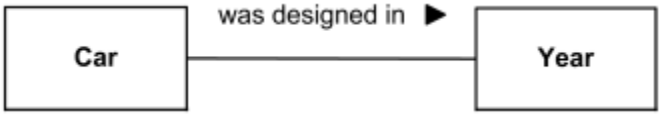
Types of Association:

- It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct).



Job and Year classifiers are associated

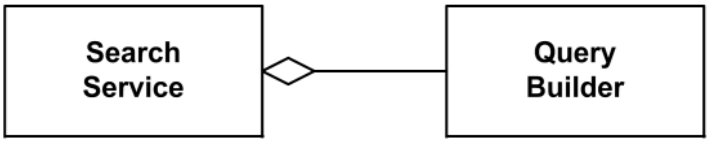
- A small solid triangle could be placed next to or in place of the name of **binary association** (drawn as a solid line) to show the **order of the ends** of the association. The arrow points along the line in the direction of **the last end** in the order of the association ends. This notation also indicates that the association is to be **read** from the first end to the last end.



Order of the ends and reading: ***Car - was designed in - Year***

- **Aggregation** (**shared aggregation**) is a "weak" form of aggregation when part instance is independent of the composite:
  - the same (shared) part could be included in several composites, and
  - if composite is deleted, shared parts may still exist.

Shared aggregation is shown as binary association decorated with a **hollow diamond** as a terminal adornment at the aggregate end of the association line.

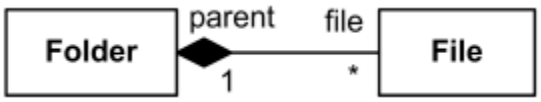


Search Service *has a* ***Query Builder*** *using shared* ***aggregation***

- **Composition** (**composite aggregation**) is a "strong" form of aggregation. It is a **whole/part relationship**,

  - It is binary association,
  - Part could be included in **at most one** composite (whole) at a time, and
  - If a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

Composite aggregation is depicted as a binary association decorated with a **filled black diamond** at the aggregate (whole) end.



Folder could contain many files, while each File has exactly one Folder parent.
If Folder is deleted, all contained Files are deleted as well.

Encapsulation:

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

Abstraction represent taking out the behavior from How exactly its implemented, one example of abstraction is interface while Encapsulation means hiding details of implementation from outside world so that when things change nobody gets affected. One example of Encapsulation is private methods; clients don't care about it, You can change, amend or even remove that method if that method is not encapsulated and it were public all your clients would have been affected.

Encapsulation is a mechanism by which you restrict the access to some of the object's components, as well as binding the data and methods operating on the data.

Now if we consider a laptop, as an end user I have access only to some features of the system. So I could use the mouse to move the cursor, or the keyboard for typing text, but I would not have access to the internal components of the laptop. Again the keyboard in turn is bound internally to a set of methods that operate in response to a user action or an event.

Abstraction is the ability to define an object that can represent abstract entities which can work, change state and communicate with other entities.

Let us take the example of our laptop Keyboard itself, here we have a number of Keys, each performing some function dependent on the value given. Now all keys have a certain value that is accepted by the CPU when you press it. So we create a common object called Key with following methods.

Module 9: (Modifiers)

There are two types of modifiers in java: access modifiers and non-access modifiers.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

There are 4 types of java access modifiers:

- Visible to the world (public).
- Visible to the package and all subclasses (protected).
- Visible to the package. The default. No modifiers are needed.
- Visible to the class only (private).

Visible to the world (public):

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However if the public class we are trying to access is in a different package, then the public class still need to be imported.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example:

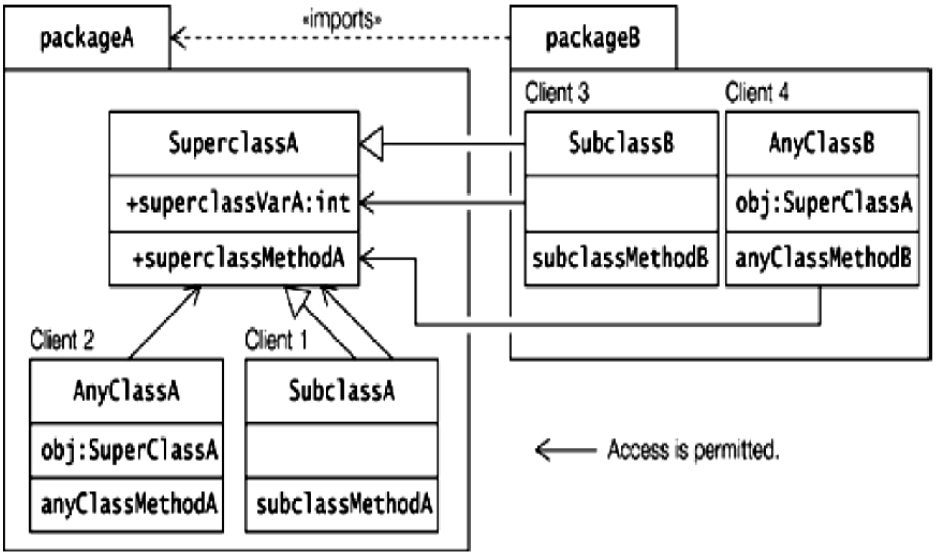
The following function uses public access control:

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

```
public static void main(String[] arguments) {
  // ...
}
```

Example:

Two source files are shown. The package hierarchy defined by the source files is depicted, showing the two packages packageA and packageB containing their respective classes. Classes in package packageB use classes from package packageA. SuperclassA in packageA has two subclasses: SubclassA in packageA and SubclassB in packageB.



Client 1: Client 1 is a subclass in the same package, which accesses an inherited field.

Client 2: Cliet 2 is a Non-subclass in the same package, which invokes a method on an instance of the class AnyClassA.

Client 3: Client 3 is a subclass in another package, which invokes an inherited method. SubclassB.

Client 4: Client 4 is a non-subclass in another package, which accesses a field in an instance of the class.

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in an interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example:

The following parent class uses protected access control, to allow its child class override openSpeaker() method:

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}

class StreamingAudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}
```

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than AudioPlayer. If we define it as public, then it would become accessible to

all the outside world. But our intension is to expose this method to its subclass only, thats why we used protected modifier.

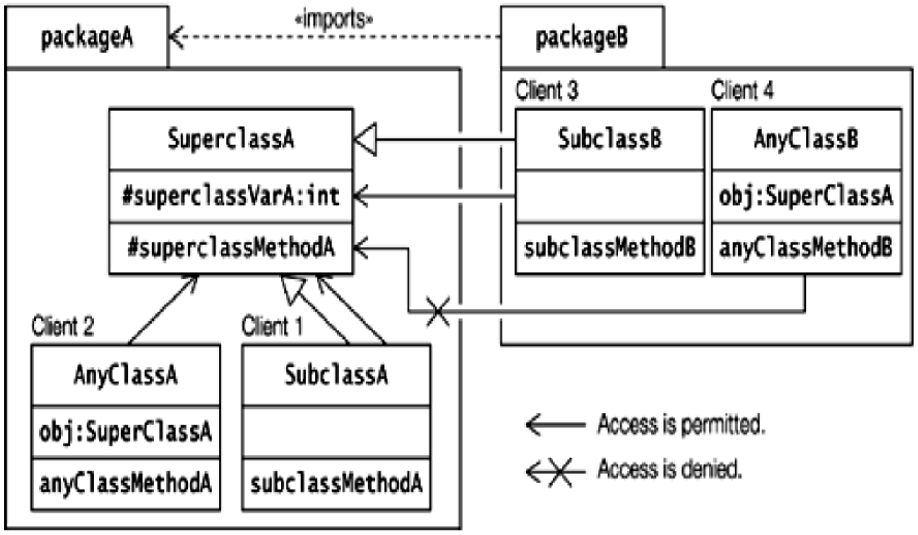
Access Control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

Example:

If the field superclassVarA and the method superclass MethodA have protected accessibility, then they are accessible within package packageA, and only accessible by subclasses in any other packages.



Client 1: From a subclass in the same package, which accesses an inherited field.

Client 2: From a non-subclass in the same package, which invokes a method on an instance of the class AnyClassA.
Client 3: From a subclass in another package, which invokes an inherited method SubclassB.
Client 4: From a non-subclass in another package, which cannot accesses a field in an instance of the class.

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

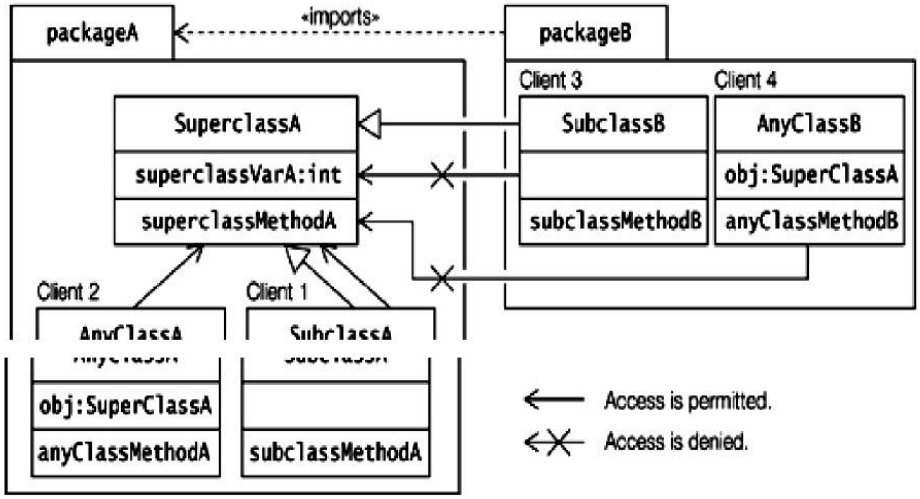
A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example:

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";

boolean processOrder() {
   return true;
}
```

If the field superClassVarA and the method superClassMethodA have default accessibility, then they are accessible within package packageA, and not accessible by subclasses or other classes in any other packages.



Client 1: From a subclass in the same package, which accesses an inherited field. SubclassA is such a client.

Client 2: From a non-subclass in the same package, which invokes a method on an instance of the class AnyClassA is such a client.

Client 3: From a subclass in another package, which cannot invokes an inherited method. SubclassB.

Client 4: From a non-subclass in another package, which cannot accesses a field in an instance of the class. AnyClassB.

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Example:

The following class uses private access control:

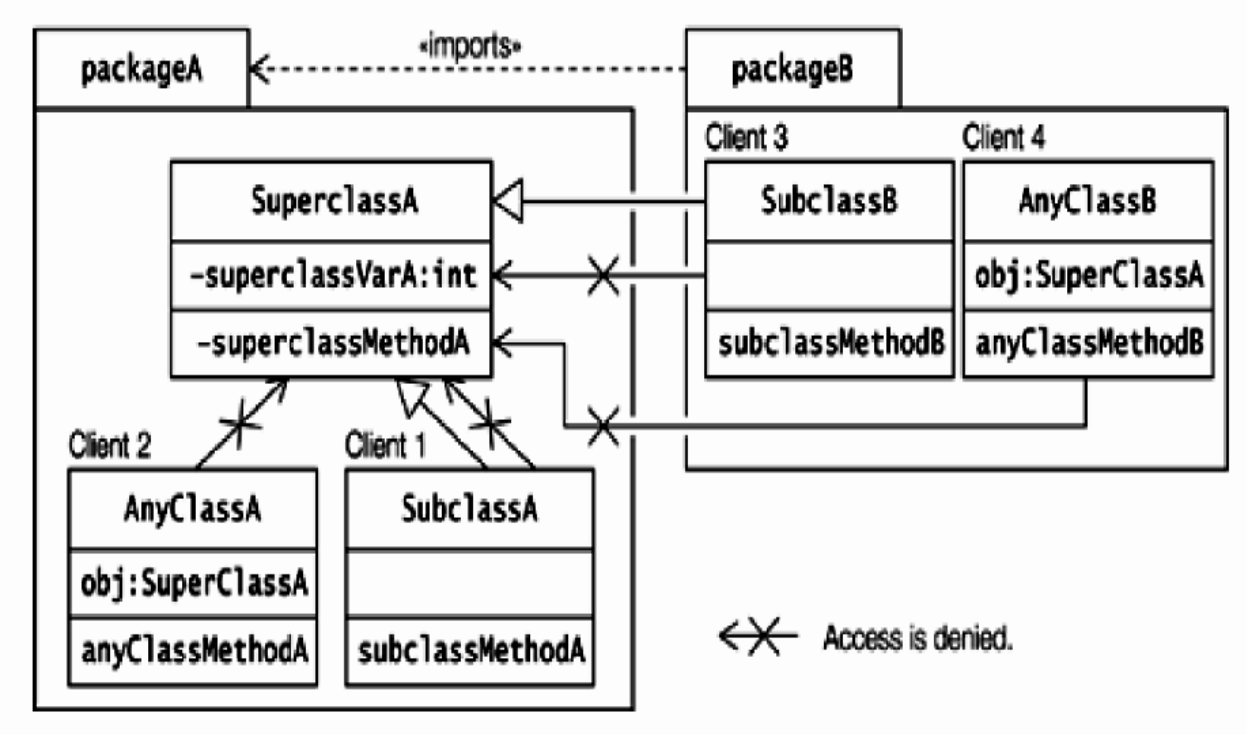
```
public class Logger {
   private String format;
   public String getFormat() {
      return this.format;
   }
   public void setFormat(String format) {
      this.format = format;
   }
}
```

Here, the format variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So to make this variable available to the outside world, we defined two public methods: getFormat(), which returns the value of format, and setFormat(String), which sets its value.

Example:

If the field superClassVarA and the method superClassMethodA have private accessibility, then they are not accessible within package packageA, and not accessible by subclasses or other classes in any other packages.



Client 1: From a subclass in the same package, which cannot accesses an inherited field. SubclassA is not able to access any private members.

Client 2: From a non-subclass in the same package, private members are not accessible on an instance of the class AnyClassA.

Client 3: From a subclass in another package, which cannot invokes a private members in SubclassB.

Client 4: From a non-subclass in another package, which cannot accesses a field in an instance of the class AnyClassB.

A brief summary of all access modifiers are given below:

Table 4.4. Summary of Accessibility Modifiers for Members

Modifiers	Members
public	Accessible everywhere.
protected	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
private	Only accessible in its own class and not anywhere else.

Non Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Static Modifier:

Static Modifiers are used to create class variable and class methods which can be accessed without instance of a class.

Static Variables:

The static key word is used to create variables that will exist independently of any instances created for the class. Static variables are also known as class variables. Local variables cannot be declared static. Static variables are defined as a class member that can be accessed without any object of that class. Static variable has only one single storage. Only one copy of the static variable exists regardless of the number of instances of the class. All the object of the class having static variable will have the same instance of static variable. Static variables are initialized only once.

Static variable are used to represent common property of a class. It saves memory. Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, company_name field will be mark as static.

Static Methods:

The static key word is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Main() method is the most common example of static method. Main() method is declared as static because it is called before any object of the class is created.

The final Modifier:

Final modifier is used to declare a field as final i.e. it prevents its content from being modified. Final field must be initialized when it is declared.

Final Variables:

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to a different object.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

With variables, the final modifier often is used with static to make the constant a class variable.

Final Methods:

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Final Classes:

The main purpose of using a class being declared as final is to prevent the class from being subclassed. A final method in a class is complete and cannot be overridden in any subclass. If a class is marked as final then no class can inherit any feature from the final class. A final class and an interface represent two extremes when it comes to providing implementation.

The abstract Modifier:

Abstract Class:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (Since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.

Abstract Methods:

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

The synchronized Modifier:

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. Their execution is then mutually exclusive among all threads. The synchronized modifier can be applied with any of the four access level modifiers.

Native Modifiers:

Native methods are also called foreign methods. It marks a method, that it will be implemented in other languages, not in Java.

The transient Modifier:

An instance variable is marked transient to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Objects can be stored using serialization. Serialization transforms objects into an output format which is helpful for storing objects. Objects can later be retrieved in the same state as when they were serialized, meaning that fields included in the serialization will have the same values at the time of serialization. Such objects are said to be Persistent.

The fields are declared with keyword Transient in their class declaration if its value should not be saved when objects of the class are written to persistent storage.

The volatile Modifier:

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of your program. The volatile modifier can be used to inform the compiler that it should not attempt to perform optimizations on the field, which could cause unpredictable results when the field is accessed by multiple threads. Volatile variables are used in case of multithreading program. Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Summary:

A brief summary of all non-access modifiers are given below:

Table 4.5. Summary of Other Modifiers for Members

Modifiers	Fields	Methods
static	Defines a class variable.	Defines a class method.
final	Defines a constant.	The method cannot be overridden.
abstract	Not relevant.	No method body is defined. Its class must also be designated abstract.
synchronized	Not relevant.	Only one thread at a time can execute the method.
native	Not relevant.	Declares that the method is implemented in another language.
transient	The value in the field will not be included when the object is serialized.	Not applicable.
volatile	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

Module 10: (Exception Handling)

Exception Handling

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception:

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions e.g. IOException, SQLException etc. These exceptions cannot simply

be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.

Unchecked Exception:

An Unchecked exception is an exception that occurs at the time of execution, these are also called as Runtime Exceptions, and these include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation. e.g. Arithmetic Exception, Null Pointer Exception, Array Index Out Of Bounds Exception etc.

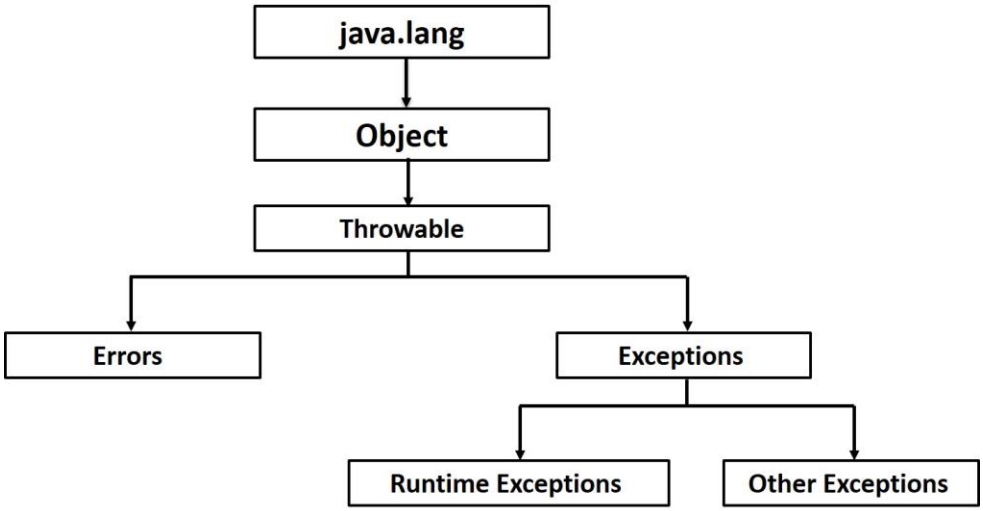
Errors:

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures. Errors are generated to indicate errors generated by the runtime environment. The Exception class has two main subclasses: IOException class and RuntimeException Class.



Try block

Try block is used to enclose the code that might throw an exception. It must be used within the method.

Try block must be followed by either catch or finally block.

Syntax of try-catch

```
try{
   //code that may throw exception
   }catch(Exception_class_Name
ref){}
```

Syntax of try-finally block

```
try{
//code that may throw
exception
}finally{}
```

Catch block

Catch block is used to handle the Exception. It must be used after the try block only.
You can use multiple catch block with a single try.

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try {
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
  file = new
FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i)
{
  i.printStackTrace();
  return -1;
}catch(FileNotFoundException f)
//Not valid!
{
  f.printStackTrace();
  return -1;
}
```

The finally block

Finally block is a block that is used to execute important code such as closing connection, stream etc.
The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Example:

```
public class ExcepTest{

  public static void main(String args[]){
    int a[] = new int[2];
    try{
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
    finally{
      a[0] = 6;
      System.out.println("First element value: " +a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This would produce the following result:

Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed

Important Statements:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Throw keyword

The Throw keyword is used to explicitly throw an exception.

The syntax of throw keyword is given below.

```
throw exception;
```

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.
You can throw an exception, either a newly instantiated one or an exception by using the throw keyword.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
  public void deposit(double amount) throws
RemoteException
  {
    // Method implementation
    throw new RemoteException();
  }
  //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
  public void withdraw(double amount) throws
RemoteException,
                    InsufficientFundsException
  {
     // Method implementation
  }
  //Remainder of class definition
}
```


Module 11: (XML)

XML

Extensible Markup Language (XML) is used to describe data. XML is a markup language much like HTML. The XML standard is a flexible way to create information formats and electronically share structured data via the public Internet, as well as via corporate networks.

The Difference between XML and HTML

XML and HTML were designed with different goals:

- HTML is about displaying information, XML is about describing information.
- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

XHTML	XML
Based on tag pairs	Based on tag pairs
Purpose: • Markup language • Displays the data • Focus: How it looks • Does not care about the meaning of contents • Predefined tags	Purpose: • Meta-markup language to define markup language • ML defined by XML describes the data • Focus: meaning (what) of data • Method of data exchange Content must be well structured • No predefined tags

Displaying XML

If you've ever tried to open an XML file and expected to view it like a typical HTML file, chances are you were disappointed with the results. When you clicked that XML file open, you probably saw something like this (sample code taken from the W3C):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
```



```
  <COUNTRY>UK</COUNTRY>
  <COMPANY>CBS Records</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
 </CD>
```

That type of layout does you no good. You can read it, sure, but you'd have a very hard time making sense of an entire report by reading it in this manner. XML files do not carry any formatting information, therefore, you simply see a raw output. The W3C recommends formatting an XML file either with CSS, XSLT or even JavaScript. If you use CSS, it's as simple as (from the W3C):

```
CATALOG
{
background-color: #ffffff;
width: 100%;
}
CD
{
display: block;
margin-bottom: 30pt;
margin-left: 0;
}
TITLE
{
color: #FF0000;
font-size: 20pt;
}
ARTIST
{
color: #0000FF;
font-size: 20pt;
}
COUNTRY,PRICE,YEAR,COMPANY
{
display: block;
color: #000000;
margin-left: 20pt;
}
```

XML Syntax

To link the XML file to the CSS, you use this code:

<?xml-stylesheet type="text/css" href="cd_catalog.css"?>

In XML, it is illegal to omit the closing tag. All elements must have a closing tag:
This is a paragraph.

XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.

Opening and closing tags must be written with the same case:
<Message>This is incorrect</message>
<message>This is correct</message>

"Opening and closing tags" are often referred to as "Start and end tags". Use whatever you prefer. It is exactly the same thing.

In XML, all elements must be properly nested within each other:
This text is bold and italic

In the example above, "Properly nested" simply means that since the element is opened inside the element, it must be closed inside the element.

XML document

All XML documents must contain a single tag pair to define the root element. All other elements must be nested within the root element. All elements can have sub (children) elements. Sub elements must be in pairs and correctly nested within their parent element:

```
<root>
  <child>
    <subchild>
    </subchild>
  </child>
</root>
```

Example:

```
<Book>
<Title>    </Title>              <Chapter>
     <para>    </para>
</Chapter>
</Book>
```

Syntax Rules for XML declaration

- The XML declaration is case sensitive and must begin with "<?xml>" where "xml" is written in lower-case.
- If document contains XML declaration, then it strictly needs to be the first statement of the XML document.
- The XML declaration strictly needs be the first statement in the XML document.
- An HTTP protocol can override the value of encoding that you put in the XML declaration.

Syntax Rules for XML Attributes

- Attribute names in XML (unlike HTML) are case sensitive. That is, HREF and href are considered two different XML attributes.

- Same attribute cannot have two values in a syntax. The following example shows incorrect syntax because the attribute b is specified twice:

  ....

- Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml syntax:

  ....

XML Text

- The names of XML-elements and XML-attributes are case-sensitive, which means the name of start and end elements need to be written in the same case.
- To avoid character encoding problems, all XML files should be saved as Unicode UTF-8 or UTF-16 files.
- Whitespace characters like blanks, tabs and line-breaks between XML-elements and between the XML-attributes will be ignored.
- Some characters are reserved by the XML syntax itself. Hence, they cannot be used directly. To use them, some replacement-entities are used, which are listed below:

not allowed character	**replacement-entity**	**character description**
<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

DTD

The purpose of a DTD (Document-Type-Information) is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML document, or as an external reference.

Internal DTD

This is an XML document with a Document Type Definition:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note    (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DTD is interpreted like this:

!ELEMENT note (in line 2) defines the element "note" as having four elements: "to,from,heading,body".
!ELEMENT to (in line 3) defines the "to" element to be of the type "CDATA".
!ELEMENT from (in line 4) defines the "from" element to be of the type "CDATA"
and so on.....

External DTD

This is the same XML document with an external DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

This is a copy of the file "note.dtd" containing the Document Type Definition:

```
<?xml version="1.0"?>
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Why use a DTD?

XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

A lot of forums are emerging to define standard DTDs for almost everything in the areas of data exchange.

Unicode Characters:

The table below lists the five XML predefined entities. The "Name" column mentions the entity's name. The "Character" column shows the character. To render the character, the format &name; is used; for example, & renders as &. The "Unicode code point" column cites the character with standard UCS/Unicode "U+" notation, which shows the character's code point in hexadecimal. The decimal equivalent of the code point is then shown in parentheses. The "Standard" column indicates the first version of XML that includes the entity. The "Description" column cites the character with its UCS/Unicode name.

Name	Character	Unicode code point (decimal)	Standard	Description
quot	"	U+0022 (34)	XML 1.0	double quotation mark
amp	&	U+0026 (38)	XML 1.0	ampersand
apos	'	U+0027 (39)	XML 1.0	apostrophe *(apostrophe-quote)*
lt	<	U+003C (60)	XML 1.0	less-than sign
gt	>	U+003E (62)	XML 1.0	greater-than sign

XML Namespaces

XML Namespaces provide a method to avoid element name conflicts.

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```


This XML carries information about a table (a piece of furniture):

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

A user or an XML application will not know how to handle these differences.

Solving the Name Conflict Using a Prefix

Name conflicts in XML can easily be avoided using a name prefix.

This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

In the example above, there will be no conflict because the two <table> elements have different names.

XML usage:

XML allows sets of documents which are all the same type to be created and handled consistently and without structural errors, because it provides a standardized way of describing, controlling, or allowing/disallowing particular types of document structure.

XML provides a common syntax for messaging systems for the exchange of information between applications. Previously, each messaging system had its own format and all were different, which made inter-system messaging unnecessarily messy, complex, and expensive. If everyone uses the same syntax it makes writing these systems much faster and more reliable.

XML is free. It doesn't belong to anyone, so it can't be hijacked or pirated. And you don't have to pay a fee to use it.

XML information can be manipulated programmatically so XML documents can be pieced together from disparate sources, or taken apart and re-used in different ways. They can be converted into any other format with no loss of information.

XML can also be used to store data in files or in databases. Applications can be written to store and retrieve information from the store, and generic applications can be used to display the data.

XML is Often a Complement to HTML

In many HTML applications, XML is used to store or transport data, while HTML is used to format and display the same data.

XML Separates Data from HTML

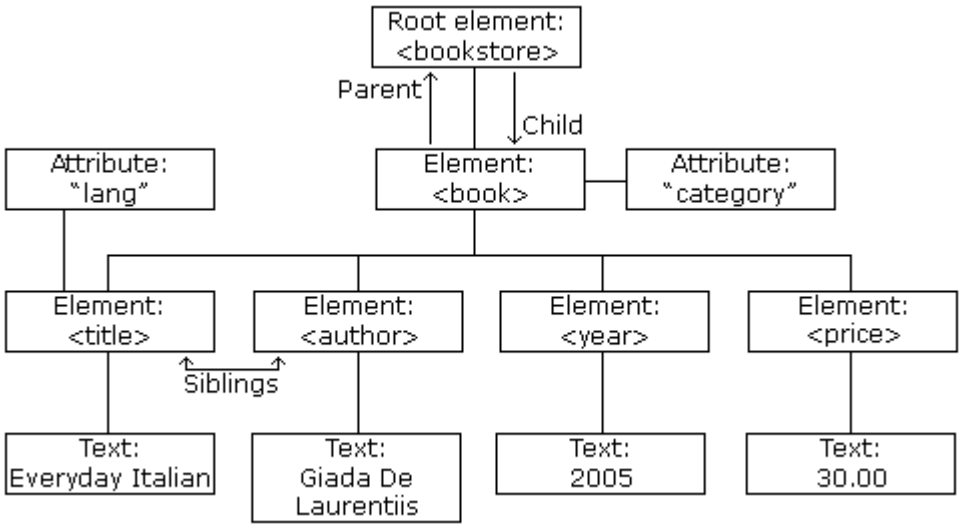
When displaying data in HTML, you should not have to edit the HTML file when the data changes.

With XML, the data can be stored in separate XML files.

With a few lines of JavaScript code, you can read an XML file and update the data content of any HTML page.

XML Tree Structure

XML documents form a tree structure that starts at "the root" and branches to "the leaves".



XML documents are formed as element trees.

An XML tree starts at a root element and branches from the root to child elements.

All elements can have sub elements (child elements):

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements.

Parent have children. Children have parents. Siblings are children on the same level (brothers and sisters).

XML Element

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

<price>29.99</price>

An element can contain:

- text
- attributes
- other elements
- or a mix of the above

```
<bookstore>
  <book category="children">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

In the example above:

<title>, <author>, <year>, and <price> have **text content** because they contain text (like 29.99).

<bookstore> and <book> have **element contents**, because they contain elements.

<book> has an **attribute** (category="children").

Empty XML Elements

XML elements can be defined as building blocks of an XML. Elements can behave as containers to hold text, elements, attributes, media objects or all of these.

Each XML document contains one or more elements, the scope of which are either delimited by start and end tags, or for empty elements, by an empty-element tag.

An element with no content is said to be empty.

In XML, you can indicate an empty element like this:

<element></element>

You can also use a so called self-closing tag:

<element />

Syntax

Following is the syntax to write an XML element:

<element-name attribute1 attribute2>
....content
</element-name>

Where

- element-name is the name of the element. The name its case in the start and end tags must match.

- attribute1, attribute2 are attributes of the element separated by white spaces. An attribute defines a property of the element. It associates a name with a value, which is a string of characters. An attribute is written as:

  name = "value"

  name is followed by an = sign and a string value inside double(" ") or single(' ') quotes.

XML Elements Rules

Following rules are required to be followed for XML elements:

- An element name can contain any alphanumeric characters. The only punctuation mark allowed in names are the hyphen (-), under-score (_) and period (.).
- Names are case sensitive. For example, Address, address, and ADDRESS are different names.
- Start and end tags of an element must be identical.
- An element, which is a container, can contain text or elements as seen in the above example.

XML Attributes:

Attributes are part of the XML elements. An element can have multiple unique attributes. Attribute gives more information about XML elements. To be more precise, they define properties of elements. An XML attribute is always a name-value pair.

Syntax

An XML attribute has following syntax:

<element-name attribute1 attribute2 >
....content..
< /element-name>

Where attribute1 and attribute2 has the following form:

name = "value"

Value has to be in double (" ") or single (' ') quotes. Here, attribute1 and attribute2 are unique attribute labels.

Attributes are used to add a unique label to an element, place the label in a category, add a Boolean flag, or otherwise associate it with some string of data. Following example demonstrates the use of attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE garden [
  <!ELEMENT garden (plants)*>
  <!ELEMENT plants (#PCDATA)>
  <!ATTLIST plants category CDATA #REQUIRED>
]>
<garden>
  <plants category="flowers" />
  <plants category="shrubs">
  </plants>
</garden>
```

Attributes are used to distinguish among elements of the same name. When you do not want to create a new element for every situation. Hence, use of an attribute can add a little more detail in differentiating two or more similar elements.

In the above example, we have categorized the plants by including attribute category and assigning different values to each of the elements. Hence we have two categories of plants, one flowers and other color. Hence we have two plant elements with different attributes.

Element Attribute Rules

Following are the rules that need to be followed for attributes:

- An attribute name must not appear more than once in the same start-tag or empty-element tag.
- An attribute must be declared in the Document Type Definition (DTD) using an Attribute-List Declaration.
- Attribute values must not contain direct or indirect entity references to external entities.
- The replacement text of any entity referred to directly or indirectly in an attribute value must not contain either less than sign <

Viewing XML Files

```
<?xml version="1.0" encoding="UTF-8"?>
- <note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
```

Look at the XML file above in your browser: note.xml

Most browsers will display an XML document with color-coded elements.

Often a plus (+) or minus sign (-) to the left of the elements can be clicked to expand or collapse the element structure.

XSL = Style Sheets for XML

XSLT is the most important part of XSL.

XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

A common way to describe the transformation process is to say that XSLT transforms an XML source-tree into an XML result-tree.

XSL consists of four parts:

- XSLT - a language for transforming XML documents
- XPath - a language for navigating in XML documents
- XSL-FO - a language for formatting XML documents (discontinued in 2013)
- XQuery - a language for querying XML documents

Example:

This example demonstrates the basics of setting up an XSLT transformation in a browser. The example will take an XML document that contains information (title, list of authors and body text) about an article and present it in a human readable form.

Figure shows the source of the basic XSLT example. The XML document (example.xml) contains the information about the article. Using the ?xml-stylesheet? processing instruction, it links to the XSLT stylesheet (example.xsl) via its href attribute.

An XSLT stylesheet starts with the xsl:stylesheet element, which contains all the templates used to create the final output. The example in Figure has two templates - one that matches the root node and one that matches Author nodes. The template that matches the root node outputs the

article's title and then says to process all templates that match Author nodes which are children of the Authors node.

Figure: Simple XSLT Example

XML Document (example.xml):

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="example.xsl"?>
<Article>
  <Title>My Article</Title>
  <Authors>
    <Author>Mr. Foo</Author>
    <Author>Mr. Bar</Author>
  </Authors>
  <Body>This is my article text.</Body>
</Article>
```

XSL Stylesheet (example.xsl):

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    Article - <xsl:value-of select="/Article/Title"/>
    Authors: <xsl:apply-templates select="/Article/Authors/Author"/>
  </xsl:template>

  <xsl:template match="Author">
    - <xsl:value-of select="." />
  </xsl:template>
```



```
</xsl:stylesheet>
```

Browser Output:

Article - My Article
Authors:
- Mr. Foo
- Mr. Bar

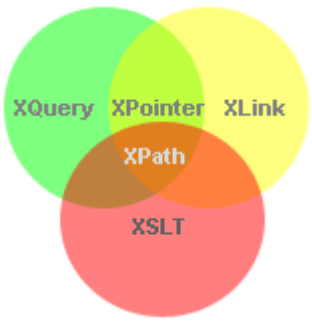
XPath:

XPath is used to navigate through elements and attributes in an XML document.

XPath is a syntax for defining parts of an XML document. It uses path expressions to navigate in XML documents. XPath contains a library of standard functions. XPath is a major element in XSLT. XPath is a W3C recommendation

XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the expressions you see when you work with a traditional computer file system.
XPath is a major element in the XSLT standard. Without XPath knowledge you will not be able to create XSLT documents.

XPath is also used in XQuery, XPointer and XLink



XLink:

Xml linking language, or xlink is an xml markup language and w3c specification that provides methods for creating internal and external links within xml documents, and associating metadata with those links.

xlink provides a framework for creating both basic unidirectional links and more complex linking structures.

Some important points about the xlink:

- xlink is short for the xml linking language
- xlink is a language for creating hyperlinks in xml documents
- xlink is similar to html links - but it is a lot more powerful
- xlink supports simple links(like html link system) and extended links (for linking multiple"more then one" resources together)
- with xlink, the links can be defined outside of the linked files
- xlink is a 'w3c recommendation'

XLink Attribute

Attribute	Value	Description
xlink:actuate	onLoad onRequest other none	Defines when the linked resource is read and shown: • onLoad - the resource should be loaded and shown when the document loads • onRequest - the resource is not read or shown before the link is clicked

xlink:href	*URL*	Specifies the URL to link to
xlink:show	embed new replace other none	Specifies where to open the link. Default is "replace"
xlink:type	simple extended locator arc resource title none	Specifies the type of link

Xpointer:

xpointer is a system for addressing components of xml base internet media.

xpointer language is divided among four specifications: a 'framework' which forms the basis for identifying xml fragments, a positional element addressing scheme, a scheme for namespaces, and a scheme for xpath-based addressing. There is no browser support for XPointer. But XPointer is used in other XML languages.

Some important points about xpointer:

- xpointer is short for the xml pointer language
- xpointer uses xpath expressions to navigate in the xml document.
- xpointer is a w3c recommendation

Module 12: (XML)

XML Validator:

XML validation is the process of checking a document written in XML (eXtensible Markup Language) to confirm that it is both well-formed and also "valid" in that it follows a defined structure. An XML document is said to be valid if its contents match with the elements, attributes and associated document type declaration (DTD), and if the document complies with the constraints expressed in it. Validation is dealt in two ways by the XML parser. They are:

- Well-formed XML document
- Valid XML document

Well-formed XML document

An XML document is said to be well-formed if it adheres to the following rules:

- Non DTD XML files must use the predefined character entities for amp(&), apos(single quote), gt(>), lt(<), quot(double quote).
- It must follow the ordering of the tag. i.e., the inner tag must be closed before closing the outer tag.
- Each of its opening tags must have a closing tag or it must be a self ending tag.(<title>....</title> or <title/>).
- It must have only one attribute in a start tag, which needs to be quoted.
- amp(&), apos(single quote), gt(>), lt(<), quot(double quote) entities other than these must be declared.

Valid XML document

If an XML document is well-formed and has an associated Document Type Declaration (DTD), then it is said to be a valid XML document. We will study more about DTD in the chapter XML – DTDs

XML – DTDs

The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language precisely. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.

An XML DTD can be either specified inside the document, or it can be kept in a separate document and then liked separately.

Syntax

Basic syntax of a DTD is as follows:

```
<!DOCTYPE element DTD identifier
[
  declaration1
  declaration2
  ........
]>
```

In the above syntax,

- The DTD starts with <!DOCTYPE delimiter.
- An element tells the parser to parse the document from the specified root element.
- DTD identifier is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet. If the DTD is pointing to external path, it is called External Subset.
- The square brackets [ ] enclose an optional list of entity declarations called Internal Subset.

DTD - XML Building Blocks

All XML documents are made up by the following building blocks:

- Elements
- Attributes
- Entities
- PCDATA
- CDATA

Elements

Elements are the main building blocks of both XML and HTML documents.

Examples of HTML elements are "body" and "table". Examples of XML elements could be "note" and "message". Elements can contain text, other elements, or be empty. Examples of empty HTML elements are "hr", "br" and "img".

Examples:

<body>some text</body>

<message>some text</message>

Attributes

Attributes provide extra information about elements.

Attributes are always placed inside the opening tag of an element. Attributes always come in name/value pairs. The following "img" element has additional information about a source file:

The name of the element is "img". The name of the attribute is "src". The value of the attribute is "computer.gif". Since the element itself is empty it is closed by a " /".

Entities

Some characters have a special meaning in XML, like the less than sign (<) that defines the start of an XML tag.

Most of you know the HTML entity: " " This "no-breaking-space" entity is used in HTML to insert an extra space in a document. Entities are expanded when a document is parsed by an XML parser.

The following entities are predefined in XML:

Entity References	Character
&lt;	<
&gt;	>
&amp;	&
&quot;	"
&apos;	'

PCDATA

PCDATA means parsed character data.

Think of character data as the text found between the start tag and the end tag of an XML element.

PCDATA is text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.

Tags inside the text will be treated as markup and entities will be expanded.

However, parsed character data should not contain any &, <, or > characters; these need to be represented by the & < and > entities, respectively.

CDATA

CDATA means character data.

CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

DTD Elements:

In a DTD, elements are declared with an ELEMENT declaration.

In a DTD, XML elements are declared with the following syntax:

<!ELEMENT element-name category>
or
<!ELEMENT element-name (element-content)>

Empty Elements

Empty elements are declared with the category keyword EMPTY:
<!ELEMENT element-name EMPTY>

Example:

<!ELEMENT br EMPTY>
XML example:

Elements with Parsed Character Data

Elements with only parsed character data are declared with #PCDATA inside parentheses:

<!ELEMENT element-name (#PCDATA)>
Example:
<!ELEMENT from (#PCDATA)>

Elements with any Contents

Elements declared with the category keyword ANY, can contain any combination of parsable data:

<!ELEMENT element-name ANY>
Example:
<!ELEMENT note ANY>

Elements with Children (sequences)

Elements with one or more children are declared with the name of the children elements inside parentheses:
<!ELEMENT element-name (child1)>
or
<!ELEMENT element-name (child1,child2,...)>
Example:
<!ELEMENT note (to,from,heading,body)>

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the "note" element is:

<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>

Declaring Only One Occurrence of an Element

<!ELEMENT element-name (child-name)>

Example:

<!ELEMENT note (message)>

The example above declares that the child element "message" must occur once, and only once inside the "note" element.

Declaring Minimum One Occurrence of an Element

<!ELEMENT element-name (child-name+)>

Example:

<!ELEMENT note (message+)>

The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

Declaring Zero or More Occurrences of an Element

<!ELEMENT element-name (child-name*)>

Example:

<!ELEMENT note (message*)>

The * sign in the example above declares that the child element "message" can occur zero or more times inside the "note" element.

Declaring Zero or One Occurrences of an Element

<!ELEMENT element-name (child-name?)>

Example:

<!ELEMENT note (message?)>

The ? sign in the example above declares that the child element "message" can occur zero or one time inside the "note" element.

Declaring either/or Content

<!ELEMENT note (to,from,header,(message|body))>

The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

Declaring Mixed Content

<!ELEMENT note (#PCDATA|to|from|header|message)*>

The example above declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

DTD – Attributes

In a DTD, attributes are declared with an ATTLIST declaration.

Declaring Attributes

An attribute declaration has the following syntax:

<!ATTLIST element-name attribute-name attribute-type attribute-value>

DTD example:

<!ATTLIST payment type CDATA "check">

XML example:

<payment type="check" />

The attribute-type can be one of the following:

Type Description

CDATA	The value is character data
(en1|en2|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is a predefined xml value

The **attribute-value** can be one of the following:

Value	Explanation
Value	The default value of the attribute
#REQUIRED	The attribute is required
#IMPLIED	The attribute is optional
#FIXED value	The attribute value is fixed

A Default Attribute Value

DTD:

<!ELEMENT square EMPTY>

<!ATTLIST square width CDATA "0">

Valid XML:

<square width="100" />

In the example above, the "square" element is defined to be an empty element with a "width" attribute of type CDATA. If no width is specified, it has a default value of 0.

#REQUIRED

Syntax

<!ATTLIST element-name attribute-name attribute-type #REQUIRED>

Example

DTD:

<!ATTLIST person number CDATA #REQUIRED>

Valid XML:

<person number="5677" />

Invalid XML:

<person />

Use the #REQUIRED keyword if you don't have an option for a default value, but still want to force the attribute to be present.

#IMPLIED

Syntax

<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

Example

DTD:

<!ATTLIST contact fax CDATA #IMPLIED>

Valid XML:

<contact fax="555-667788" />

Valid XML:

<contact />

Use the #IMPLIED keyword if you don't want to force the author to include an attribute, and you don't have an option for a default value.

#FIXED

Syntax

<!ATTLIST element-name attribute-name attribute-type #FIXED "value">

Example

DTD:
<!ATTLIST sender company CDATA #FIXED "Microsoft">
Valid XML:
<sender company="Microsoft" />
Invalid XML:
<sender company="BNU" />
Use the #FIXED keyword when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

Enumerated Attribute Values

Syntax

<!ATTLIST element-name attribute-name (en1|en2|..) default-value>

Example

DTD:

<!ATTLIST payment type (check|cash) "cash">

XML example:

<payment type="check" />

or

<payment type="cash" />

Enumerated attribute values are used when you want the attribute value to be one of a fixed set of legal values.

XML Elements vs. Attributes

Data can be stored in child elements or in attributes.

Examples:

<person sex="female">

<firstname>Anna</firstname>

<lastname>Smith</lastname>

</person>

<person>

```
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example sex is an attribute. In the last, sex is a child element. Both examples provide the same information.

There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

My Favorite Way

I like to store data in child elements.

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="12/11/2002">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

A date element is used in the second example:

```
<note>
  <date>12/11/2002</date>
  <to>Tove</to>
```

```
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

An expanded date element is used in the third: (THIS IS MY FAVORITE):

```
<note>
  <date>
    <day>12</day>
    <month>11</month>
    <year>2002</year>
  </date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Some of the problems with attributes are:

- attributes cannot contain multiple values (child elements can)
- attributes are not easily expandable (for future changes)
- attributes cannot describe structures (child elements can)
- attributes are more difficult to manipulate by program code
- attribute values are not easy to test against a DTD

If attributes as containers for data are used, you end up with documents that are difficult to read and maintain. Try to use elements to describe data. Use attributes only to provide information that is not relevant to the data.

DTD – Entities

Entities are used to define shortcuts to special characters.

Entities can be declared internal or external.

An Internal Entity Declaration

Syntax

<!ENTITY entity-name "entity-value">

Example

DTD Example:

<!ENTITY writer "Donald Duck.">

<!ENTITY copyright "Copyright BNU.">

XML example:

<author>&writer;&copyright;</author>

An entity has three parts: an ampersand (&), an entity name, and a semicolon (;).

An External Entity Declaration

Syntax

<!ENTITY entity-name SYSTEM "URI/URL">

DTD Example:

<!ENTITY writer SYSTEM "http://www.nouman.com/entities.dtd">

<!ENTITY copyright SYSTEM "http://www.vu.edu.pk/entities.dtd">

XML example:

<author>&writer;&copyright;</author>

XML Schema

An XML Schema describes the structure of an XML document.

The purpose of an XML Schema is to define the legal building blocks of an XML document:

- the elements and attributes that can appear in a document
- the number of (and order of) child elements
- data types for elements and attributes
- default and fixed values for elements and attributes

The <schema> Element

The <schema> element is the root element of every XML Schema:

<?xml version="1.0"?>

<xs:schema>

...

...

</xs:schema>

The <schema> element may contain some attributes. A schema declaration often looks

something like this:

<?xml version="1.0"?>

```
<xs:schema xmlns:xs="http://www.vu.edu.pk/XMLSchema"
targetNamespace="http://www.vu.edu.pk"
xmlns="http://www.vu.edu.pk"
elementFormDefault="qualified">
...
...
</xs:schema>
```

The following fragment:

xmlns:xs="http://www.vu.edu.pk/XMLSchema"

Indicates that the elements and data types used in the schema come from the "http://www.vu.edu.pk/XMLSchema" namespace. It also specifies that the elements and data types that come from the "http://www.vu.edu.pk/XMLSchema"namespace should be prefixed with xs:

This fragment:

targetNamespace=" http://www.vu.edu.pk"

Indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.vu.edu.pk"namespace.

This fragment:

xmlns="http://www.vu.edu.pk"

Indicates that the default namespace is “http://www.vu.edu.pk".

This fragment:

elementFormDefault="qualified"

Indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

XML on the Server

XML can easily be stored and generated by a standard web server.

XML files can be stored on an Internet server exactly the same way as HTML files.

Start Windows Notepad and write the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <from>Jani</from>
  <to>Tove</to>
  <message>Remember me this weekend</message>
</note>
```

Generating XML with PHP

XML can be generated on a server without any installed XML software.

To generate an XML response from the server using PHP, use following code:

```
<?php
header("Content-type: text/xml");
echo "<?xml version='1.0' encoding='UTF-8'?>";
```

```
echo "<note>";
echo "<from>Jani</from>";
echo "<to>Tove</to>";
echo "<message>Remember me this weekend</message>";
echo "</note>";
?>
```

The content type of the response header must be set to "text/xml".

Generating XML with ASP

To generate an XML response from the server - simply write the following code and save it as an ASP file on the web server:

```
<%
response.ContentType="text/xml"
response.Write("<?xml version='1.0' encoding='UTF-8'?>")
response.Write("<note>")
response.Write("<from>Jani</from>")
response.Write("<to>Tove</to>")
response.Write("<message>Remember me this weekend</message>")
response.Write("</note>")
%>
```

The content type of the response must be set to "text/xml".

Generating XML from a Database

XML can be generated from a database without any installed XML software.

To generate an XML database response from the server, simply write the following code and save it as an ASP file on the web server:

```
<%
response.ContentType = "text/xml"
set conn=Server.CreateObject("ADODB.Connection")
conn.provider="Microsoft.Jet.OLEDB.4.0;"
conn.open server.mappath("/datafolder/database.mdb")

sql="select fname,lname from tblGuestBook"
set rs=Conn.Execute(sql)

response.write("<?xml version='1.0' encoding='UTF-8'?>")
response.write("<guestbook>")
while (not rs.EOF)
response.write("<guest>")
response.write("<fname>" & rs("fname") & "</fname>")
response.write("<lname>" & rs("lname") & "</lname>")
```

```
response.write("</guest>")
rs.MoveNext()
wend

rs.close()
conn.close()
response.write("</guestbook>")
%>
```

Transforming XML with XSLT on the Server

This ASP transforms an XML file to XHTML on the server:

```
<%
'Load XML
set xml = Server.CreateObject("Microsoft.XMLDOM")
xml.async = false
xml.load(Server.MapPath("simple.xml"))

'Load XSL
set xsl = Server.CreateObject("Microsoft.XMLDOM")
xsl.async = false
xsl.load(Server.MapPath("simple.xsl"))

'Transform file
Response.Write(xml.transformNode(xsl))
```

```
%>
```

Example explained

- The first block of code creates an instance of the Microsoft XML parser (XMLDOM), and loads the XML file into memory.
- The second block of code creates another instance of the parser and loads the XSL file into memory.
- The last line of code transforms the XML document using the XSL document, and sends the result as XHTML.

Module 13 :(XML DOM)

DOM:

Document **O**bject **M**odel (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document like HTML and XML.

DOM defines the objects and properties and methods (interface) to access all XML elements.

The DOM is separated into 3 different parts:

- Core DOM - standard model for any structured document
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

HTML DOM

The HTML DOM defines a standard way for accessing and manipulating HTML documents. All HTML elements can be accessed through the HTML DOM. The HTML DOM defines the **objects, properties and methods** of all HTML elements.

Change the Value of an HTML Element

This example changes the value of an HTML element with id="demo":

Example

```
<h1 id="demo">This is a Heading</h1>
<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>
```


This example changes the value of the first <h1> element in an HTML document:

Example

```
<h1>This is a Heading</h1>
<h1>This is a Heading</h1>
<script>
document.getElementsByTagName("h1")[0].innerHTML = "Hello World!";
</script>
```

Even if the HTML document containes only ONE <h1> element you still have to specify the array index [0], because the getElementsByTagName() method always returns an array.

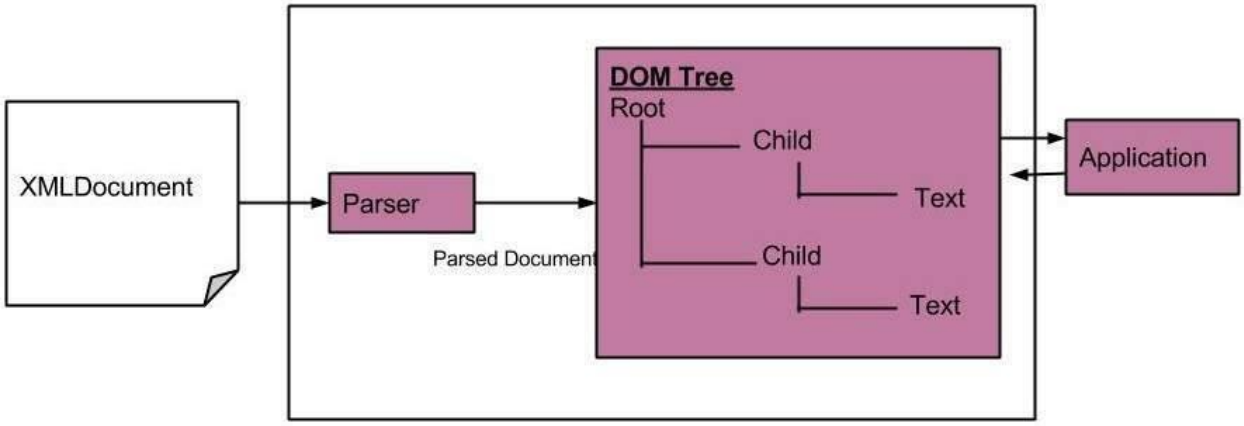
XML DOM

XML DOM is a standard object model for XML. XML documents have a hierarchy of informational units called nodes; DOM is a standard programming interface of describing those nodes and the relationships between them.

As XML DOM also provides an API that allows a developer to add, edit, move or remove nodes at any point on the tree in order to create an application.

The XML DOM presents an XML document as a tree-structure. The HTML DOM presents an HTML document as a tree-structure.

Below is the diagram for the DOM structure which depicts that parser evaluates an XML document as a DOM structure by traversing through each nodes.



Loading an XML File

The XML file used in the examples below is books.xml.

This example reads "books.xml" into xmlDoc and retrieves the text value of the first <title> element in books.xml:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
   if (xhttp.readyState == 4 && xhttp.status == 200) {
   myFunction(xhttp);
   }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
   var xmlDoc = xml.responseXML;
   document.getElementById("demo").innerHTML =
   xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
}
</script>
```

```
</body>
</html>
```

Example Explained

xmlDoc - the XML DOM object created by the parser.

getElementsByTagName("title")[0] - get the first <title> element

childNodes[0] - the first child of the <title> element (the text node)

nodeValue - the value of the node (the text itself)

Loading an XML String

This example loads a text string into an XML DOM object, and extracts the info from it with JavaScript:

Example

```
<html>
<body>

<p id="demo"></p>

<script>
var text, parser, xmlDoc;

text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";

parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");
```

```
document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script>

</body>
</html>
```

Programming Interface

The DOM models XML as a set of node objects. The nodes can be accessed with JavaScript or other programming languages. In this tutorial we use JavaScript.

The programming interface to the DOM is defined by a set standard properties and methods.

Properties are often referred to as something that is (i.e. nodename is "book").

Methods are often referred to as something that is done (i.e. delete "book").

XML DOM Properties

These are some typical DOM properties:

- x.nodeName - the name of x
- x.nodeValue - the value of x
- x.parentNode - the parent node of x
- x.childNodes - the child nodes of x
- x.attributes - the attributes nodes of x

In the list above, x is a node object.

XML DOM Methods

- x.getElementsByTagName(*name*) - get all elements with a specified tag name
- x.appendChild(*node*) - insert a child node to x
- x.removeChild(*node*) - remove a child node from x

Advantages

- XML DOM is language and platform independent.
- XML DOM is traversable - Information in XML DOM is organized in a hierarchy which allows developer to navigate around the hierarchy looking for specific information.
- XML DOM is modifiable - It is dynamic in nature providing developer a scope to add, edit, move or remove nodes at any point on the tree.

Disadvantages

- It consumes more memory (if the XML structure is large) as program written once remains in memory all the time until and unless removed explicitly.
- Due to the larger usage of memory its operational speed, compared to SAX is slower.

XML DOM Nodes

A DOM document is a collection of nodes or pieces of information, organized in a hierarchy. Some types of nodes may have child nodes of various types and others are leaf nodes that cannot have anything below them in the document structure. Below is a list of the node types, and which node types they may have as children:

- ***Document*** **--** Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
- ***DocumentFragment*** **--** Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ***EntityReference*** **--** Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ***Element*** **--** Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
- ***Attr*** **--** Text, EntityReference
- ***ProcessingInstruction*** **--** no children
- ***Comment*** **--** no children
- ***Text*** **--** no children

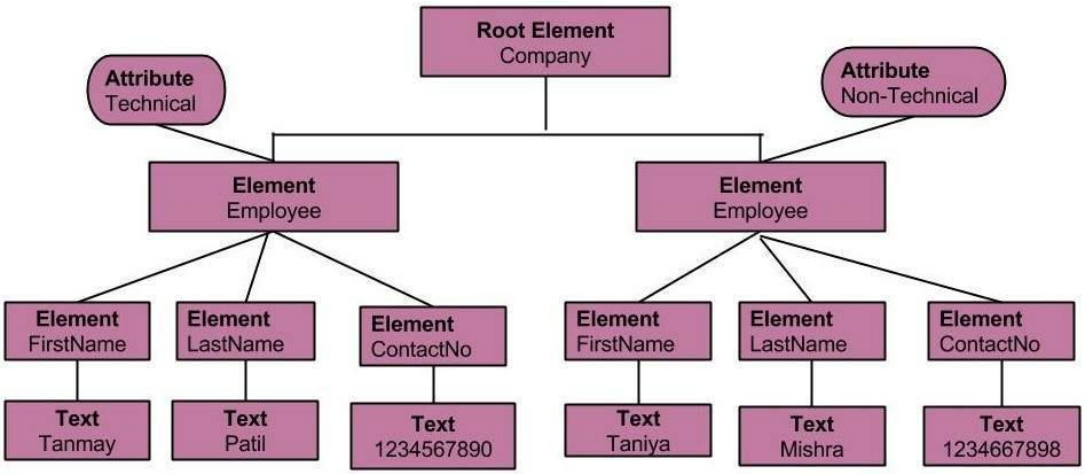
- ***CDATASection*** -- no children
- ***Entity*** -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ***Notation*** -- no children

Example

Consider the DOM representation of the following XML document node.xml.

```
<?xml version="1.0"?>
<Company>
  <Employee category="technical">
    <FirstName>Tanmay</FirstName>
    <LastName>Patil</LastName>
    <ContactNo>1234567890</ContactNo>
  </Employee>
  <Employee category="non-technical">
    <FirstName>Taniya</FirstName>
    <LastName>Mishra</LastName>
    <ContactNo>1234667898</ContactNo>
  </Employee>
</Company>
```

The Document Object Model of the above XML document would be as follows:



From the above diagram we can interface:

- Node object can have only one parent node object. This occupies the position above all the nodes. Here it is Company.

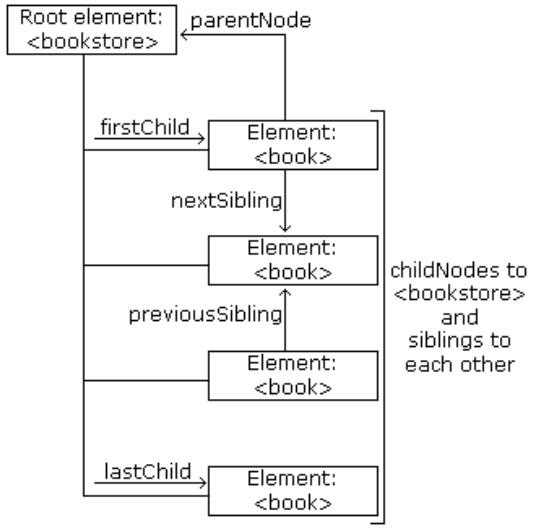
- The parent node can have multiple nodes called as child nodes. These child nodes can have additional node called as attribute node. In the above example we have two attribute nodes Technical and Non-Technical. The attribute node is not actually a child of the element node, but is still associated with it.

- These child nodes in turn can have multiple child nodes. The text within the nodes is called as text node.

- The node objects at the same level are called as siblings.

XML DOM Node Relationship

- Top node is the root
- Every node, except the root, has exactly one parent node
- A node can have multiple children
- A leaf node with no children
- Siblings have same parent



First Child - Last Child

Look at the following XML fragment:

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

In the XML above, the <title> element is the first child of the <book> element, and the <price> element is the last child of the <book> element.

Furthermore, the <book> element is the parent node of the <title>, <author>, <year>, and <price> elements.

The DOM Identifies:

- The objects to represent the interface and manipulate the document.
- The relationship among the objects and interfaces.

The XMLHttpRequest Object

All modern browsers have a built-in XMLHttpRequest object to request data from a server. All major browsers have a built-in XML parser to access and manipulate XML.

The XMLHttpRequest Object

- The XMLHttpRequest object can be used to request data from a web server.
- The XMLHttpRequest object is a developers dream, because you can:
- Update a web page without reloading the page
- Request data from a server - after the page has loaded
- Receive data from a server - after the page has loaded
- Send data to a server - in the background

Sending an XMLHttpRequest

All modern browsers have a built-in XMLHttpRequest object.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
   if (xhttp.readyState == 4 && xhttp.status == 200) {
      // Action to be performed when the document is read;
   }
};
xhttp.open("GET", "filename", true);
xhttp.send();
}
```

Creating an XMLHttpRequest Object

The first line in the example above creates an XMLHttpRequest objet:

```
var xhttp = new XMLHttpRequest();
```

The onreadystatechange Event

The readyState property holds the status of the XMLHttpRequest. The onreadystatechange event is triggered every time the readyState changes.

During a server request, the readyState changes from 0 to 4:

0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

In the onreadystatechange property, specify a function to be executed when the readyState changes:

xhttp.onreadystatechange = function()

When readyState is 4 and status is 200, the response is ready:

if (xhttp.readyState == 4 && xhttp.status == 200)

XMLHttpRequest Properties and Methods

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
open(*method, url, async*)	Specifies the type of request *method*: the type of request: GET or POST *url*: the file location *async*: true (asynchronous) or false (synchronous)
send()	Sends a request to the server (used for GET)
send(*string*)	Sends a request string to the server (used for POST)
onreadystatechange	A function to be called when the readyState property changes
readyState	The status of the XMLHttpRequest 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready

status	200: OK 404: Page not found
responseText	The the response data as a string
responseXML	The response data as XML data

Access Across Domains

For security reasons, modern browsers do not allow access across domains.

This means that both the web page and the XML file it tries to load, must be located on the same server.

The response Text Property

The responseText property returns the response as a string. If you want to use the response as a text string, use the responseText property:

Example

document.getElementById("demo").innerHTML = xmlhttp.responseText;

The response XML Property

The responseXML property returns the response as an XML DOM object. If you want to use the response as an XML DOM object, use the responseXML property:

Example

Request the file cd_catalog.xml and use the response as an XML DOM object:

xmlDoc = xmlhttp.responseXML;

txt = "";

x = xmlDoc.getElementsByTagName("ARTIST");


```
for (i = 0; i < x.length; i++) {
   txt += x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
```

GET or POST

GET is simpler and faster than POST, and can be used in most cases.

However, always use POST requests when:

- A cached file is not an option (update a file or database on the server)
- Sending a large amount of data to the server (POST has no size limitations)
- Sending user input (which can contain unknown characters), POST is more robust and secure than GET

The url - A File On a Server

The url parameter of the open() method, is an address to a file on a server:

```
xmlhttp.open("GET", "xmlhttp_info.txt", true);
```

The file can be any kind of file, like .txt and .xml, or server scripting files like .asp and .php (which can perform actions on the server before sending the response back).

Asynchronous - True or False

To send the request asynchronously, the async parameter of the open() method has to be set to true:

```
xmlhttp.open("GET", "xmlhttp_info.txt", true);
```

Sending asynchronously requests is a huge improvement for web developers. Many of the tasks performed on the server are very time consuming.

By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:

- Execute other scripts while waiting for server response
- Deal with the response when the response is ready

Async = true

When using async = true, specify a function to execute when the response is ready in the onreadystatechange event:

Example

```
xmlhttp.onreadystatechange = function() {
   if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
      document.getElementById("demo").innerHTML = xmlhttp.responseText;
   }
};
xmlhttp.open("GET", "xmlhttp_info.txt", true);
xmlhttp.send();
```

Async = false

To use async = false, change the third parameter in the open() method to false:

```
xmlhttp.open("GET", "xmlhttp_info.txt", false);
```

Using async = false is not recommended, but for a few small requests this can be ok.

When you use async = false, do NOT write an onreadystatechange function - just put the code after the send() statement:

Example

```
xmlhttp.open("GET", "xmlhttp_info.txt", false);
xmlhttp.send();
document.getElementById("demo").innerHTML = xmlhttp.responseText;
```

XML Parser

All modern browsers have a built-in XML parser. The XML Document Object Model (the XML DOM) contains a lot of methods to access and edit XML. However, before an XML document can be accessed, it must be loaded into an XML DOM object. An XML parser can read plain text and convert it into an XML DOM object.

Parsing a Text String

This example parses a text string into an XML DOM object, and extracts the information from it with JavaScript:

Example

```
<html>
<body>

<p id="demo"></p>

<script>
var text, parser, xmlDoc;

text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";

parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");

document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
```

```
</script>

</body>
</html>
```

Accessing Nodes

You can access a node in three ways:

1. By using the getElementsByTagName() method

2. By looping through (traversing) the nodes tree.

3. By navigating the node tree, using the node relationships.

The getElementsByTagName() Method

getElementsByTagName() returns all elements with a specified tag name.

Syntax

```
node.getElementsByTagName("tagname");
```

Example

The following example returns all <title> elements under the x element:

```
x.getElementsByTagName("title");
```

The example above only returns <title> elements under the x node. To return all <title> elements in the XML document use:

```
xmlDoc.getElementsByTagName("title");
```

where xmlDoc is the document itself (document node).

DOM Node List

The getElementsByTagName() method returns a node list. A node list is an array of nodes.

x = xmlDoc.getElementsByTagName("title");

The <title> elements in x can be accessed by index number. To access the third <title> you can write:

y = x[2];

The index starts at 0.

DOM Node List Length

The length property defines the length of a node list (the number of nodes).

You can loop through a node list by using the length property:

Example

```
var x = xmlDoc.getElementsByTagName("title");
for (i = 0; i <x.length; i++) {
  // do something for each node
  }
```

Node Types

The **documentElement** property of the XML document is the root node.

The **nodeName** property of a node is the name of the node.

The **nodeType** property of a node is the type of the node.

Traversing Nodes

The following code loops through the child nodes, that are also element nodes, of the root node:

Example

```
txt = "";
x = xmlDoc.documentElement.childNodes;

for (i = 0; i <x.length; i++) {
  // Process only element nodes (type 1)
  if (x[i].nodeType == 1) {
    txt += x[i].nodeName + "<br>";
  }
}
```

Example explained:

- Suppose you have loaded "books.xml" into xmlDoc
- Get the child nodes of the root element (xmlDoc)
- For each child node, check the node type. If the node type is "1" it is an element node
- Output the name of the node if it is an element node

Navigating Node Relationships

The following code navigates the node tree using the node relationships:

Example

```
x = xmlDoc.getElementsByTagName("book")[0];
xlen = x.childNodes.length;
y = x.firstChild;

txt = "";
for (i = 0; i <xlen; i++) {
  // Process only element nodes (type 1)
  if (y.nodeType == 1) {
    txt += y.nodeName + "<br>";
  }
```

```
  y = y.nextSibling;
}
```

Example explained:

- Suppose you have loaded "books.xml"into xmlDoc
- Get the child nodes of the first book element
- Set the "y" variable to be the first child node of the first book element
- For each child node (starting with the first child node "y"):
- Check the node type. If the node type is "1" it is an element node
- Output the name of the node if it is an element node
- Set the "y" variable to be the next sibling node, and run through the loop again

Node Properties

In the XML DOM, each node is an object. Objects have methods and properties that can be accessed and manipulated by JavaScript.

Three important node properties are:

- nodeName
- nodeValue
- nodeType

The node Name Property

The nodeName property specifies the name of a node.

- nodeName is read-only
- nodeName of an element node is the same as the tag name
- nodeName of an attribute node is the attribute name
- nodeName of a text node is always #text
- nodeName of the document node is always #document

The node Value Property

The nodeValue property specifies the value of a node.

- nodeValue for element nodes is undefined
- nodeValue for text nodes is the text itself
- nodeValue for attribute nodes is the attribute value

Get the Value of an Element

The following code retrieves the text node value of the first <title> element:

Example

```
var x = xmlDoc.getElementsByTagName("title")[0].childNodes[0];
var txt = x.nodeValue;
```

Result: txt = "Everyday Italian"

Example explained:

- Suppose you have loaded "books.xml" into xmlDoc
- Get text node of the first <title> element node
- Set the txt variable to be the value of the text node

Change the Value of an Element

The following code changes the text node value of the first <title> element:

Example

```
var x = xmlDoc.getElementsByTagName("title")[0].childNodes[0];
x.nodeValue = "Easy Cooking";
```

Example explained:

- Suppose you have loaded "books.xml" into xmlDoc
- Get text node of the first <title> element node
- Change the value of the text node to "Easy Cooking"

The nodeType Property

The nodeType property specifies the type of node. nodeType is read only.

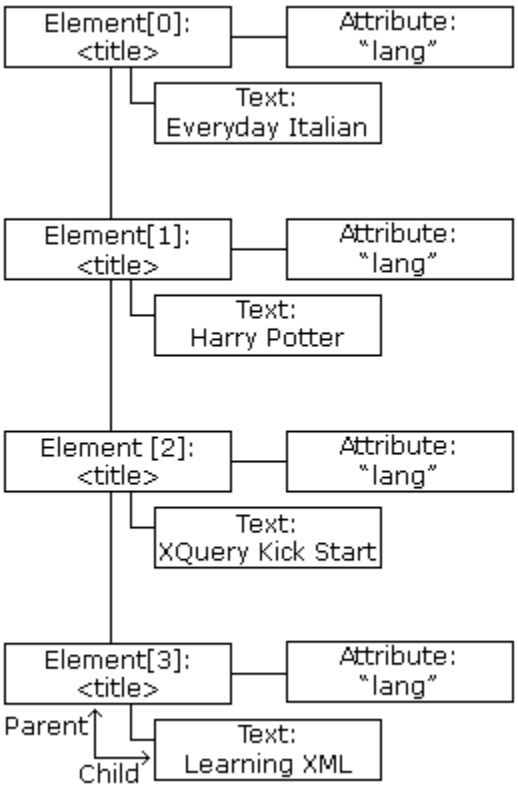
The most important node types are:

Node type	NodeType
Element	1
Attribute	2
Text	3
Comment	8
Document	9

Node List

When using properties or methods like childNodes or getElementsByTagName(), a node list object is returned. A node list object represents a list of nodes, in the same order as in the XML. Nodes in the node list are accessed with index numbers starting from 0.

The following image represents a node list of the <title> elements in "books.xml":



Suppose "books.xml" is loaded into the variable xmlDoc.

This code fragment returns a node list of title elements in "books.xml":

x = xmlDoc.getElementsByTagName("title");

After the execution of the statement above, x is a node list object.

The following code fragment returns the text from the first <title> element in the node list (x):

Example

var txt = x[0].childNodes[0].nodeValue;

Node List Length

A node list object keeps itself up-to-date. If an element is deleted or added, the list is automatically updated. The length property of a node list is the number of nodes in the list.

This code fragment returns the number of <title> elements in "books.xml":

```
x = xmlDoc.getElementsByTagName('title').length;
```

After the execution of the statement above, the value of x will be 4. The length of the node list can be used to loop through all the elements in the list.

This code fragment uses the length property to loop through the list of <title> elements:

Example

```
x = xmlDoc.getElementsByTagName('title');
xLen = x.length;

for (i = 0; i <xLen; i++) {
   txt += x[i].childNodes[0].nodeValue) + " ";
}
```

Output:

Everyday Italian
Harry Potter
XQuery Kick Start
Learning XML

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Set the x variable to hold a node list of all title elements
3. Collect the text node values from <title> elements

DOM Attribute List (Named Node Map)

The attributes property of an element node returns a list of attribute nodes. This is called a named node map, and is similar to a node list, except for some differences in methods and properties. A attribute list keeps itself up-to-date. If an attribute is deleted or added, the list is automatically updated.

This code fragment returns a list of attribute nodes from the first <book> element in "books.xml":

```
x = xmlDoc.getElementsByTagName('book')[0].attributes;
```

After the execution of the code above, x.length = is the number of attributes and x.getNamedItem() can be used to return an attribute node.

This code fragment gets the value of the "category" attribute, and the number of attributes, of a book:

Example

```
x = xmlDoc.getElementsByTagName("book")[0].attributes;

txt = x.getNamedItem("category").nodeValue + " " + x.length);
```

Output:

cooking 1

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Set the x variable to hold a list of all attributes of the first <book> element
- Get the value of the "category" attribute and the length of the attribute list

Traverse Node Tree

Traversing means looping through or traveling across the node tree.

Often you want to loop an XML document, for example: when you want to extract the value of each element. This is called "Traversing the node tree".

The example below loops through all child nodes of <book>, and displays their names and values:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var x, i ,xmlDoc;
var txt = "";
var text = "<book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book>";

parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");

// documentElement always represents the root node
x = xmlDoc.documentElement.childNodes;
for (i = 0; i < x.length ;i++) {
   txt += x[i].nodeName + ": " + x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
```

```
</script>

</body>
</html>
```

Output:

title: Everyday Italian
author: Giada De Laurentiis
year: 2005

Example explained:

- Load the XML string into xmlDoc
- Get the child nodes of the root element
- For each child node, output the node name and the node value of the text node

Browser Differences in DOM Parsing

All modern browsers support the W3C DOM specification. However, there are some differences between browsers. One important difference is:
The way they handle white-spaces and new lines

DOM - White Spaces and New Lines

XML often contains new line, or white space characters, between nodes. This is often the case when the document is edited by a simple editor like Notepad.
The following example (edited by Notepad) contains CR/LF (new line) between each line and two spaces in front of each child node:

```
<book>
  <title>Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
```

Internet Explorer 9 and earlier do NOT treat empty white-spaces, or new lines as text nodes, while other browsers do.

The following example will output the number of child nodes the root element (of books.xml) has. IE9 and earlier will output 4 child nodes, while IE10 and later versions, and other browsers will output 9 child nodes:

Example

```
function myFunction(xml) {
var xmlDoc = xml.responseXML;
   x = xmlDoc.documentElement.childNodes;
   document.getElementById("demo").innerHTML =
   "Number of child nodes: " + x.length;
}
```

PCDATA - Parsed Character Data

XML parsers normally parse all the text in an XML document. When an XML element is parsed, the text between the XML tags is also parsed:

<message>This text is also parsed</message>

The parser does this because XML elements can contain other elements, as in this example, where the <name> element contains two other elements (first and last):

<name><first>Bill</first><last>Gates</last></name>

and the parser will break it up into sub-elements like this:

```
<name>
  <first>Bill</first>
  <last>Gates</last>
</name>
```

Parsed Character Data (PCDATA) is a term used about text data that will be parsed by the XML parser.

CDATA - (Unparsed) Character Data

The term CDATA is used about text data that should not be parsed by the XML parser.

Characters like "<" and "&" are illegal in XML elements.

"<" will generate an error because the parser interprets it as the start of a new element.

"&" will generate an error because the parser interprets it as the start of a character entity.

Some text, like JavaScript code, contains a lot of "<" or "&" characters. To avoid errors script code can be defined as CDATA. Everything inside a CDATA section is ignored by the parser.

A CDATA section starts with "<![CDATA[" and ends with "]]>":

```
<script>
<![CDATA[
function matchwo(a,b) {
   if (a < b && a < 0) {
      return 1;
   } else {
      return 0;
   }
}
]]>
</script>
```

In the example above, everything inside the CDATA section is ignored by the parser. A CDATA section cannot contain the string "]]>". Nested CDATA sections are not allowed. The "]]>" that marks the end of the CDATA section cannot contain spaces or line breaks.

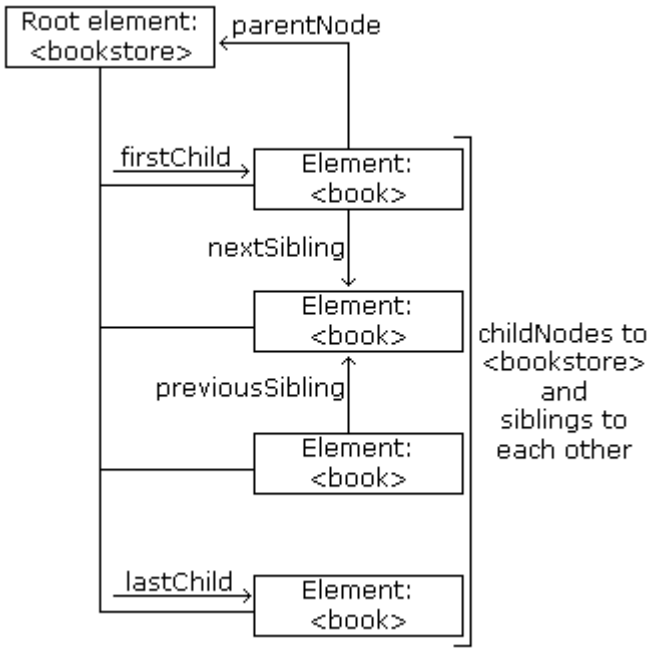
Navigating Nodes

Nodes can be navigated using node relationships. Accessing nodes in the node tree via the relationship between nodes, is often called "navigating nodes".

In the XML DOM, node relationships are defined as properties to the nodes:

- parentNode
- childNodes
- firstChild
- lastChild
- nextSibling
- previousSibling

The following image illustrates a part of the node tree and the relationship between nodes in books.xml:



Parent Node

All nodes have exactly one parent node. The following code navigates to the parent node of <book>:

Example

```
function myFunction(xml) {
var xmlDoc = xml.responseXML;
   var x = xmlDoc.getElementsByTagName("book")[0];
   document.getElementById("demo").innerHTML = x.parentNode.nodeName;
}
```

Example explained:

1. Load "books.xml" into xmlDoc
2. Get the first <book> element
3. Output the node name of the parent node of "x"

Avoid Empty Text Nodes

Firefox, and some other browsers, will treat empty white-spaces or new lines as text nodes, Internet Explorer will not. This causes a problem when using the properties: firstChild, lastChild, nextSibling, previousSibling. To avoid navigating to empty text nodes (spaces and new-line characters between element nodes), we use a function that checks the node type:

```
function get_nextSibling(n) {
   var y = n.nextSibling;
   while (y.nodeType! = 1) {
      y = y.nextSibling;
   }
   return y;}
```


The function above allows you to use get_nextSibling(node) instead of the property node.nextSibling.

Code explained:

Element nodes are type 1. If the sibling node is not an element node, it moves to the next nodes until an element node is found. This way, the result will be the same in both Internet Explorer and Firefox.

Get the First Child Element

The following code displays the first element node of the first <book>:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState == 4 && xhttp.status == 200) {
        myFunction(xhttp);
    }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
   var xmlDoc = xml.responseXML;
   var x = get_firstChild(xmlDoc.getElementsByTagName("book")[0]);
   document.getElementById("demo").innerHTML = x.nodeName;
```

```
}

//check if the first node is an element node
function get_firstChild(n) {
   var y = n.firstChild;
   while (y.nodeType != 1) {
      y = y.nextSibling;
   }
   return y;
}
</script>

</body>
</html>
```

Output:

Title

Example explained:

1. Load "books.xml" into xmlDoc
2. Use the get_firstChild function on the first <book> element node to get the first child node that is an element node
3. Output the node name of first child node that is an element node

Get Node Values

The nodeValue property is used to get the text value of a node. The getAttribute() method returns the value of an attribute.

Get the Value of an Element

In the DOM, everything is a node. Element nodes do not have a text value. The text value of an element node is stored in a child node. This node is called a text node.

The getElementsByTagName Method

The getElementsByTagName() method returns a node list of all elements, with the specified tag name, in the same order as they appear in the source document.

For example: "books.xml" has been loaded into xmlDoc.

This code retrieves the first <title> element:

```
var x = xmlDoc.getElementsByTagName("title")[0];
```

The ChildNodes Property

The childNodes property returns a list of an element's child nodes.

The following code retrieves the text node of the first <title> element:

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.childNodes[0];
```

The nodeValue Property

The nodeValue property returns the text value of a text node. The following code retrieves the text value of the text node of the first <title> element:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.childNodes[0];
```



```
z = y.nodeValue;
```

Resul in z: "Everyday Italian"

Get the Value of an Attribute

In the DOM, attributes are nodes. Unlike element nodes, attribute nodes have text values. The way to get the value of an attribute, is to get its text value. This can be done using the getAttribute() method or using the nodeValue property of the attribute node.

Get an Attribute Value - getAttribute()

The getAttribute() method returns an attribute's value. The following code retrieves the text value of the "lang" attribute of the first <title> element:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
txt = x.getAttribute("lang");
```

Get an Attribute Value - getAttributeNode()

The getAttributeNode() method returns an attribute node. The following code retrieves the text value of the "lang" attribute of the first <title> element:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.getAttributeNode("lang");
txt = y.nodeValue;
```

Change Node Values

The nodeValue property is used to change a node value. The setAttribute() method is used to change an attribute value.

Change the Value of an Element

Element nodes do not have a text value. The text value of an element node is stored in a child node. This node is called a text node.

Change the Value of a Text Node

The nodeValue property can be used to change the value of a text node. For Example: "books.xml" has been loaed into xmlDoc. This code changes the text node value of the first <title> element:

Example

xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue = "new content"

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Get the first child node of the <title> element
- Change the node value to "new content"

Change the Value of an Attribute

Attributes are nodes. Unlike element nodes, attribute nodes have text values. The way to change the value of an attribute, is to change its text value. This can be done using the setAttribute() method or setting the nodeValue property of the attribute node.

Change an Attribute Using setAttribute()

The setAttribute() method changes the value of an attribute. If the attribute does not exist, a new attribute is created. This code changes the category attribute of the <book> element:

Example

```
xmlDoc.getElementsByTagName("book")[0].setAttribute("category","food");
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Get the first <book> element
- Change the "category" attribute value to "food"

Change an Attribute Using nodeValue

The nodeValue property is the value of an attribute node. Changing the value property changes the value of the attribute.

Example

```
xmlDoc.getElementsByTagName("book")[0].getAttributeNode("category").nodeValue = "food";
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Get the "category" attribute of the first <book> element
- Change the attribute node value to "food"

Remove Nodes

The removeChild() method removes a specified node. The removeAttribute() method removes a specified attribute.

Remove an Element Node

The removeChild() method removes a specified node. When a node is removed, all its child nodes are also removed. This code will remove the first <book> element from the loaded xml:

Example

```
y = xmlDoc.getElementsByTagName("book")[0];

xmlDoc.documentElement.removeChild(y);
```

Example explained:

- Suppose "books.xml" is loaded xmlDoc
- Set the variable y to be the element node to remove
- Remove the element node by using the removeChild() method from the parent node

Remove Myself - Remove the Current Node

The removeChild() method is the only way to remove a specified node. When you have navigated to the node you want to remove, it is possible to remove that node using the parentNode property and the removeChild() method:

Example

```
x = xmlDoc.getElementsByTagName("book")[0];

x.parentNode.removeChild(x);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Set the variable y to be the element node to remove
- Remove the element node by using the parentNode property and the removeChild() method

Remove a Text Node

The removeChild() method can also be used to remove a text node:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.childNodes[0];
x.removeChild(y);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Set the variable x to be the first title element node
- Set the variable y to be the text node to remove
- Remove the element node by using the removeChild() method from the parent node

It is not very common to use removeChild() just to remove the text from a node. The nodeValue property can be used instead.

Clear a Text Node

The nodeValue property can be used to change the value of a text node:

Example

```
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue = "";
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Get the first title element's first child node.
- Use the nodeValue property to clear the text from the text node

Remove an Attribute Node by Name

The removeAttribute() method removes an attribute node by its name.

Example: removeAttribute('category')

This code removes the "category" attribute in the first <book> element:

Example

```
x = xmlDoc.getElementsByTagName("book");
x[0].removeAttribute("category");
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Use getElementsByTagName() to get book nodes
- Remove the "category" attribute form the first book element node

Remove Attribute Nodes by Object

The removeAttributeNode() method removes an attribute node, using the node object as parameter.

Example: removeAttributeNode(x)

This code removes all the attributes of all <book> elements:

Example

```
x = xmlDoc.getElementsByTagName("book");

for (i = 0; i < x.length; i++) {
   while (x[i].attributes.length > 0) {
      attnode = x[i].attributes[0];
      old_att = x[i].removeAttributeNode(attnode);
   }
}
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Use getElementsByTagName() to get all book nodes
- For each book element check if there are any attributes
- While there are attributes in a book element, remove the attribute

Replace Nodes

The replaceChild() method replaces a specified node. The nodeValue property replaces text in a text node.

Replace an Element Node

The replaceChild() method is used to replace a node. The following code fragment replaces the first <book> element:

Example

```
xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.documentElement;

//create a book element, title element and a text node
newNode=xmlDoc.createElement("book");
newTitle=xmlDoc.createElement("title");
newText=xmlDoc.createTextNode("A Notebook");

//add the text node to the title node,
newTitle.appendChild(newText);
//add the title node to the book node
newNode.appendChild(newTitle);

y=xmlDoc.getElementsByTagName("book")[0]
//replace the first book node with the new node
x.replaceChild(newNode,y);
```

Example explained:

- Load "books.xml" into xmlDoc
- Create a new element node <book>
- Create a new element node <title>
- Create a new text node with the text "A Notebook"
- Append the new text node to the new element node <title>
- Append the new element node <title> to the new element node <book>
- Replace the first <book> element node with the new <book> element node

Replace Data In a Text Node

The replaceData() method is used to replace data in a text node. The replaceData() method has three parameters:

- offset - Where to begin replacing characters. Offset value starts at zero
- length - How many characters to replace
- string - The string to insert

Example

```
xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.getElementsByTagName("title")[0].childNodes[0];

x.replaceData(0,8,"Easy");
```

Example explained:

- Load "books.xml" into xmlDoc
- Get the text node of the first <title> element node
- Use the replaceData method to replace the eight first characters from the text node with "Easy"

Use the nodeValue Property Instead

It is easier to replace the data in a text node using the nodeValue property. The following code fragment will replace the text node value in the first <title> element with "Easy Italian":

Example

```
xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.getElementsByTagName("title")[0].childNodes[0];

x.nodeValue="Easy Italian";
```

Example explained:

- Load "books.xml" into xmlDoc
- Get the text node of the first <title> element node
- Use the nodeValue property to change the text of the text node

Create a New Element Node

The createElement() method creates a new element node:

Example

```
newElement = xmlDoc.createElement("edition");

xmlDoc.getElementsByTagName("book")[0].appendChild(newElement);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new element node <edition>
- Append the element node to the first <book> element

Create a New Attribute Node

The createAttribute() is used to create a new attribute node:

Example

```
newAtt = xmlDoc.createAttribute("edition");
newAtt.nodeValue = "first";

xmlDoc.getElementsByTagName("title")[0].setAttributeNode(newAtt);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new attribute node "edition"
- Set the value of the attribute node to "first"
- Add the new attribute node to the first <title> element

Create an Attribute Using setAttribute()

Since the setAttribute() method creates a new attribute if the attribute does not exist, it can be used to create a new attribute.

Example

```
exmlDoc.getElementsByTagName('book')[0].setAttribute("edition","first");
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Set the attribute "edition" value to "first" for the first <book> element

Create a Text Node

The createTextNode() method creates a new text node:

Example

```
newEle = xmlDoc.createElement("edition");
newText = xmlDoc.createTextNode("first");
newEle.appendChild(newText);
```



```
xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new element node <edition>
- Create a new text node with the text "first"
- Append the new text node to the element node
- Append the new element node to the first <book> element

Create a CDATA Section Node

The createCDATASection() method creates a new CDATA section node.

Example

```
newCDATA = xmlDoc.createCDATASection("Special Offer & Book Sale");

xmlDoc.getElementsByTagName("book")[0].appendChild(newCDATA);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new CDATA section node
- Append the new CDATA node to the first <book> element

Loop through, and add a CDATA section, to all <book> elements: Try it yourself

Create a Comment Node

The createComment() method creates a new comment node.

Example

```
newComment = xmlDoc.createComment("Revised March 2015");

xmlDoc.getElementsByTagName("book")[0].appendChild(newComment);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc using
- Create a new comment node
- Append the new comment node to the first <book> element

Add a Node - appendChild()

The appendChild() method adds a child node to an existing node. The new node is added (appended) after any existing child nodes.

This code fragment creates an element (<edition>), and adds it after the last child of the first <book> element:

Example

```
newEle = xmlDoc.createElement("edition");

xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new node <edition>
- Append the node to the first <book> element

This code fragment does the same as above, but the new element is added with a value:

Example

```
newEle = xmlDoc.createElement("edition");
newText=xmlDoc.createTextNode("first");
newEle.appendChild(newText);
xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new node <edition>
- Create a new text node "first"

- Append the text node to the <edition> node
- Append the <addition> node to the <book> element

Insert a Node - insertBefore()

The insertBefore() method inserts a node before a specified child node. This method is useful when the position of the added node is important:

Example

```
newNode = xmlDoc.createElement("book");

x = xmlDoc.documentElement;
y = xmlDoc.getElementsByTagName("book")[3];

x.insertBefore(newNode,y);
```

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Create a new element node <book>
- Insert the new node in front of the last <book> element node

If the second parameter of insertBefore() is null, the new node will be added after the last existing child node.

x.insertBefore(newNode,null) and x.appendChild(newNode) will both append a new child node to x.

Add a New Attribute

The setAttribute() method sets the value of an attribute.

Example

```
xmlDoc.getElementsByTagName('book')[0].setAttribute("edition","first");
```

Example explained:

- Suppose "books.xml" has been loaded into xmlDoc
- Set the value of the attribute "edition" to "first" for the first <book> element

If the attribute already exists, the setAttribute() method will overwrite the existing value.

Add Text to a Text Node - insertData()

The insertData() method inserts data into an existing text node. The insertData() method has two parameters:

- offset - Where to begin inserting characters (starts at zero)
- string - The string to insert

The following code fragment will add "Easy" to the text node of the first <title> element of the loaded XML:

Example

```
xmlDoc.getElementsByTagName("title")[0].childNodes[0].insertData(0,"Easy ");
```

Copy a Node

The cloneNode() method creates a copy of a specified node. The cloneNode() method has a parameter (true or false). This parameter indicates if the cloned node should include all attributes and child nodes of the original node. The following code fragment copies the first <book> node and appends it to the root node of the document:

Example

```
oldNode = xmlDoc.getElementsByTagName('book')[0];
newNode = oldNode.cloneNode(true);
xmlDoc.documentElement.appendChild(newNode);
```

Result:

Everyday Italian

Harry Potter

XQuery Kick Start

Learning XML

Everyday Italian

Example explained:

- Suppose "books.xml" is loaded into xmlDoc
- Get the node to copy (oldNode)
- Clone the node into "newNode"
- Append the new node to the the root node of the XML document

Node Types

The following table lists the different W3C node types, and which node types they may have as children:

Node Type	Description	Children
Document	Represents the entire document (the root-node of the DOM tree)	Element (max. one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Represents a "lightweight" Document object, which can hold a portion of a document	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	Provides an interface to the entities defined for the document	None
ProcessingInstruction	Represents a processing instruction	None
EntityReference	Represents an entity reference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference

Element	Represents an element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Represents an attribute	Text, EntityReference
Text	Represents textual content in an element or attribute	None
CDATASection	Represents a CDATA section in a document (text that will NOT be parsed by a parser)	None
Comment	Represents a comment	None
Entity	Represents an entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	Represents a notation declared in the DTD	None

Module 14: Introduction to JAXP

The Java API for XML Processing (JAXP) is for processing XML data using applications written in the Java programming language. JAXP leverages the parser standards Simple API for XML Parsing (SAX) and Document Object Model (DOM) so that you can choose to parse your data as a stream of events or to build an object representation of it. JAXP also supports the Extensible Stylesheet Language Transformations (XSLT) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts. Finally, as of version 1.4, JAXP implements the Streaming API for XML (StAX) standard.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which lets you plug in an implementation of the SAX or DOM API. The pluggability layer also allows you to plug in an XSL processor, letting you control how your XML data is displayed.

Overview of the Packages

The SAX and DOM APIs are defined by the XML-DEV group and by the W3C, respectively. The libraries that define those APIs are as follows:

- **javax.xml.parsers:** The JAXP APIs, which provide a common interface for different vendors' SAX and DOM parsers.
- **org.w3c.dom:** Defines the Document class (a DOM) as well as classes for all the components of a DOM.
- **org.xml.sax:** Defines the basic SAX APIs.
- **javax.xml.transform:** Defines the XSLT APIs that let you transform XML into other forms.

- **javax.xml.stream:** Provides StAX-specific transformation APIs.

SAX Packages

The Simple API for XML (SAX) is the event-driven, serial-access mechanism that does element-by-element processing. The API for this level reads and writes XML to a data repository or the web. For server-side and high-performance applications, you will want to fully understand this level. But for many applications, a minimal understanding will suffice.

DOM Packages

The DOM API is generally an easier API to use. It provides a familiar tree structure of objects. You can use the DOM API to manipulate the hierarchy of application objects it encapsulates. The DOM API is ideal for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user.

Comparison:

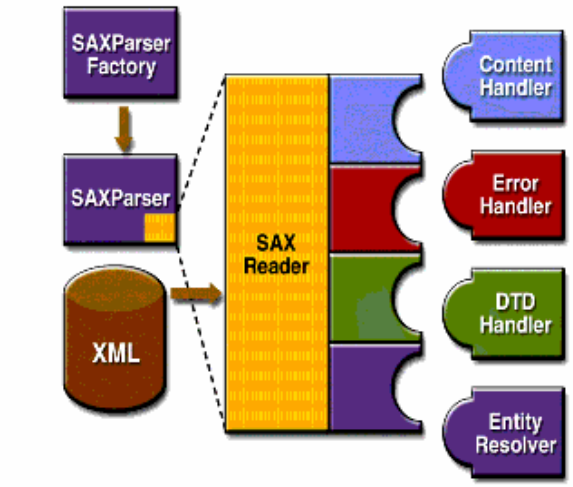
On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU- and memory-intensive. For that reason, the SAX API tends to be preferred for server-side applications and data filters that do not require an in-memory representation of the data.

The XSLT APIs defined in javax.xml.transform let you write XML data to a file or convert it into other forms. As shown in the XSLT section of this tutorial, you can even use it in conjunction with the SAX APIs to convert legacy data to XML.

Finally, the StAX APIs defined in javax.xml.stream provide a streaming Java technology-based, event-driven, pull-parsing API for reading and writing XML documents. StAX offers a simpler programming model than SAX and more efficient memory management than DOM.

Simple API for XML APIs

The basic outline of the SAX parsing APIs is shown below. To start the process, an instance of the SAXParserFactory class is used to generate an instance of the parser.



The parser wraps a SAXReader object. When the parser's parse() method is invoked, the reader invokes one of several callback methods implemented in the application. Those methods are defined by the interfaces ContentHandler, ErrorHandler, DTDHandler, and EntityResolver.

Summary of the key SAX APIs:

SAX Parser Factory

A SAXParserFactory object creates an instance of the parser determined by the system property, javax.xml.parsers.SAXParserFactory.

SAX Parser

The SAXParser interface defines several kinds of parse() methods. In general, you pass an XML data source and a DefaultHandler object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

SAX Reader

The SAXParser wraps a SAXReader. It is the SAXReader that carries on the conversation with the SAX event handlers you define.

Default Handler

A DefaultHandler implements the ContentHandler, ErrorHandler, DTDHandler, and EntityResolver interfaces (with null methods), so you can override only the ones you are interested in.

Content Handler

Methods such as startDocument, endDocument, startElement, and endElement are invoked when an XML tag is recognized. This interface also defines the methods characters() and processingInstruction(), which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

Error Handler

Methods error(), fatalError(), and warning() are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). Sometimes, the application may be able to recover from a validation error. Other times, it may need to generate an exception. To ensure the correct handling, you will need to supply your own error handler to the parser.

DTD Handler

Defines methods you will generally never be called upon to use. Used when processing a DTD to recognize and act on declarations for an unparsed entity.

Entity Resolver

The resolveEntity method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN - a public identifier, or name, that is unique in the web space. The public identifier may be specified in addition to the URL. The EntityResolver can then use the public identifier instead of the URL to find the document-for example, to access a local copy of the document if one exists.

A typical application implements most of the ContentHandler methods, at a minimum. Because the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may also want to implement the ErrorHandler methods.

SAX Packages

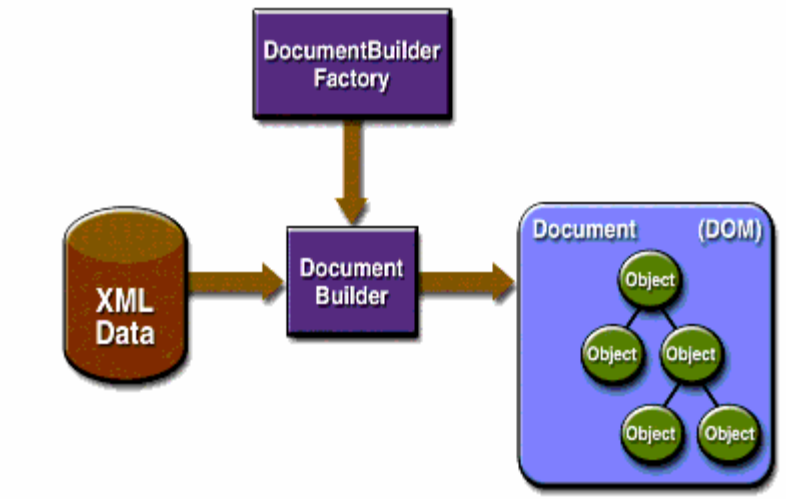
The SAX parser is defined in the packages listed in the following Table.

Packages	Description
org.xml.sax	Defines the SAX interfaces. The name org.xml is the package prefix that was settled on by the group that defined the SAX API.
org.xml.sax.ext	Defines SAX extensions that are used for doing more sophisticated SAX processing-for example, to process a document type definition (DTD) or to see the detailed syntax for a file.
org.xml.sax.helpers	Contains helper classes that make it easier to use SAX-for example, by defining a default handler that has null methods for all the interfaces, so that you only need to override the ones you actually want to implement.

javax.xml.parsers Defines the SAXParserFactory class, which returns the SAXParser. Also defines exception classes for reporting errors.

Document Object Model APIs

Figure DOM APIs



javax.xml.parsers.DocumentBuilderFactory class is used to get a DocumentBuilder instance, and that instance can also be used to produce a Document object that conforms to the DOM specification. The builder which is get, in fact, is determined by the system property javax.xml.parsers.DocumentBuilderFactory, which selects the factory implementation that is used to produce the builder.

DocumentBuilder newDocument() method can also be used to create an empty Document that implements the org.w3c.dom.Document interface. Alternatively, you can use one of the builder's parse methods to create a Document from existing XML data.

Although they are called objects, the entries in the DOM tree are actually fairly low-level data structures. For example, consider this structure: <color>blue</color>. There is an element node for the color tag, and under that there is a text node that contains the data, blue!

The Document Object Model implementation is defined in the packages listed in the following Table.

Package	Description
org.w3c.dom	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
javax.xml.parsers	Defines the DocumentBuilderFactory class and the DocumentBuilder class, which returns an object that implements the W3C Document interface. The factory that is used to create the builder is determined by the javax.xml.parsers system property, which can be set from the command line or overridden when invoking the new Instance method. This package also defines the ParserConfigurationException class for reporting errors.

Overview

SAX (the Simple API for XML) is an event-based parser for xml documents. Unlike a DOM parser, a SAX parser creates no parse tree. SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element.

- Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document
- Tokens are processed in the same order that they appear in the document
- Reports the application program the nature of tokens that the parser has encountered as they occur
- The application program provides an "event" handler that must be registered with the parser
- As the tokens are identified, callback methods in the handler are invoked with the relevant information

Content Handler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

- **void startDocument()** - Called at the beginning of a document.
- **void endDocument()** - Called at the end of a document.
- **void startElement(String uri, String localName, String qName, Attributes atts)** - Called at the beginning of an element.
- **void endElement(String uri, String localName,String qName)** - Called at the end of an element.
- **void characters(char[] ch, int start, int length)** - Called when character data is encountered.
- **void ignorableWhitespace(char[] ch, int start, int length)** - Called when a DTD is present and ignorable whitespace is encountered.
- **void processingInstruction(String target, String data)** - Called when a processing instruction is recognized.
- **void setDocumentLocator(Locator locator))** - Provides a Locator that can be used to identify positions in the document.
- **void skippedEntity(String name)** - Called when an unresolved entity is encountered.
- **void startPrefixMapping(String prefix, String uri)** - Called when a new namespace mapping is defined.
- **void endPrefixMapping(String prefix)** - Called when a namespace definition ends its scope.

JDOM

JDOM is an open source, java based library to parse XML document and it is typically java developer friendly API. It has a straightforward API, is a lightweight and fast, and is optimized for the Java programmer. It is java optimized, it uses java collection like List and Arrays. It works with DOM and SAX APIs and combines the better of the two. It is of low memory footprint and is nearly as fast as SAX.

Advantages

JDOM gives java developers flexibility and easy maintainability of xml parsing code. It is light weight and quick API.

JDOM classes

The JDOM defines several Java classes. Here are the most common classes:

- **Document** - Represents the entire XML document. A Document object is often referred to as a DOM tree.
- **Element** - Represents an XML element. Element object has methods to manipulate its child elements,its text, attributes and namespaces.
- **Attribute** Represents an attribute of an element. Attribute has method to get and set the value of attribute. It has parent and attribute type.
- **Text** Represents the text of XML tag.
- **Comment** Represents the comments in a XML document.

Steps to Using JDOM

Following are the steps used while parsing a document using JDOM Parser.

- Import XML-related packages.
- Create a SAXBuilder
- Create a Document from a file or stream
- Extract the root element
- Examine attributes
- Examine sub-elements

Import XML-related packages

```
import java.io.*;
import java.util.*;
import org.jdom2.*;
```

Create a DocumentBuilder

```
SAXBuilder saxBuilder = new SAXBuilder();
```

Create a Document from a file or stream

```
File inputFile = new File("input.txt");
SAXBuilder saxBuilder = new SAXBuilder();
Document document = saxBuilder.build(inputFile);
```

Extract the root element

```
Element classElement = document.getRootElement();
```

Examine attributes

```
//returns specific attribute
getAttribute("attributeName");
```

Examine sub-elements

```
//returns a list of subelements of specified name
getChildren("subelementName");
//returns a list of all child nodes
getChildren();
//returns first child node
getChild("subelementName");
```

JDOM XML parser modify an existing XML file:

- Add a new element
- Update existing element attribute
- Update existing element value
- Delete existing element

Example

- Add a new “age” element under staff
- Update the staff attribute id = 2
- Update salary value to 7000
- Delete “firstname” element under staff

Introduction to JAXB

Java Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. JAXB also provides a way to generate XML schema from Java objects.

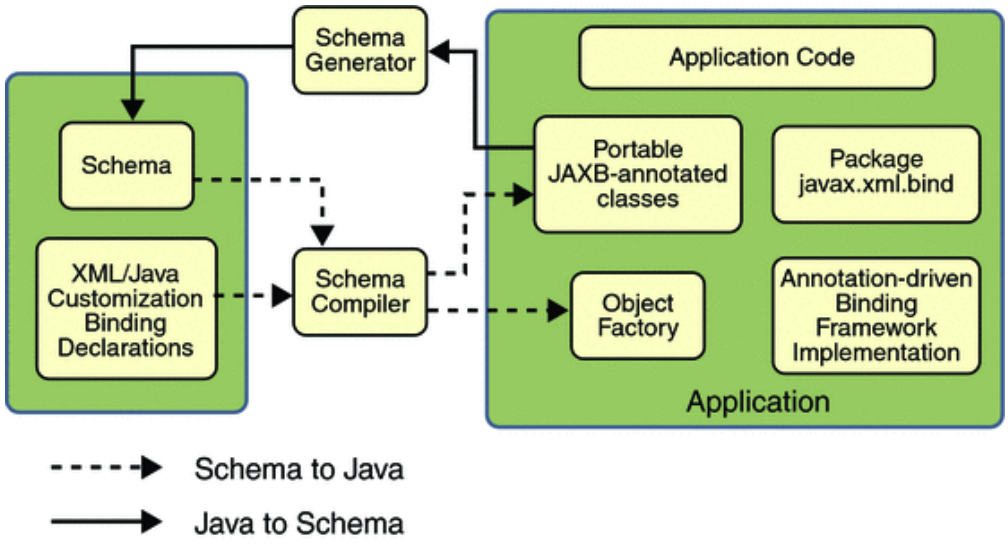
JAXB 2.0 includes several important improvements to JAXB 1.0:

- Support for all W3C XML Schema features. (JAXB 1.0 did not specify bindings for some of the W3C XML Schema features.)
- Support for binding Java-to-XML, with the addition of the javax.xml.bind.annotation package to control this binding. (JAXB 1.0 specified the mapping of XML Schema-to-Java, but not Java-to-XML Schema.)
- A significant reduction in the number of generated schema-derived classes.
- Additional validation capabilities through the JAXP 1.3 validation APIs.
- Smaller runtime libraries.

Architectural Overview

The following figure shows the components that make up a JAXB implementation.

Figure: JAXB Architectural Overview



A JAXB implementation consists of the following architectural components:

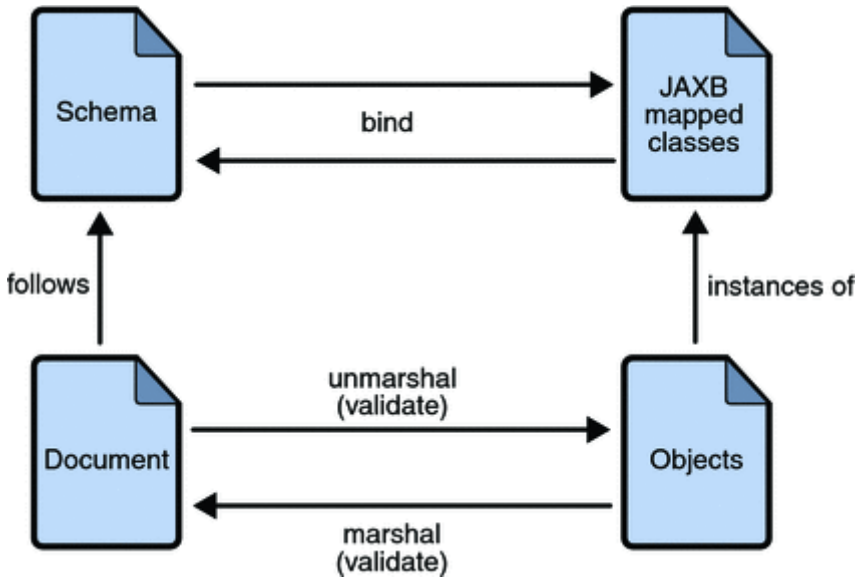
- **Schema compiler:** Binds a source schema to a set of schema-derived program elements. The binding is described by an XML-based binding language.
- **Schema generator:** Maps a set of existing program elements to a derived schema. The mapping is described by program annotations.

- **Binding runtime framework:** Provides unmarshalling (reading) and marshalling (writing) operations for accessing, manipulating, and validating XML content using either schema-derived or existing program elements.

The JAXB Binding Process

The following figure shows what occurs during the JAXB binding process.

Figure: Steps in the JAXB Binding Process



The general steps in the JAXB data binding process are:

1. **Generate classes:** An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.

2. **Compile classes:** All of the generated classes, source files, and application code must be compiled.
3. **Unmarshal:** XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files and documents, such as DOM nodes, string buffers, SAX sources, and so forth.
4. **Generate content tree:** The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.
5. **Validate (optional):** The unmarshalling process involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. **Process content:** The client application can modify the XML data represented by the Java content tree by using interfaces generated by the binding compiler.
7. **Marshal:** The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

More about Unmarshalling

Unmarshalling provides a client application the ability to convert XML data into JAXB-derived Java objects.

More about Marshalling

Marshalling provides a client application the ability to convert a JAXB-derived Java object tree into XML data.

By default, the Marshaller uses UTF-8 encoding when generating XML data.

Client applications are not required to validate the Java content tree before marshalling. There is also no requirement that the Java content tree be valid with respect to its original schema to marshal it into XML data.

More about Validation

Validation is the process of verifying that an XML document meets all the constraints expressed in the schema. JAXB 1.0 provided validation at unmarshal time and also enabled on-demand validation on a JAXB content tree. JAXB 2.0 only allows validation at unmarshal and marshal time. A web service processing model is to be lax in reading in data and strict on writing it out. To meet that model, validation was added to marshal time so users could confirm that they did not invalidate an XML document when modifying the document in JAXB form.

XML Binding:

XML data binding refers to a means of representing information in an XML document as a business object in computer memory. This allows applications to access the data in the XML from the object rather than using the DOM or SAX to retrieve the data from a direct representation of the XML itself.

An XML data binder accomplishes this by automatically creating a mapping between elements of the XML schema of the document we wish to bind and members of a class to be represented in memory.

When this process is applied to convert an XML document to an object, it is called unmarshalling. The reverse process, to serialize an object as XML, is called marshalling.

JAXB is Java Architecture for XML Binding

JAXB, stands for Java Architecture for XML Binding, using JAXB annotation to convert Java object to / from XML file.

- Marshalling – Convert a Java object into a XML file.
- Unmarshalling – Convert XML content into a Java Object.

Annotations

JAXB uses Java's annotations for augmenting the generated classes with additional information that bridges the gap between what is described by an XML schema and the information available from a set of Java class definitions. Adding such annotations to existing Java classes prepares them for being used by JAXB's runtime.

The Java code generated by the JAXB schema compiler contains *annotations* providing metadata on packages, classes, fields and methods. Together, this metadata is intended to reflect the information contained in an XML schema.

Annotations can be easily retrieved from their target construct with methods contained in classes such as java.lang.Class or java.lang.reflect.Field. Each annotation type has its own set of attributes, which are accessed in the usual way. Given some class, an annotation of type XmlType can be retrieved with

```
Class clazz = ...;
XmlType typeAnn = clazz.getAnnotation( XmlType.class );
```

If the result of the annotation getter is not null, annotation element values may be obtained by calling methods on the returned XmlType object. To retrieve the name of the corresponding XML Schema type you would write

```
String schemaName = typeAnn.name();
```

Classes that can be used for marshalling and unmarshalling XML need not be generated by the JAXB schema compiler. It is equally possible to write these classes by hand, adding the JAXB annotations.

Marshalling

Marshalling is the process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one program to another. It simplifies complex communication, using custom/complex objects to communicate instead of *primitives*.

Unmarshalling

A simple approach for unmarshalling an XML document consists of the creation of a JAXB context and the call to unmarshal the document. A JAXBContext object provides the entry point to the JAXB API and maintains the binding information between XML and Java. One way of creating a context instance is by calling the static method newInstance with a list of colon separated names of the packages containing the JAXB schema-derived classes. From this context, an Unmarshaller object is obtained, which functions as the driver for processing an XML text to create the equivalent set of Java objects. It offers several unmarshal methods, accepting a wide range of object types as the source for XML text data. The method shown below illustrates this with a single package containing the class of the type defining the top level element of the XML document.

```
public <T> T unmarshal( Class<T> docClass, InputStream inputStream )
   throws JAXBException {
   String packageName = docClass.getPackage().getName();
   JAXBContext jc = JAXBContext.newInstance( packageName );
   Unmarshaller u = jc.createUnmarshaller();
   JAXBElement<T> doc = (JAXBElement<T>)u.unmarshal( inputStream );
   return doc.getValue();
}
```

The return value of the call to JAXB's unmarshal is a representation of the root node of the parsed XML document in an instance of JAXBElement<T>. If we're not interested in the tag of the root element we might just as well return the extracted content value.

Module 15: (Servlets)

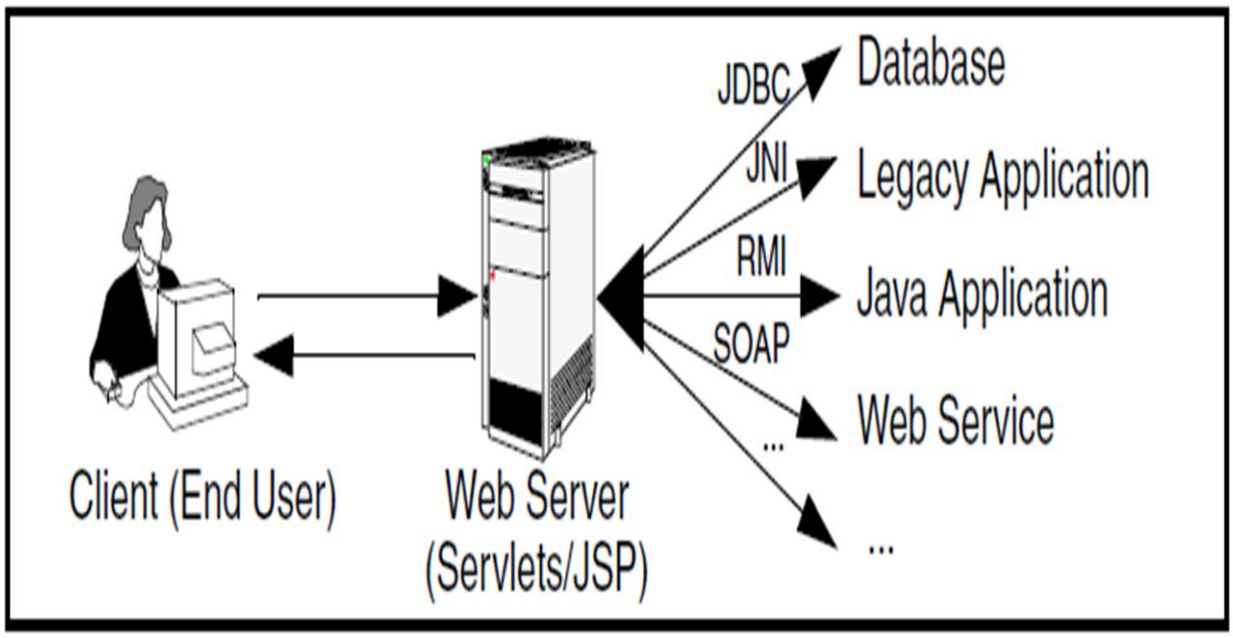
Servlet technology is used to create web application (resides at server side and generates dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there was many disadvantages of this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

Servlet can be described in many ways, depending on the context.

- Servlet is a technology i.e. used to create web application.
- Servlet is an API that provides many interfaces and classes including documentations.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.

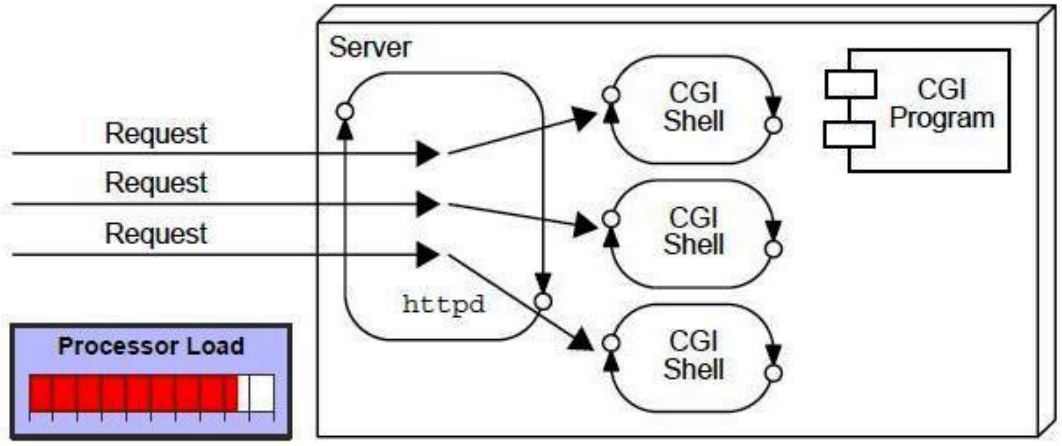


What is web application?

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.



The basic benefits of servlet are as follows:

- Better performance: because it creates a thread for each request not process.
- Portability: because it uses java language.
- Robust: Servlets are managed by JVM so no need to worry about momory leak, garbage collection etc.
- Secure: because it uses java language..

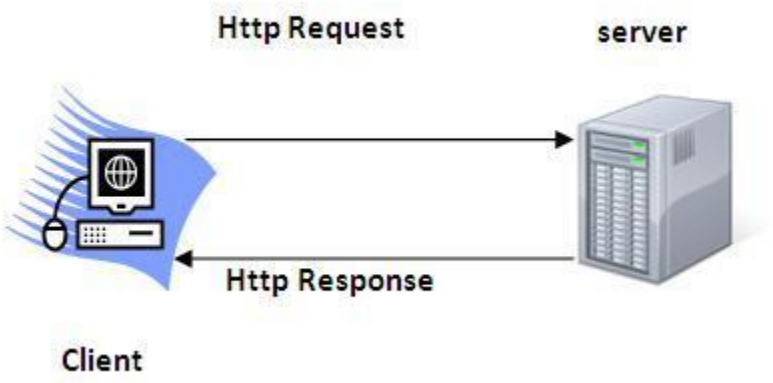
The basic terminology used in servlet are given below:

HTTP
HTTP Request Types
Difference between Get and Post method
Container
Server and Difference between web server and application server
Content Type
Introduction of XML
Deployment

HTTP (Hyper Text Transfer Protocol)

- Http is the protocol that allows web servers and browsers to exchange data over the web.

- It is a request response protocol.
- Http uses reliable TCP connections by default on TCP port 80.
- It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user by default.



Http Request Methods

Every request has a header that tells the status of the client. There are many request methods. Get and Post requests are mostly used.

The http request methods are:

GET
POST
HEAD
PUT
DELETE
OPTIONS
TRACE

HTTP Request	Description

GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

Get Request

Data is sent in request header in case of get request. It is the default request type. Let's see what information are sent to the server.

Post Request

In case of post request original data is sent in message body. Let's see how information are passed to the server

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the init () method.
- The servlet calls service() method to process a client's request.
- The servlet is terminated by calling the destroy() method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

The init() method :

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
  // Initialization code...
}
```

The service() method :

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

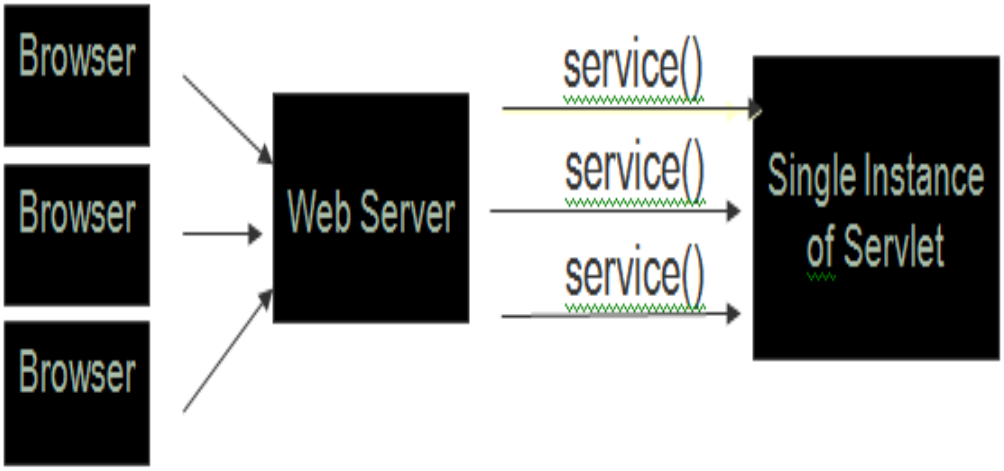
Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

```
public void service(ServletRequest request,
                    ServletResponse response)
    throws ServletException, IOException{
```

}

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.



The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
  throws ServletException, IOException {
  // Servlet code
}
```

The destroy() method :

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy() {
  // Finalization code...
}
```

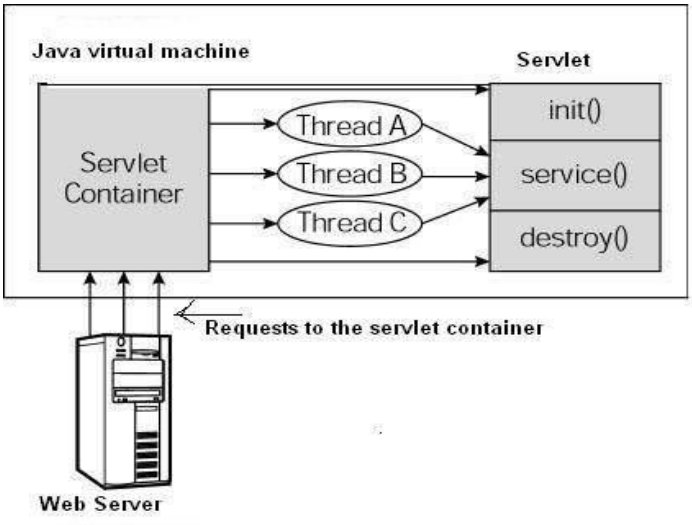
Architecture Diagram:

The following figure depicts a typical servlet life-cycle scenario.

First the HTTP requests coming to the server are delegated to the servlet container.

The servlet container loads the servlet before invoking the service() method.

Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



Steps to create a servlet example

There are given 6 steps to create a servlet example. These steps are required for all the servers.

The servlet example can be created by three ways:

- By implementing Servlet interface,
- By inheriting GenericServlet class, (or)
- By inheriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

Here, we are going to use apache tomcat server in this example. The steps are as follows:

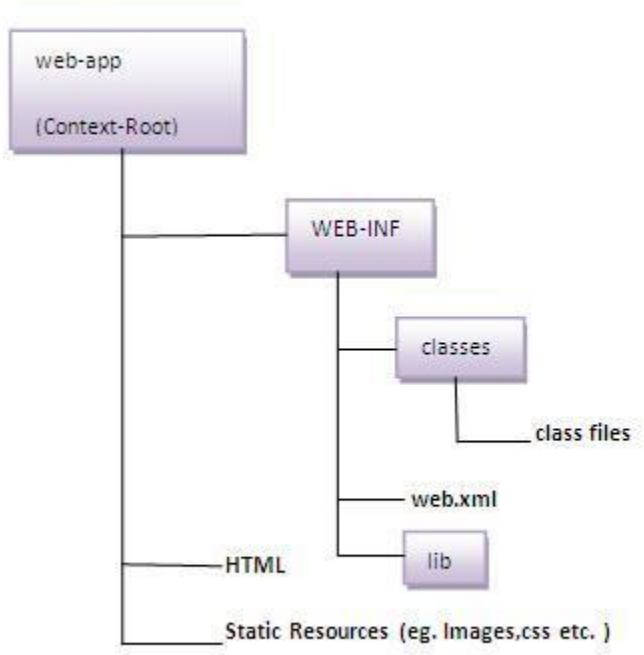
- Create a directory structure
- Create a Servlet
- Compile the Servlet

- Create a deployment descriptor
- Start the server and deploy the project
- Access the servlet

1) Create a directory structures

The directory structure defines that where to put the different types of files so that web container may get the information and respond to the client.

The Sun Microsystem defines a unique standard to be followed by all the server vendors. Let's see the directory structure that must be followed to create the servlet.



Servlet class file must be in the classes folder. The web.xml file must be under the WEB-INF folder.

2) Create a Servlet

There are three ways to create the servlet.

- By implementing the Servlet interface
- By inheriting the GenericServlet class
- By inheriting the HttpServlet class

The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as doGet(), doPost, doHead() etc.

In this example we are going to create a servlet that extends the HttpServlet class. In this example, we are inheriting the HttpServlet class and providing the implementation of the doGet() method. Notice that get request is the default request.

DemoServlet.java

```
1. import javax.servlet.http.*;
2. import javax.servlet.*;
3. import java.io.*;
4. public class DemoServlet extends HttpServlet{
5. public void doGet(HttpServletRequest req,HttpServletResponse res)
6. throws ServletException,IOException
7. {
8. res.setContentType("text/html");//setting the content type
9. PrintWriter pw=res.getWriter();//get the stream to write the data
10.
11. //writing html in the stream
12. pw.println("<html><body>");
13. pw.println("Welcome to servlet");
14. pw.println("</body></html>");
15.
16. pw.close();//closing the stream
17. }}
```

3) Compile the servlet

For compiling the Servlet, jar file is required to be loaded. Different Servers provide different jar files:

Jar file	Server
1) servlet-api.jar	Apache Tomcat
2) weblogic.jar	Weblogic
3) javaee.jar	Glassfish
4) javaee.jar	JBoss

Two ways to load the jar file

- set classpath
- paste the jar file in JRE/lib/ext folder

Put the java file in any folder. After compiling the java file, paste the class file of servlet in WEB-INF/classes directory.

4) Create the deployment descriptor (web.xml file)

The deployment descriptor is an xml file, from which Web Container gets the information about the servet to be invoked.

The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull.

There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

web.xml file

```
<web-app>
<servlet>
<servlet-name>sonoojaiswal</servlet-name>
<servlet-class>DemoServlet</servlet-class>
```

```
</servlet>

<servlet-mapping>
<servlet-name>sonoojaiswal</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>

</web-app>
```

Description of the elements of web.xml file

There are too many elements in the web.xml file. Here is the illustration of some elements that is used in the above web.xml file. The elements are as follows:

<web-app> represents the whole application.
<servlet> is sub element of <web-app> and represents the servlet.
<servlet-name> is sub element of <servlet> represents the name of the servlet.
<servlet-class> is sub element of <servlet> represents the class of the servlet.
<servlet-mapping> is sub element of <web-app>. It is used to map the servlet.
<url-pattern> is sub element of <servlet-mapping>. This pattern is used at client side to invoke the servlet.

5) Start the Server and deploy the project

To start Apache Tomcat server, double click on the startup.bat file under apache-tomcat/bin directory.

One Time Configuration for Apache Tomcat Server

You need to perform 2 tasks:

- set JAVA_HOME or JRE_HOME in environment variable (It is required to start server).
- Change the port number of tomcat (optional). It is required if another server is running on same port (8080).

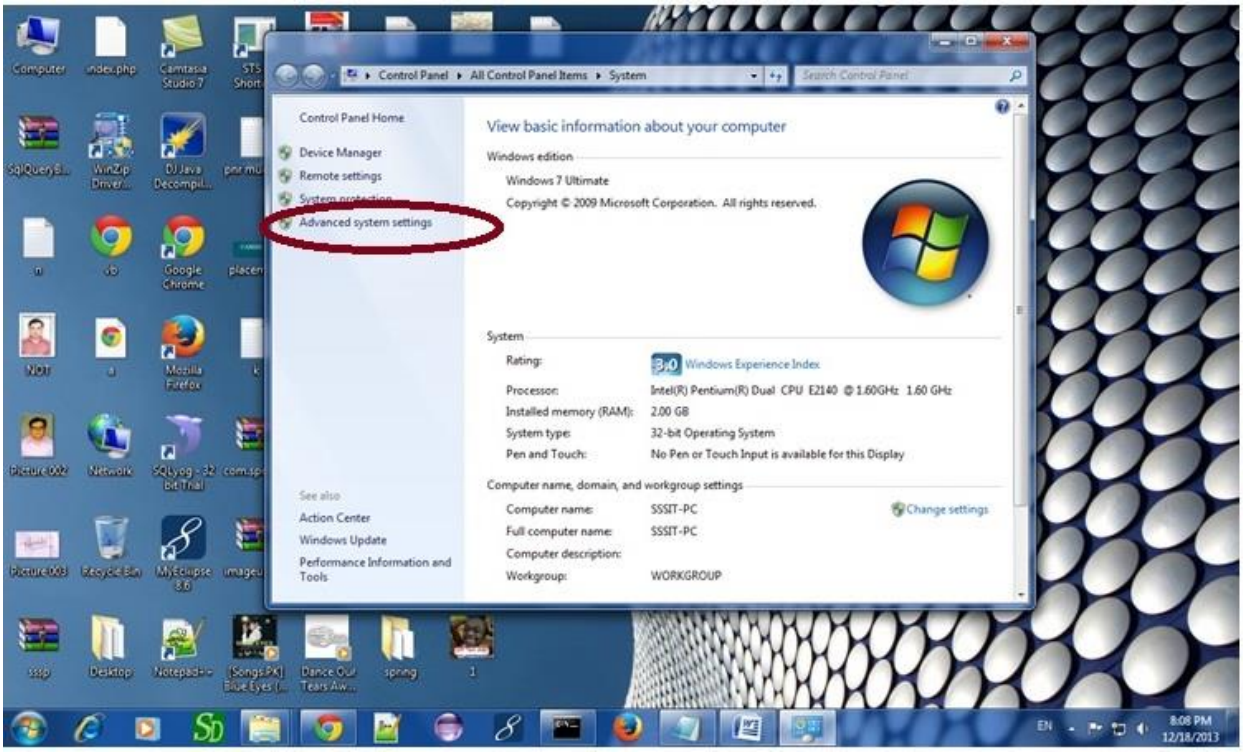
1) How to set JAVA_HOME in environment variable?

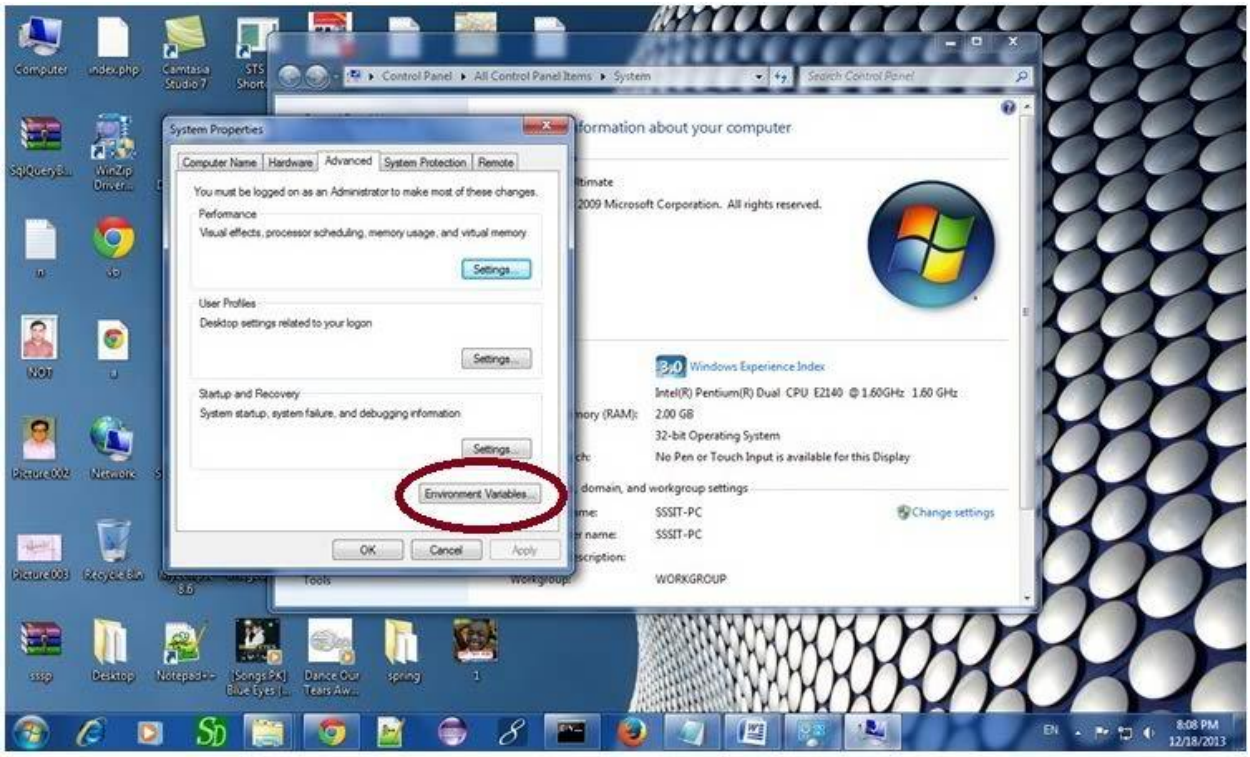
To start Apache Tomcat server JAVA_HOME and JRE_HOME must be set in Environment variables.

Go to My Computer properties -> Click on advanced tab then environment variables -> Click on the new tab of user variable -> Write JAVA_HOME in variable name and paste the path of jdk folder in variable value -> ok -> ok -> ok.

Go to My Computer properties:

Click on advanced system settings tab then environment variables:





Click on the new tab of user variable or system variable:

Write JAVA_HOME in variable name and paste the path of jdk folder in variable value:

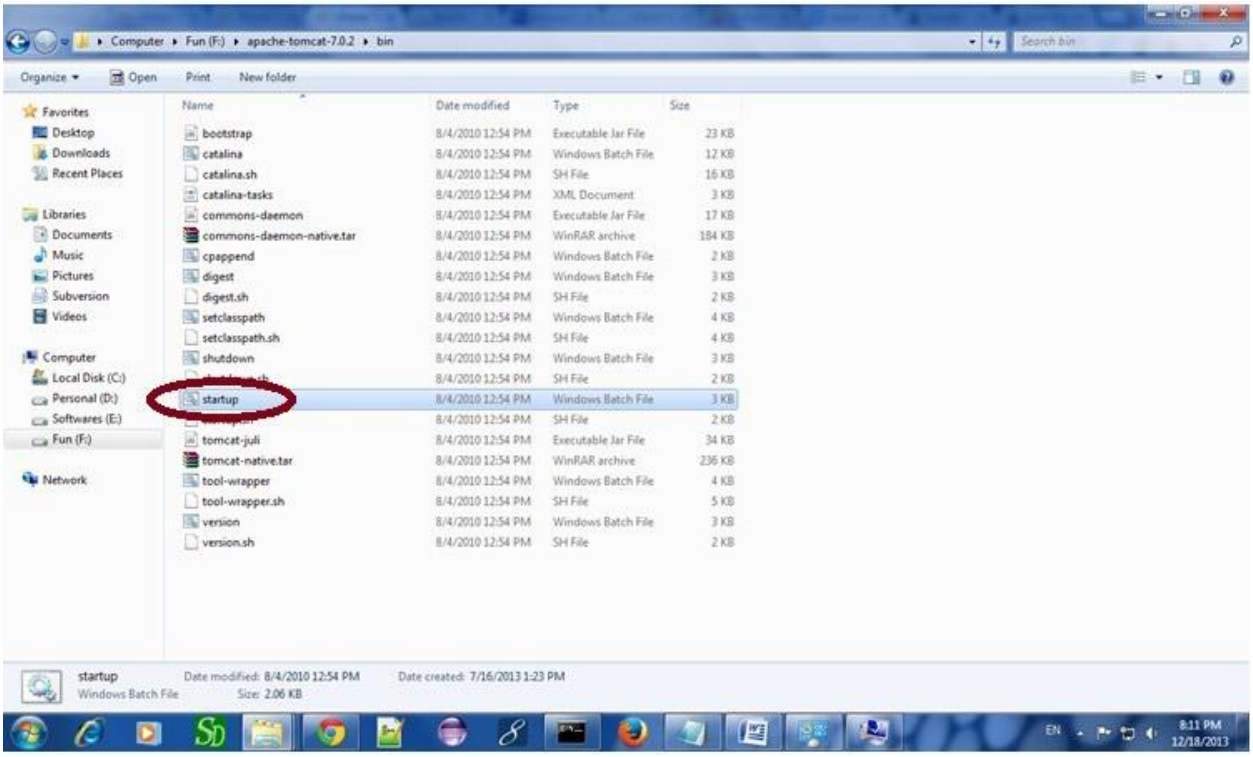
There must not be semicolon (;) at the end of the path.

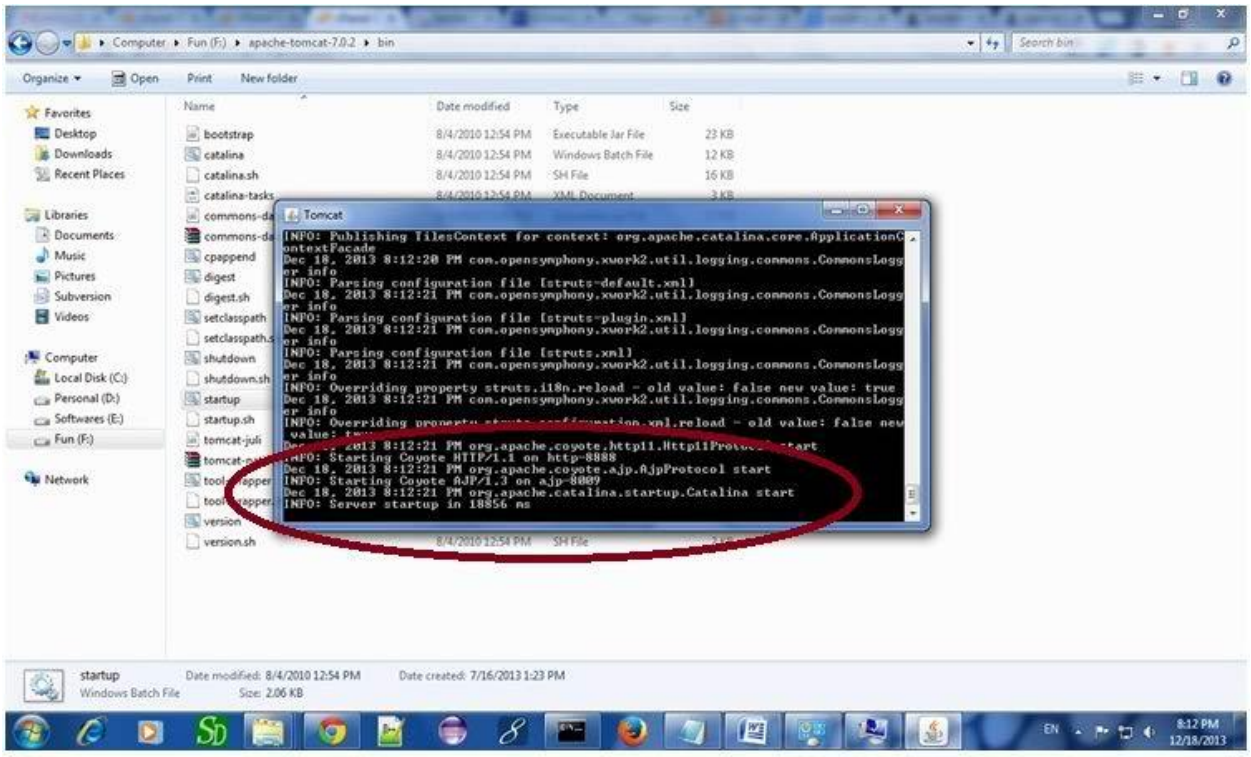
After setting the JAVA_HOME double click on the startup.bat file in apache tomcat/bin.

Note: There are two types of tomcat available:

- Apache tomcat that needs to extract only (no need to install)
- Apache tomcat that needs to install

It is the example of apache tomcat that needs to extract only.





Now server is started successfully.

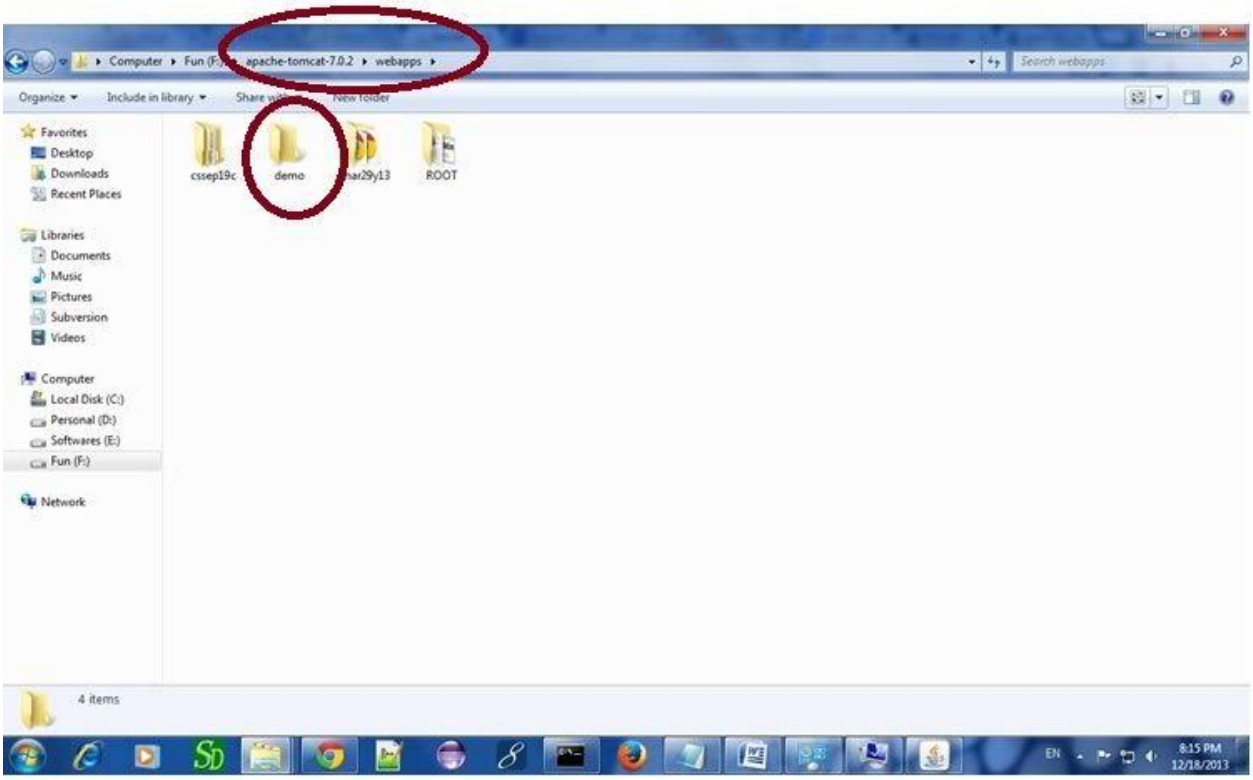
2) How to change port number of apache tomcat

Changing the port number is required if there is another server running on the same system with same port number.Suppose you have installed oracle, you need to change the port number of apache tomcat because both have the default port number 8080.

Open server.xml file in notepad. It is located inside the apache-tomcat/conf directory . Change the Connector port = 8080 and replace 8080 by any four digit number instead of 8080. Let us replace it by 9999 and save this file.

5) How to deploy the servlet project

Copy the project and paste it in the webapps folder under apache tomcat.



But there are several ways to deploy the project. They are as follows:

- By copying the context(project) folder into the webapps directory
- By copying the war folder into the webapps directory
- By selecting the folder path from the server
- By selecting the war file from the server

Here, we are using the first approach.

You can also create war file, and paste it inside the webapps directory. To do so, you need to use jar tool to create the war file. Go inside the project directory (before the WEB-INF), then write:

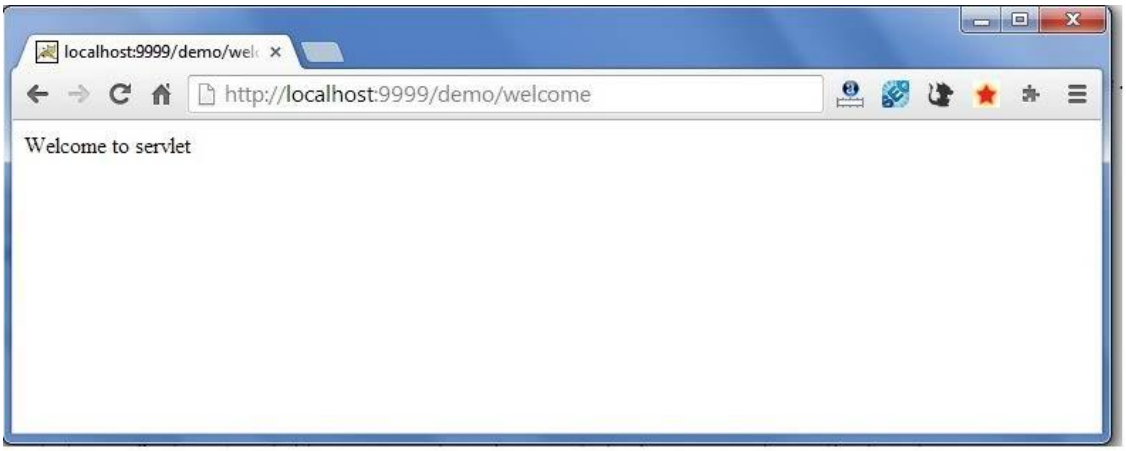
```
projectfolder> jar cvf myproject.war *
```

Creating war file has an advantage that moving the project from one location to another takes less time.

6) How to access the servlet

Open browser and write http://hostname:portno/contextroot/urlpatternofservlet. For example:

http://localhost:9999/demo/welcome



RequestDispatcher in Servlet:

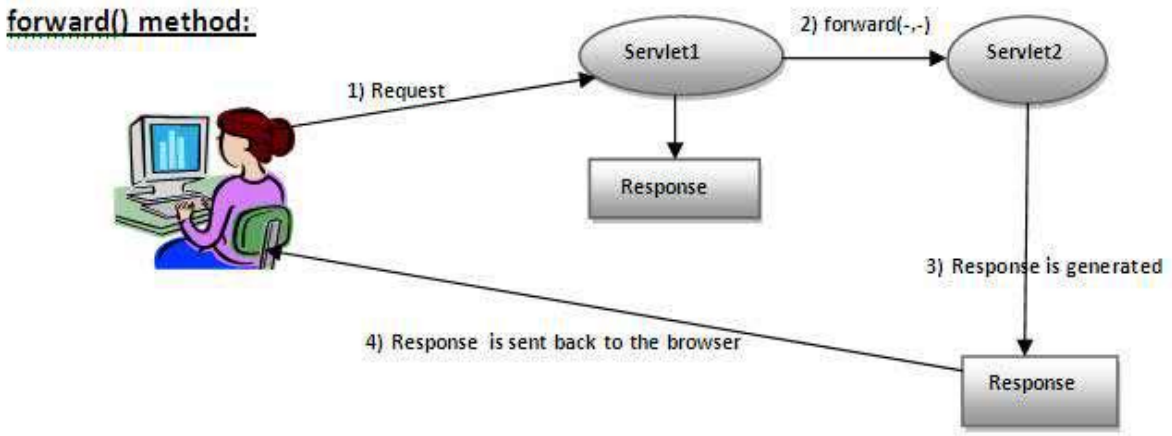
The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration.

There are two methods defined in the RequestDispatcher interface.

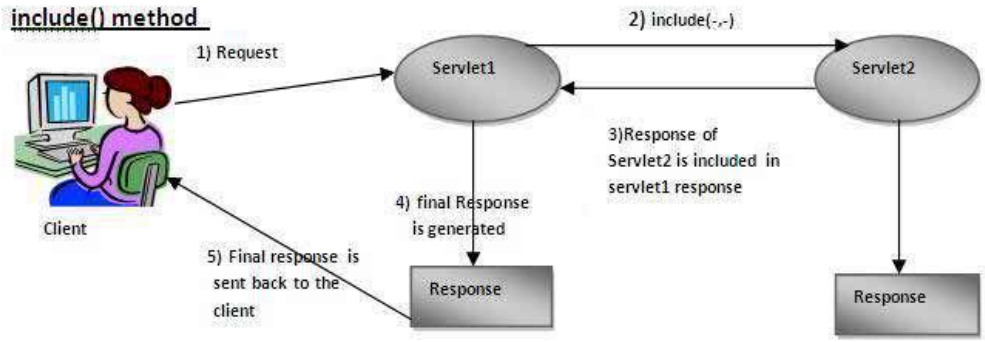
Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

- public void forward(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
- public void include(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:Includes the content of a resource (servlet, JSP page, or HTML file) in the response.



As in the above figure, response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.



As you can see in the above figure, response of second servlet is included in the response of the first servlet that is being sent to the client.

How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher. Syntax:

Syntax of getRequestDispatcher method

```
public RequestDispatcher getRequestDispatcher(String resource);
```

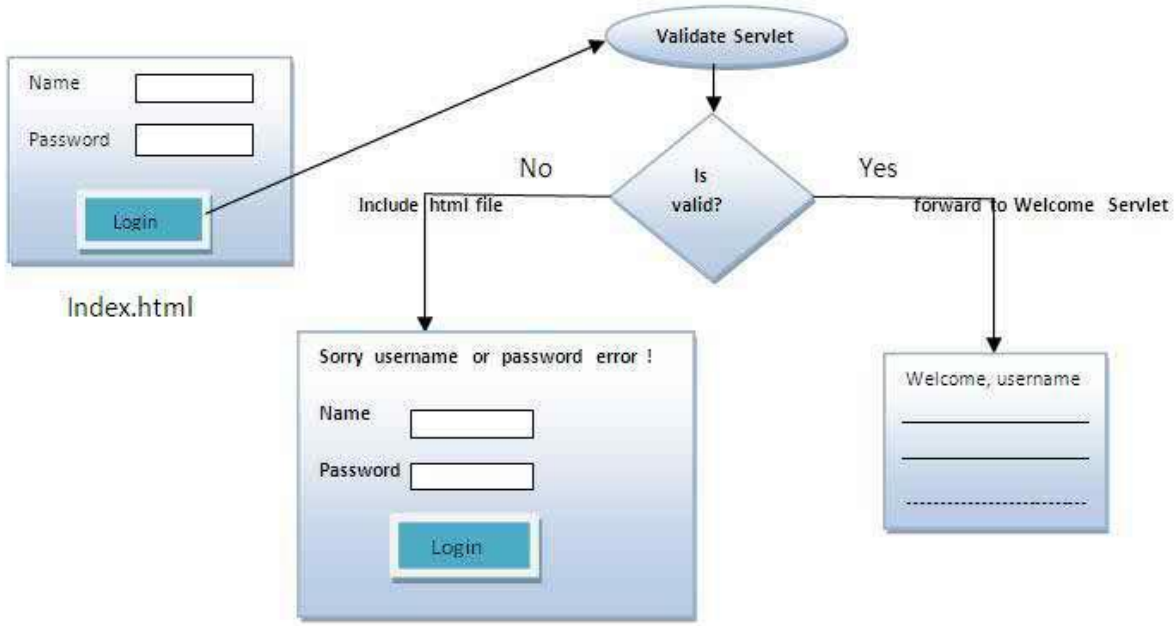
Example of using getRequestDispatcher method

```
RequestDispatcher rd=request.getRequestDispatcher("servlet2");
//servlet2 is the url-pattern of the second servlet

rd.forward(request, response);//method may be include or forward
```

Example of RequestDispatcher interface

In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: sorry username or password error!. In this program, we are cheking for hardcoded information. But you can check it to the database also that we will see in the development chapter. In this example, we have created following files:

- index.html file: for getting input from the user.
- Login.java file: a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
- WelcomeServlet.java file: a servlet class for displaying the welcome message.
- web.xml file: a deployment descriptor file that contains the information about the servlet.



index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
Password:<input type="password" name="userPass"/><br/>
<input type="submit" value="login"/>
</form>
```

Login.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class Login extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        String p=request.getParameter("userPass");

        if(p.equals("servlet")){
            RequestDispatcher rd=request.getRequestDispatcher("servlet2");
            rd.forward(request, response);
        }
        else{
            out.print("Sorry UserName or Password Error!");
            RequestDispatcher rd=request.getRequestDispatcher("/index.html");
            rd.include(request, response);

            }
        }

    }
```

WelcomeServlet.java

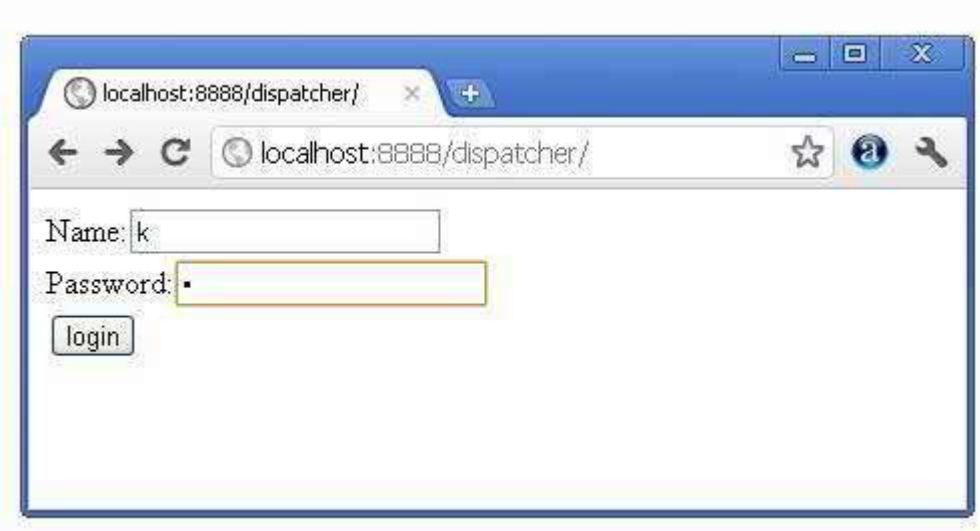
```
    import java.io.*;
    import javax.servlet.*;
    import javax.servlet.http.*;

    public class WelcomeServlet extends HttpServlet {

        public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

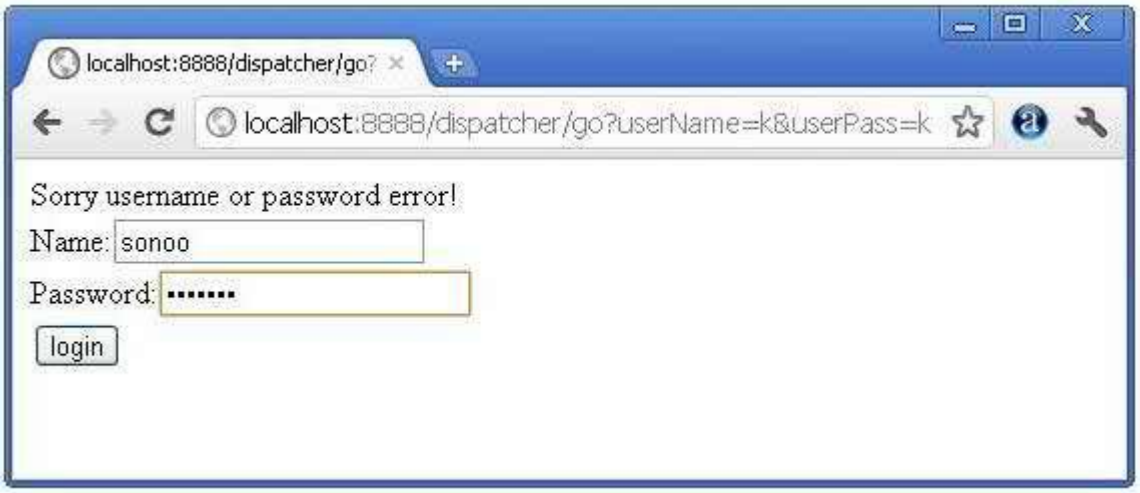
```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);
        }

    }
```

web.xml

```
    <web-app>
     <servlet>
       <servlet-name>Login</servlet-name>
       <servlet-class>Login</servlet-class>
     </servlet>
     <servlet>
       <servlet-name>WelcomeServlet</servlet-name>
       <servlet-class>WelcomeServlet</servlet-class>
     </servlet>


     <servlet-mapping>
       <servlet-name>Login</servlet-name>
       <url-pattern>/servlet1</url-pattern>
     </servlet-mapping>
     <servlet-mapping>
       <servlet-name>WelcomeServlet</servlet-name>
       <url-pattern>/servlet2</url-pattern>
     </servlet-mapping>

     <welcome-file-list>
```

```
    <welcome-file>index.html</welcome-file>
   </welcome-file-list>
  </web-app>
```





ServletContext Interface

An object of ServletContext is created by the web container at time of deploying the project. This object can be used to get configuration information from web.xml file. There is only one ServletContext object per web application.

If any information is shared to many servlet, it is better to provide it from the web.xml file using the <context-param> element.

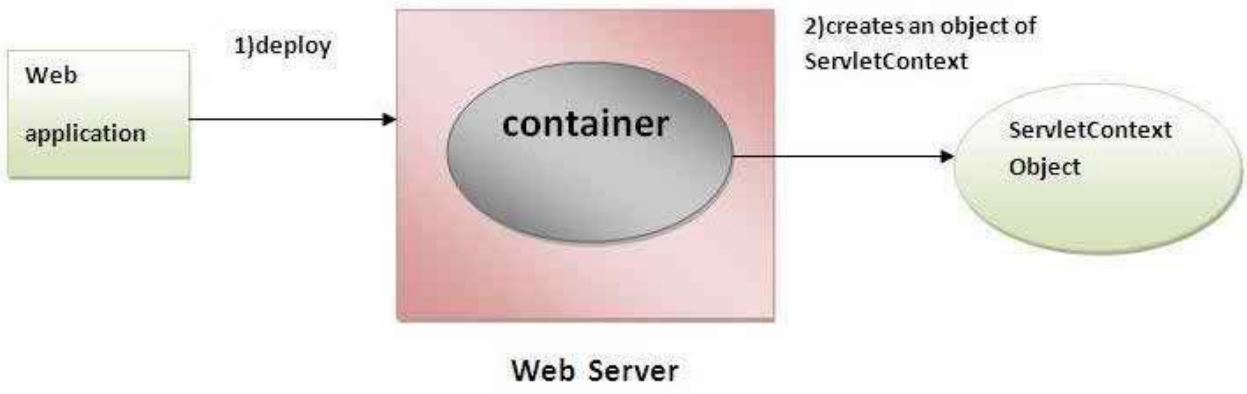
Advantage of ServletContext

Easy to maintain if any information is shared to all the servlet, it is better to make it available for all the servlet. We provide this information from the web.xml file, so if the information is changed, we don't need to modify the servlet. Thus it removes maintenance problem.

Usage of ServletContext Interface

There can be a lot of usage of ServletContext object. Some of them are as follows:

- The object of ServletContext provides an interface between the container and servlet.
- The ServletContext object can be used to get configuration information from the web.xml file.
- The ServletContext object can be used to set, get or remove attribute from the web.xml file.
- The ServletContext object can be used to provide inter-application communication.



Commonly used methods of ServletContext interface

There is given some commonly used methods of ServletContext interface.

- public String getInitParameter(String name):Returns the parameter value for the specified parameter name.
- public Enumeration getInitParameterNames():Returns the names of the context's initialization parameters.

- public void setAttribute(String name,Object object):sets the given object in the application scope.
- public Object getAttribute(String name):Returns the attribute for the specified name.
- public Enumeration getInitParameterNames():Returns the names of the context's initialization parameters as an Enumeration of String objects.
- public void removeAttribute(String name):Removes the attribute with the given name from the servlet context.

How to get the object of ServletContext interface

- getServletContext() method of ServletConfig interface returns the object of ServletContext.
- getServletContext() method of GenericServlet class returns the object of ServletContext.

Syntax of getServletContext() method

```
public ServletContext getServletContext()
```

Example of getServletContext() method

```
//We can get the ServletContext object from ServletConfig object
ServletContext application=getServletConfig().getServletContext();

//Another convenient way to get the ServletContext object
ServletContext application=getServletContext();
```

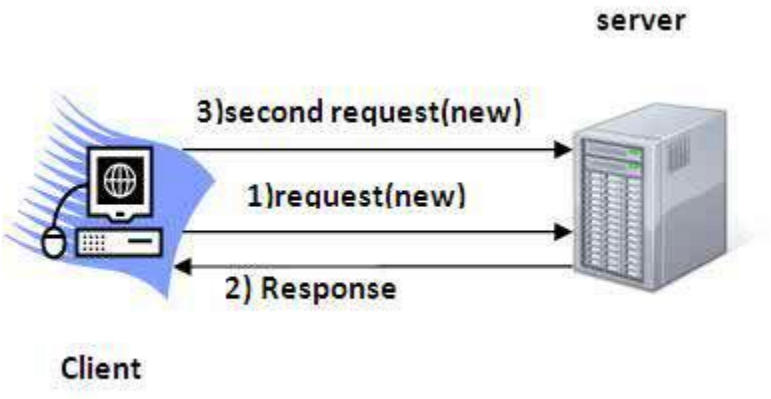
Session Tracking in Servlets

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of a user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



Session Tracking Techniques

There are four techniques used in Session tracking:

- Cookies
- Hidden Form Field
- URL Rewriting
- HttpSession

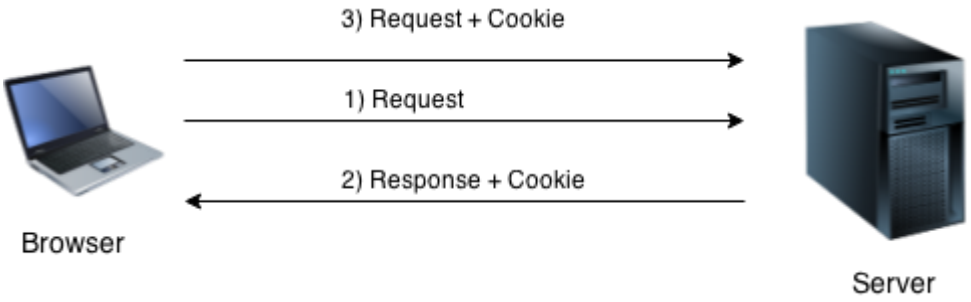
Cookies in Servlet

A cookie is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



Types of Cookie

There are 2 types of cookies in servlets.

- Non-persistent cookie
- Persistent cookie

Non-persistent cookie

It is valid for single session only. It is removed each time when user closes the browser.

Persistent cookie

It is valid for multiple session . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

- Simplest technique of maintaining the state.
- Cookies are maintained at client side.

Disadvantage of Cookies

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in Cookie object.

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

Useful Methods of Cookie class

There are given some commonly used methods of the Cookie class.

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

- public void addCookie(Cookie ck):method of HttpServletResponse interface is used to add cookie in response object.
- public Cookie[] getCookies():method of HttpServletRequest interface is used to return all the cookies from the browser.

How to create Cookie?

Let's see the simple code to create cookie.

```
Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object
response.addCookie(ck);//adding cookie in the response
```

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

```
Cookie ck=new Cookie("user","");//deleting value of cookie
ck.setMaxAge(0);//changing the maximum age to 0 seconds
response.addCookie(ck);//adding cookie in the response
```

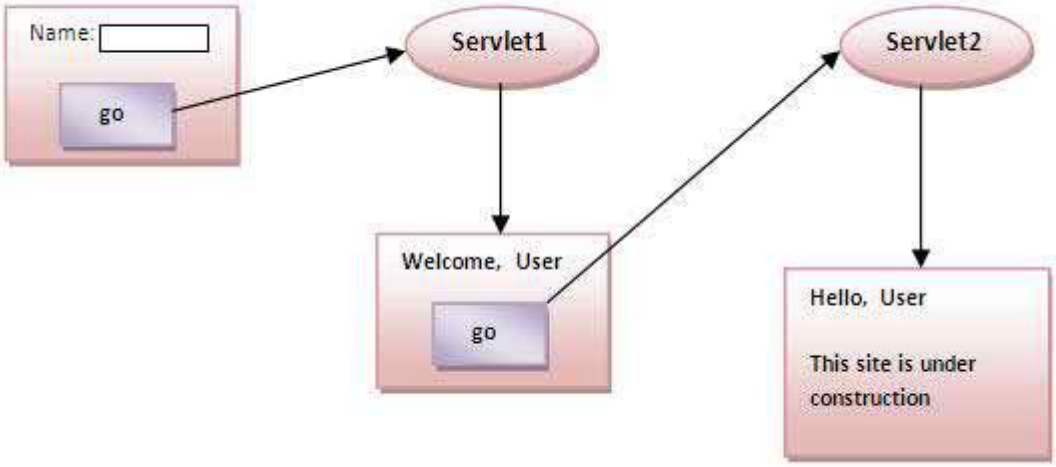
How to get Cookies?

Let's see the simple code to get all the cookies.

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++){
 out.print("<br>"+ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie
}
```

Simple example of Servlet Cookies

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.



In case of Hidden Form Field a hidden (invisible) textfield is used for maintaining the state of an user.

In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.

Let's see the code to store value in hidden field.

```
<input type="hidden" name="uname" value="Vimal Jaiswal">
```

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.
Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

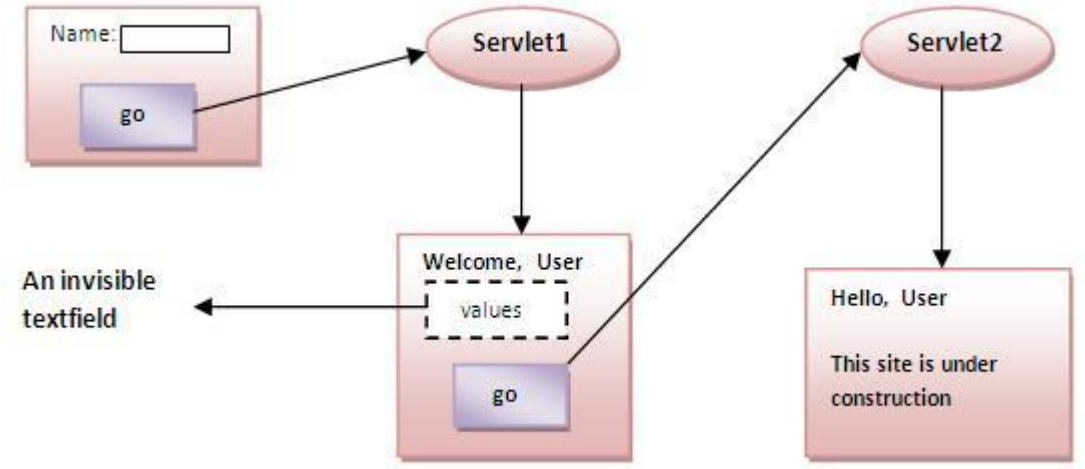
- It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

- It is maintained at server side.
- Extra form submission is required on each pages.
- Only textual information can be used.

Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.

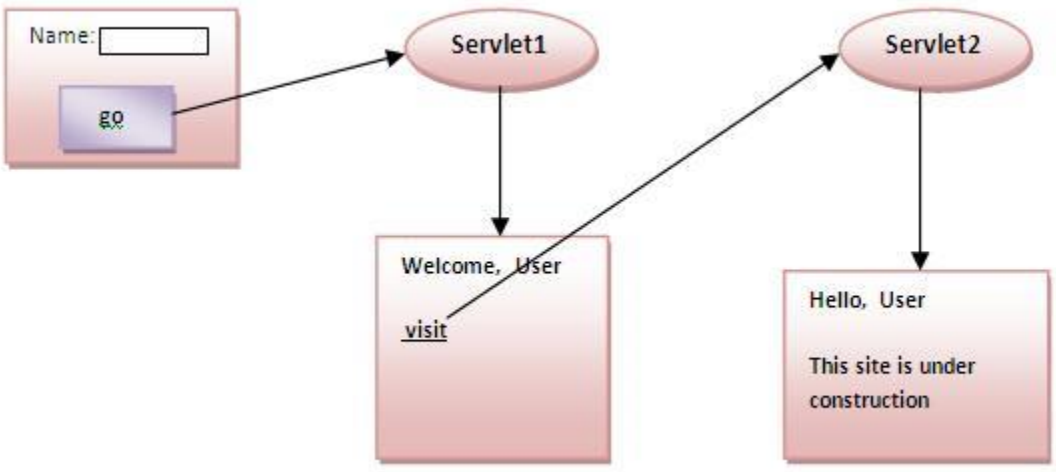


URL Rewriting

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.



Advantage of URL Rewriting

- It will always work whether cookie is disabled or not (browser independent).
- Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

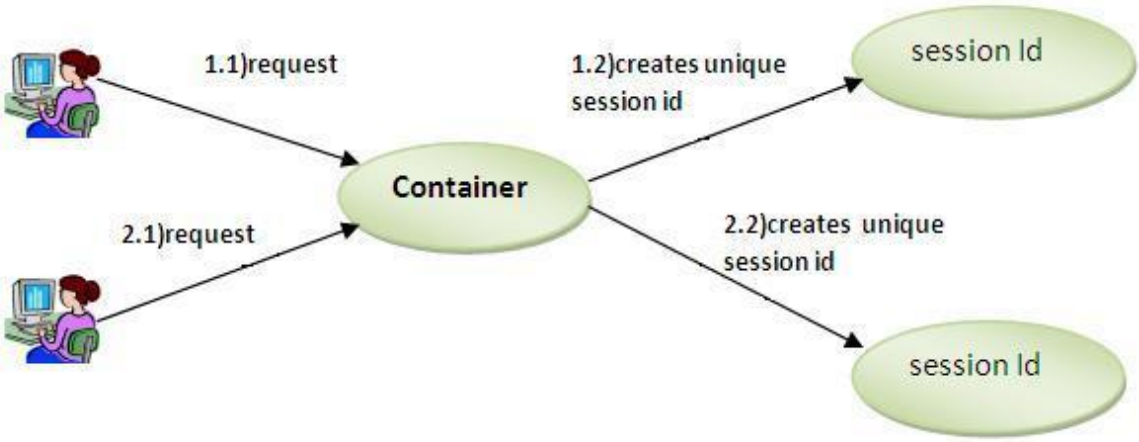
- It will work only with links.
- It can send Only textual information.

HttpSession interface

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

- bind objects

- view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



How to get the HttpSession object ?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

- public HttpSession getSession():Returns the current session associated with this request, or if the request does not have a session, creates one.
- public HttpSession getSession(boolean create):Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

- public String getId():Returns a string containing the unique identifier value.
- public long getCreationTime():Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- public long getLastAccessedTime():Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- public void invalidate():Invalidates this session then unbinds any objects bound to it.

SendRedirect in servlet

The sendRedirect() method of HttpServletResponse interface can be used to redirect response to another resource, it may be servlet, jsp or html file.

It accepts relative as well as absolute URL.

It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

Difference between forward() and sendRedirect() method

There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:

forward() method	sendRedirect() method
The forward() method works at server side.	The sendRedirect() method works at client side.
It sends the same request and response objects to another servlet.	It always sends a new request.
It can work within the server only.	It can be used within and outside the server.
Example: request.getRequestDispacher("servlet2").forward(request,response);	Example: response.sendRedirect("servlet2")

Syntax of sendRedirect() method

```
public void sendRedirect(String URL)throws IOException;
```

Full example of sendRedirect method in servlet

In this example, we are redirecting the request to the google server. Notice that sendRedirect method works at client side that is why we can our request to anywhere. We can send our request within and outside the server.

DemoServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();

response.sendRedirect("http://www.google.com");

pw.close();
}}
```

Creating custom google search using sendRedirect

In this example, we are using sendRedirect method to send request to google server with the request data.

index.html

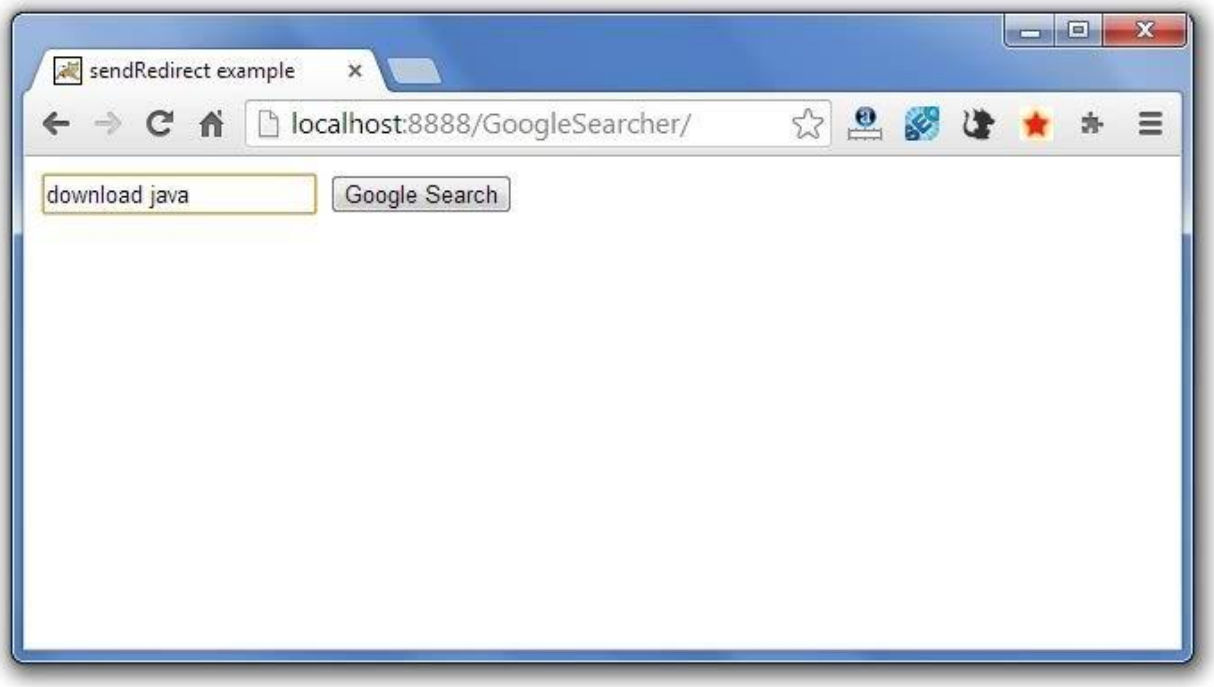
```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>sendRedirect example</title>
</head>
<body>
```

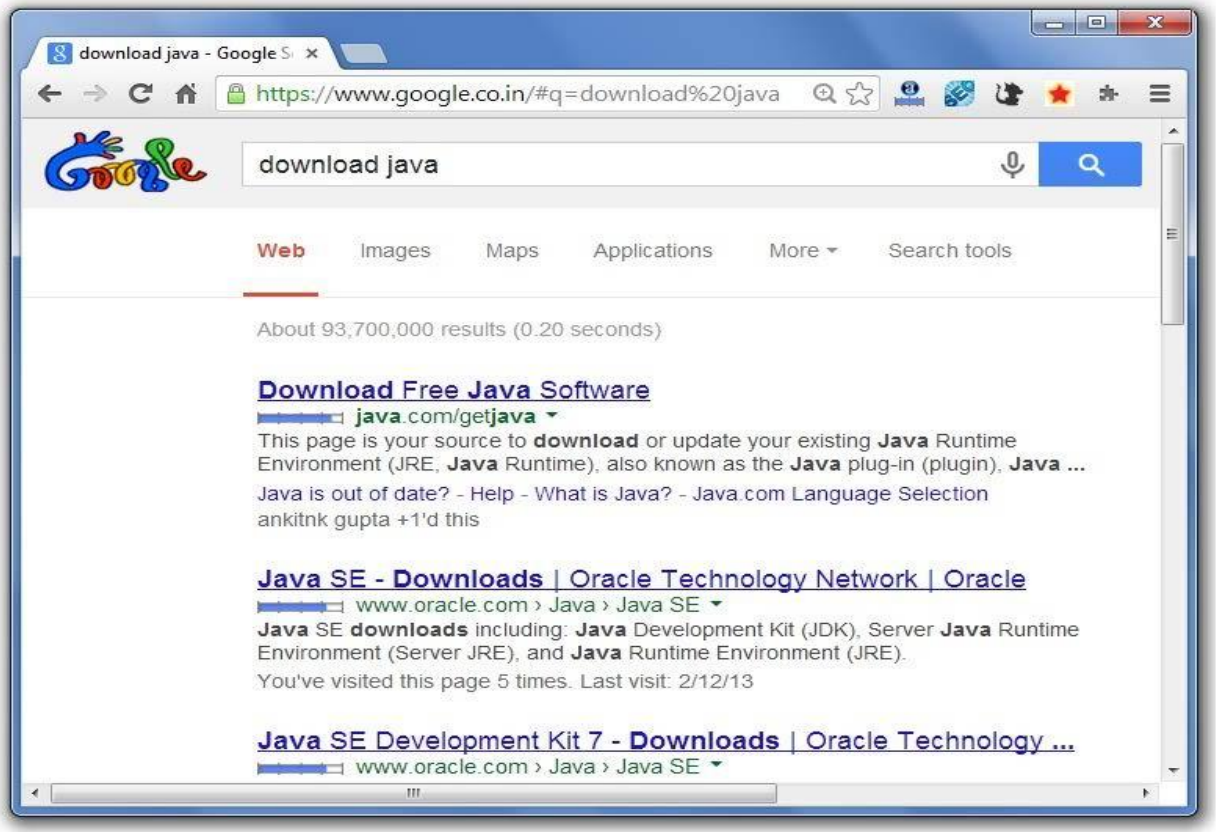
```
<form action="MySearcher">
<input type="text" name="name">
<input type="submit" value="Google Search">
</form>

</body>
</html>
```

MySearcher.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MySearcher extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      String name=request.getParameter("name");
      response.sendRedirect("https://www.google.co.in/#q="+name);
   }
}
```

Output


sendRedirect example
localhost:8888/GoogleSearcher/
download java
Google Search


download java - Google S
https://www.google.co.in/#q=download%20java
download java
Web
Images
Maps
Applications
More
Search tools
About 93,700,000 results (0.20 seconds)
Download Free Java Software
java.com/getjava
This page is your source to download or update your existing Java Runtime Environment (JRE, Java Runtime), also known as the Java plug-in (plugin), Java ...
Java is out of date? - Help - What is Java? - Java.com Language Selection
ankitnk gupta +1'd this
Java SE - Downloads | Oracle Technology Network | Oracle
www.oracle.com › Java › Java SE
Java SE downloads including: Java Development Kit (JDK), Server Java Runtime Environment (Server JRE), and Java Runtime Environment (JRE).
You've visited this page 5 times. Last visit: 2/12/13
Java SE Development Kit 7 - Downloads | Oracle Technology ...
www.oracle.com › Java › Java SE

Servlet Login and Logout Example using Cookies

A cookie is a kind of information that is stored at client side.

Here, we are going to create a login and logout example using servlet cookies.

In this example, we are creating 3 links: login, logout and profile. User can't go to profile page until he/she is logged in. If user is logged out, he need to login again to visit profile.

In this application, we have created following files.

index.html
link.html
login.html
LoginServlet.java
LogoutServlet.java
ProfileServlet.java
web.xml

File: index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Servlet Login Example</title>
</head>
<body>

<h1>Welcome to Login App by Cookie</h1>
<a href="login.html">Login</a>|
<a href="LogoutServlet">Logout</a>|
<a href="ProfileServlet">Profile</a>
```

```
</body>
</html>
```

File: link.html

```
<a href="login.html">Login</a> |
<a href="LogoutServlet">Logout</a> |
<a href="ProfileServlet">Profile</a>
<hr>
```

File: login.html

```
<form action="LoginServlet" method="post">
Name:<input type="text" name="name"><br>
Password:<input type="password" name="password"><br>
<input type="submit" value="login">
</form>
```

File: LoginServlet.java

```
package com.javatpoint;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class LoginServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
      response.setContentType("text/html");
```

```
        PrintWriter out=response.getWriter();

        request.getRequestDispatcher("link.html").include(request, response);

        String name=request.getParameter("name");
        String password=request.getParameter("password");

        if(password.equals("admin123")){
           out.print("You are successfully logged in!");
           out.print("<br>Welcome, "+name);

           Cookie ck=new Cookie("name",name);
           response.addCookie(ck);
        }else{
           out.print("sorry, username or password error!");
           request.getRequestDispatcher("login.html").include(request, response);
        }

        out.close();
    }

}
```

File: LogoutServlet.java

```
package com.javatpoint;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class LogoutServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out=response.getWriter();



      request.getRequestDispatcher("link.html").include(request, response);

      Cookie ck=new Cookie("name","");
      ck.setMaxAge(0);
      response.addCookie(ck);

      out.print("you are successfully logged out!");
   }
}
```

File: ProfileServlet.java

```
package com.javatpoint;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ProfileServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
                    throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out=response.getWriter();

        request.getRequestDispatcher("link.html").include(request, response);

        Cookie ck[]=request.getCookies();
        if(ck!=null){
         String name=ck[0].getValue();
        if(!name.equals("")||name!=null){
           out.print("<b>Welcome to Profile</b>");
           out.print("<br>Welcome, "+name);
        }
        }else{
           out.print("Please login first");
           request.getRequestDispatcher("login.html").include(request, response);
        }
        out.close();
      }
   }
```

File: web.xml

```
   <?xml version="1.0" encoding="UTF-8"?>
   <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">

     <servlet>
       <description></description>
       <display-name>LoginServlet</display-name>
```

```
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>com.javatpoint.LoginServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/LoginServlet</url-pattern>
  </servlet-mapping>
  <servlet>
    <description></description>
    <display-name>ProfileServlet</display-name>
    <servlet-name>ProfileServlet</servlet-name>
    <servlet-class>com.javatpoint.ProfileServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ProfileServlet</servlet-name>
    <url-pattern>/ProfileServlet</url-pattern>
  </servlet-mapping>
  <servlet>
    <description></description>
    <display-name>LogoutServlet</display-name>
    <servlet-name>LogoutServlet</servlet-name>
    <servlet-class>com.javatpoint.LogoutServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LogoutServlet</servlet-name>
    <url-pattern>/LogoutServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Hidden Form Field

In case of Hidden Form Field a hidden (invisible) textfield is used for maintaining the state of an user.

In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.

Let's see the code to store value in hidden field.

```
<input type="hidden" name="uname" value="Vimal Jaiswal">
```

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

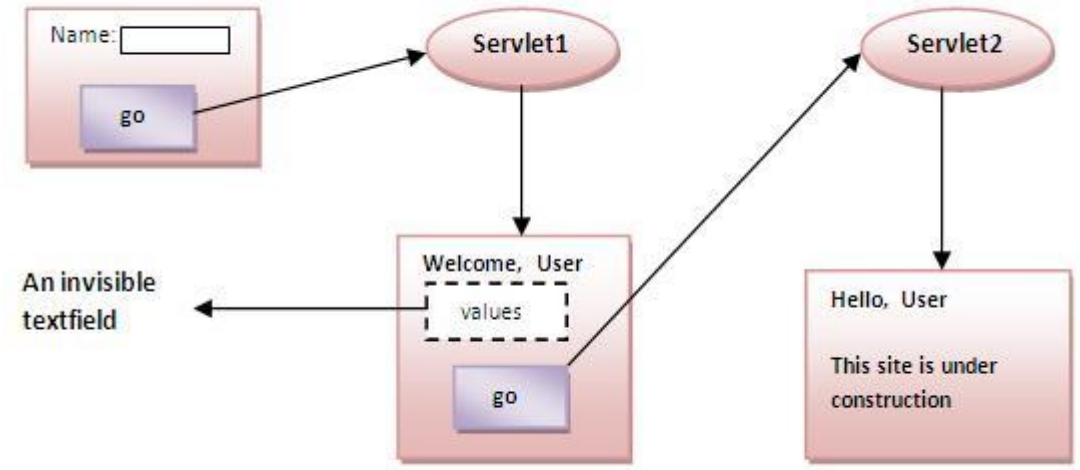
- It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

- It is maintained at server side.
- Extra form submission is required on each pages.
- Only textual information can be used.

Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
```

```
        out.print("Welcome "+n);

        //creating form that have invisible textfield
        out.print("<form action='servlet2'>");
        out.print("<input type='hidden' name='uname' value='"+n+"'>");
        out.print("<input type='submit' value='go'>");
        out.print("</form>");
        out.close();

            }catch(Exception e){System.out.println(e);}
    }

}
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SecondServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //Getting the value from the hidden field
        String n=request.getParameter("uname");
        out.print("Hello "+n);

        out.close();
            }catch(Exception e){System.out.println(e);}
    }
```

```
}
```

web.xml

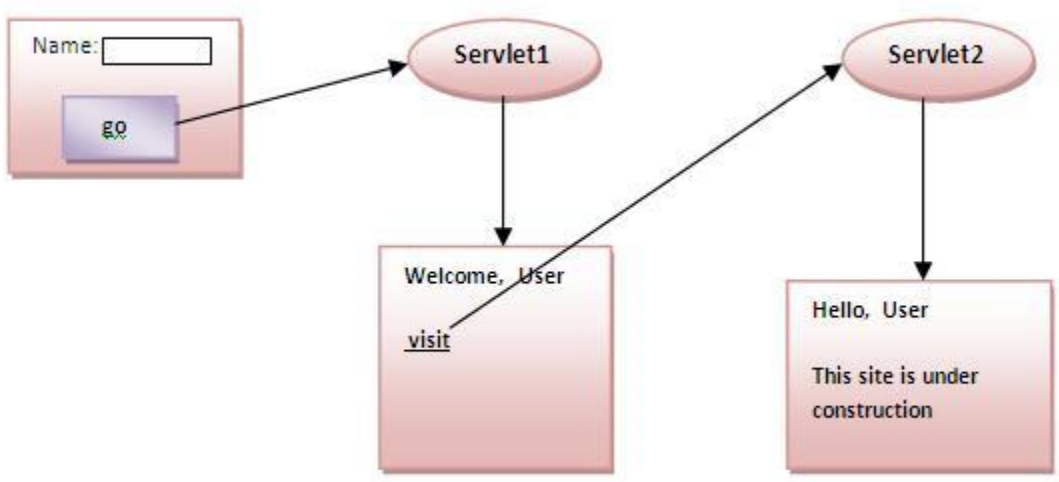
```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

URL Rewriting

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.



Advantage of URL Rewriting

- It will always work whether cookie is disabled or not (browser independent).
- Extra form submission is not required on each pages.

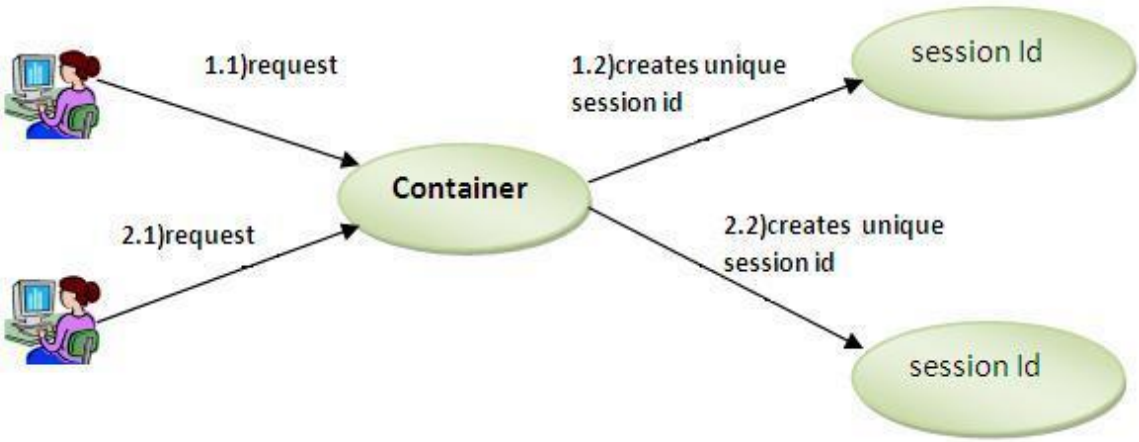
Disadvantage of URL Rewriting

- It will work only with links.
- It can send Only textual information.

HttpSession interface

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

- bind objects
- view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



How to get the HttpSession object ?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

- public HttpSession getSession():Returns the current session associated with this request, or if the request does not have a session, creates one.
- public HttpSession getSession(boolean create):Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

- public String getId():Returns a string containing the unique identifier value.
- public long getCreationTime():Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

- public long getLastAccessedTime():Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- public void invalidate():Invalidates this session then unbinds any objects bound to it.

Servlet HttpSession Login and Logout Example

We can bind the objects on HttpSession instance and get the objects by using setAttribute and getAttribute methods.

Here, we are going to create a real world login and logout application without using database code. We are assuming that password is admin123.

Visit here for login and logout application using cookies only servlet login and logout example using cookies

In this example, we are creating 3 links: login, logout and profile. User can't go to profile page until he/she is logged in. If user is logged out, he need to login again to visit profile.

In this application, we have created following files.

index.html
link.html
login.html
LoginServlet.java
LogoutServlet.java
ProfileServlet.java
web.xml

File: index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Servlet Login Example</title>
</head>
<body>

<h1>Login App using HttpSession</h1>
<a href="login.html">Login</a>|
<a href="LogoutServlet">Logout</a>|
<a href="ProfileServlet">Profile</a>

</body>
</html>
```

File: link.html

```
<a href="login.html">Login</a> |
<a href="LogoutServlet">Logout</a> |
<a href="ProfileServlet">Profile</a>
<hr>
```

File: login.html

```
<form action="LoginServlet" method="post">
Name:<input type="text" name="name"><br>
Password:<input type="password" name="password"><br>
<input type="submit" value="login">
</form>
```

File: LoginServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class LoginServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out=response.getWriter();
      request.getRequestDispatcher("link.html").include(request, response);

      String name=request.getParameter("name");
      String password=request.getParameter("password");

      if(password.equals("admin123")){
      out.print("Welcome, "+name);
      HttpSession session=request.getSession();
      session.setAttribute("name",name);
      }
      else{
         out.print("Sorry, username or password error!");
         request.getRequestDispatcher("login.html").include(request, response);
      }
      out.close();
   }
}
```

File: LogoutServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class LogoutServlet extends HttpServlet {
      protected void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
         response.setContentType("text/html");
         PrintWriter out=response.getWriter();

         request.getRequestDispatcher("link.html").include(request, response);

         HttpSession session=request.getSession();
         session.invalidate();

         out.print("You are successfully logged out!");

         out.close();
    }
}
```

File: ProfileServlet.java

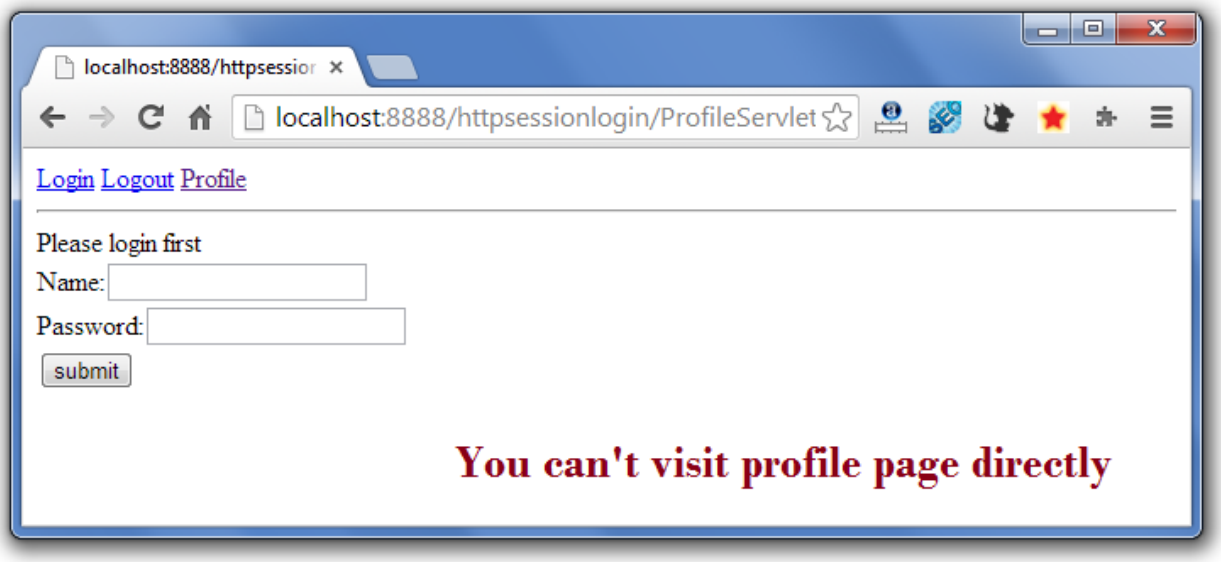
```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class ProfileServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out=response.getWriter();
      request.getRequestDispatcher("link.html").include(request, response);

      HttpSession session=request.getSession(false);
      if(session!=null){
      String name=(String)session.getAttribute("name");

      out.print("Hello, "+name+" Welcome to Profile");
      }
      else{
         out.print("Please login first");
         request.getRequestDispatcher("login.html").include(request, response);
      }
      out.close();
   }
}
```

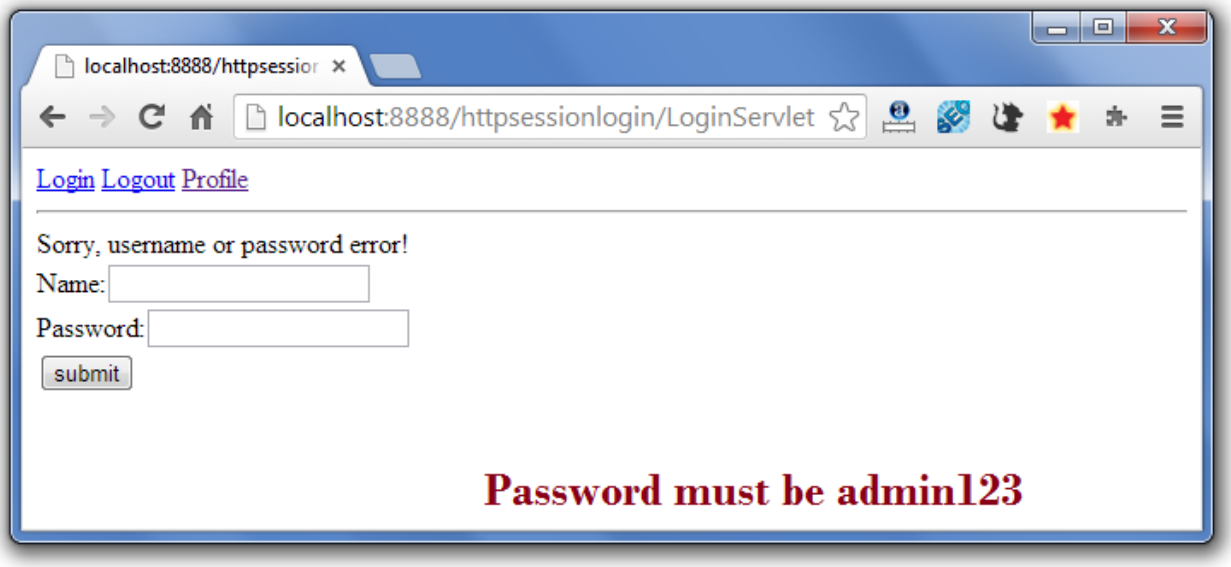
File: web.xml

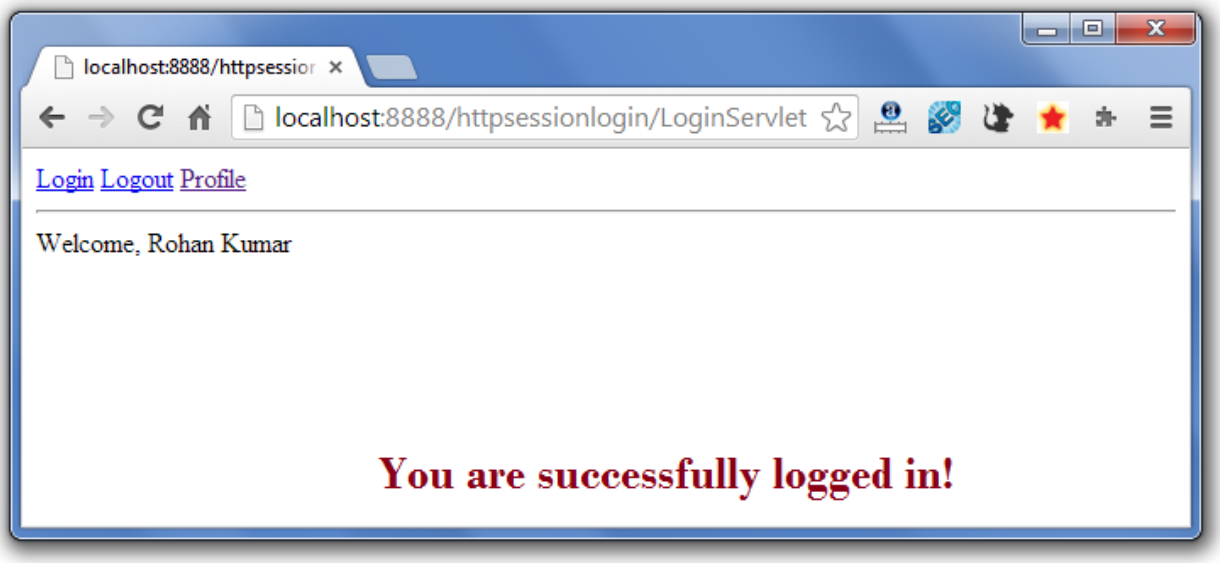
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">

  <servlet>
    <description></description>
    <display-name>LoginServlet</display-name>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>LoginServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/LoginServlet</url-pattern>
  </servlet-mapping>
  <servlet>
    <description></description>
    <display-name>ProfileServlet</display-name>
    <servlet-name>ProfileServlet</servlet-name>
    <servlet-class>ProfileServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ProfileServlet</servlet-name>
    <url-pattern>/ProfileServlet</url-pattern>
  </servlet-mapping>
  <servlet>
    <description></description>
    <display-name>LogoutServlet</display-name>
    <servlet-name>LogoutServlet</servlet-name>
```

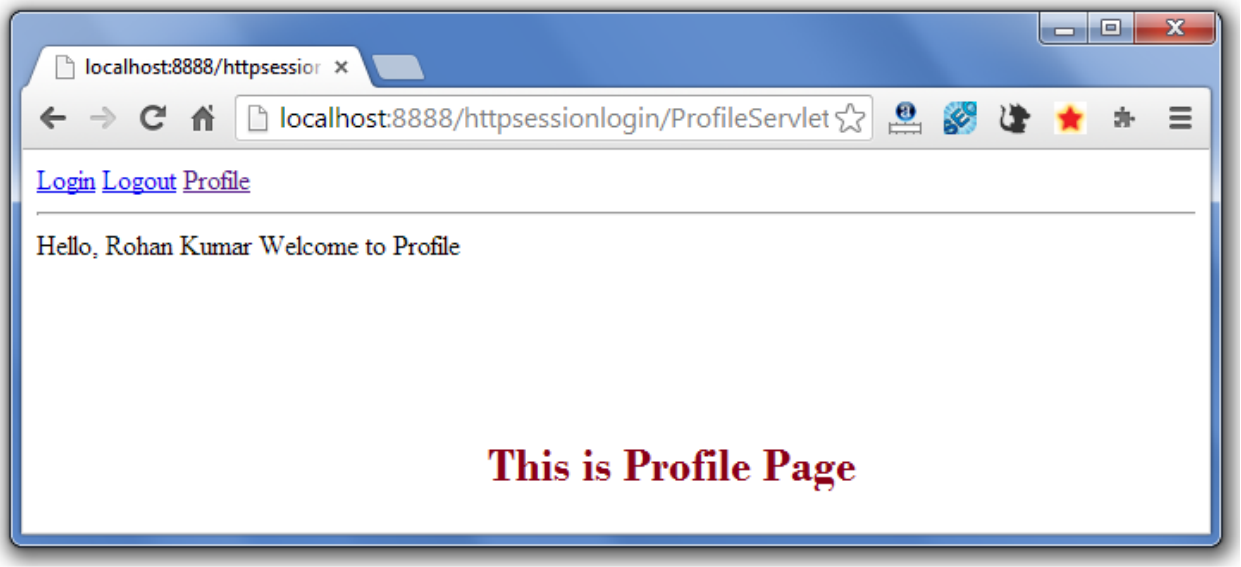
```
    <servlet-class>LogoutServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LogoutServlet</servlet-name>
    <url-pattern>/LogoutServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

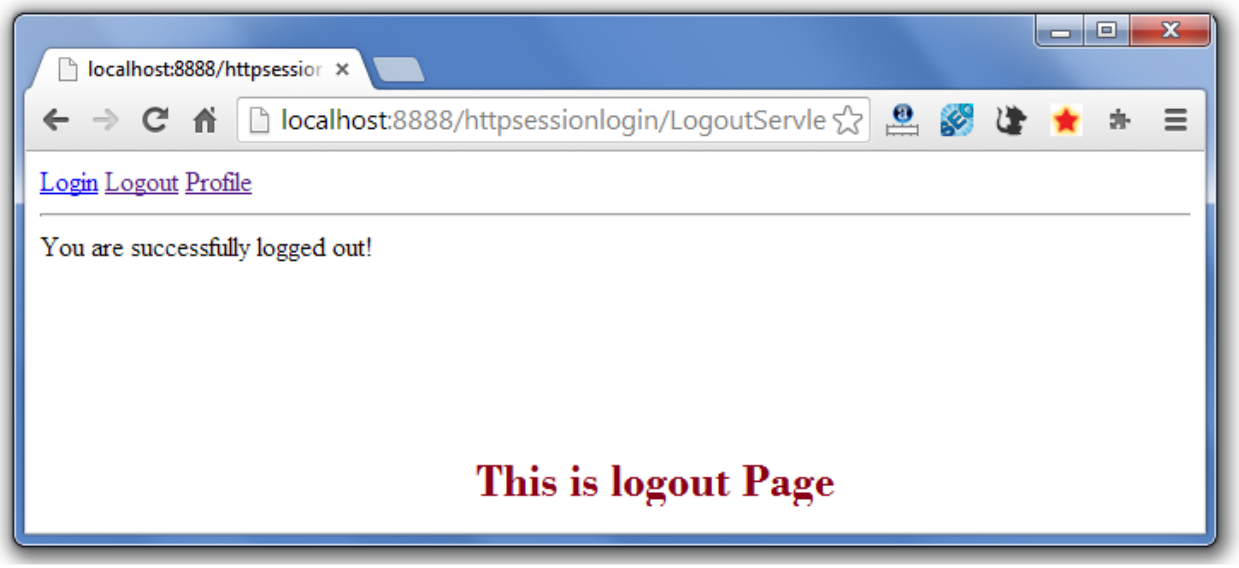
Output







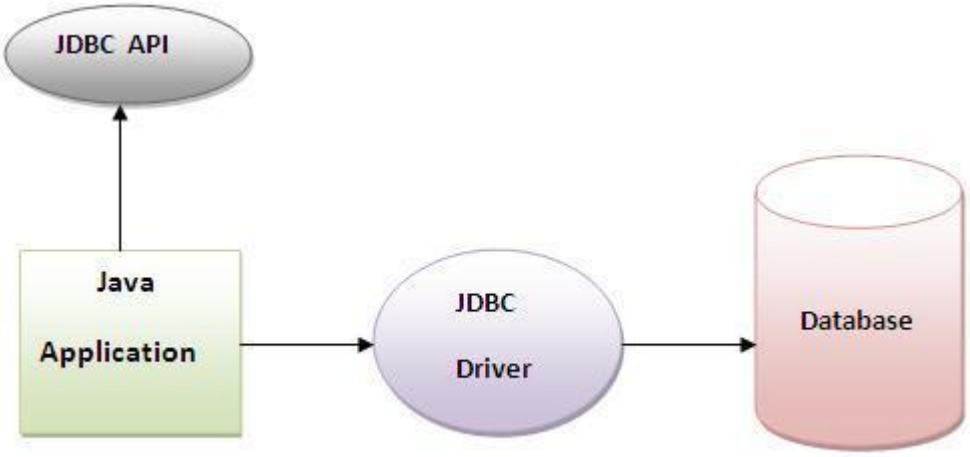



localhost:8888/httpsessior
localhost:8888/httpsessionlogin/LogoutServle
Login Logout Profile
You are successfully logged out!
This is logout Page

Module-16: Java Database Connectivity (JDBC)

Java JDBC

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



Why use JDBC

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

What is API

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:

- JDBC-ODBC bridge driver
- Native-API driver (partially java driver)
- Network Protocol driver (fully java driver)
- Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

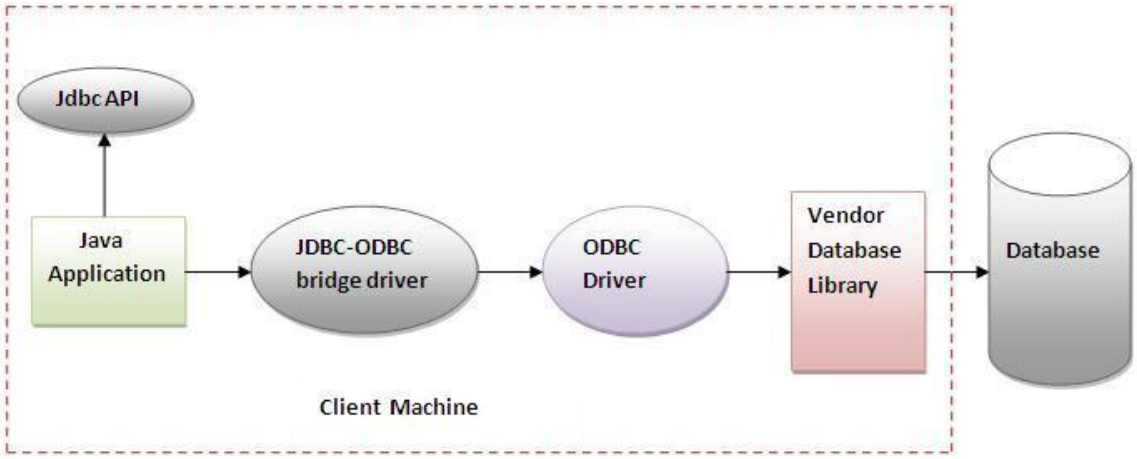


Figure- JDBC-ODBC Bridge Driver

Advantages:

- Easy to use.
- Can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

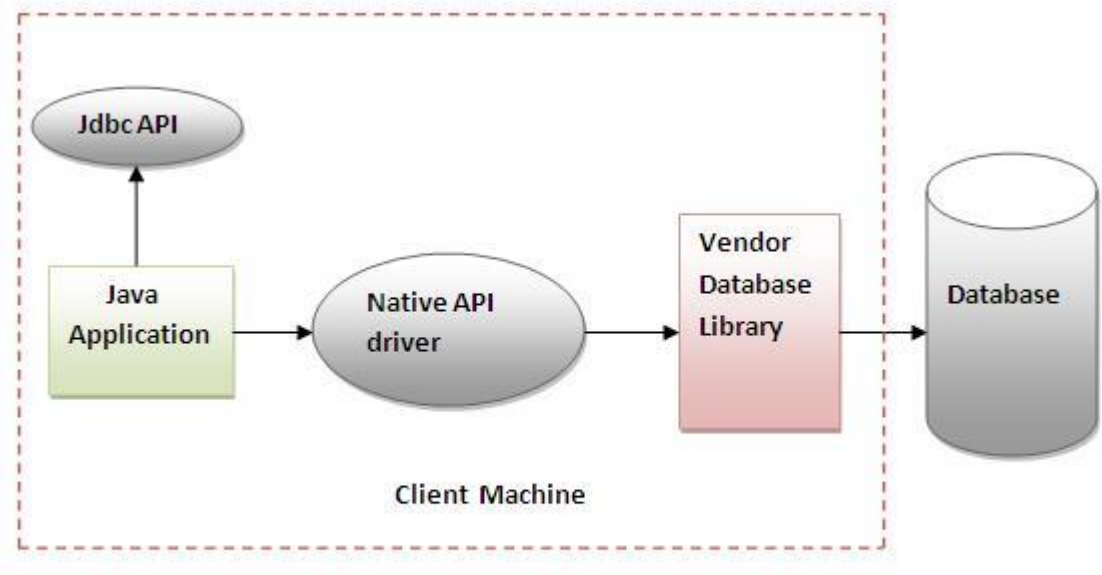


Figure- Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

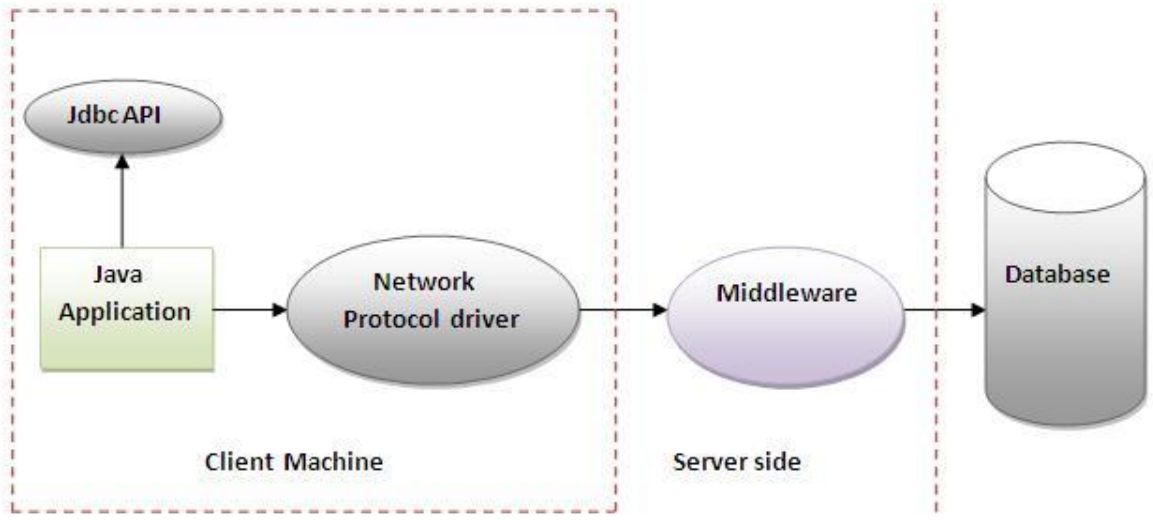


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

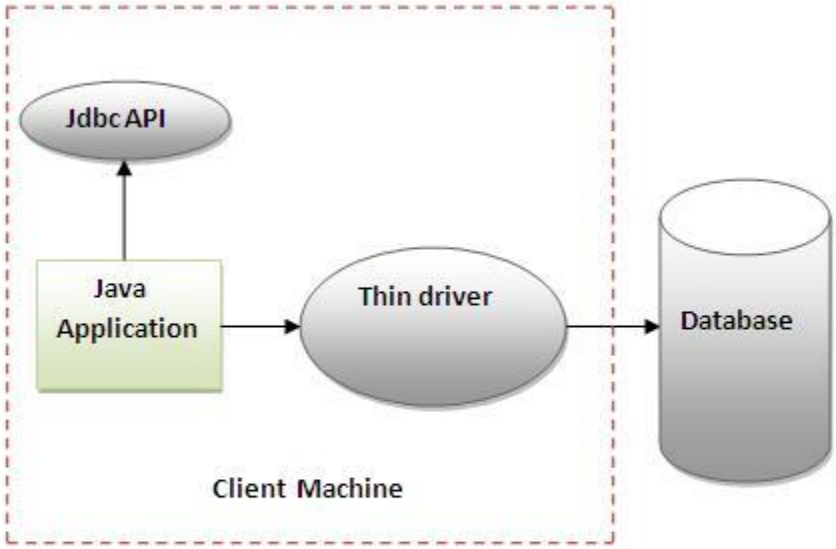


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection

- Creating statement
- Executing queries
- Closing connection

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException

2) public static Connection getConnection(String url,String name,String password) throws SQLException

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

```
public Statement createStatement()throws SQLException
```

Example to create the statement object

```
Statement stmt=con.createStatement();
```

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
public void close()throws SQLException
```

Example to close connection

```
con.close();
```

Example to connect to the mysql database

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

- **Driver class:** The driver class for the mysql database is com.mysql.jdbc.Driver.
- **Connection URL:** The connection URL for the mysql database is jdbc:mysql://localhost:3306/sonoo where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.
- **Username:** The default username for the mysql database is root.
- **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
create database sonoo;
use sonoo;
create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
```

```
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");

//here sonoo is database name, root is username and password

Statement stmt=con.createStatement();

ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

con.close();

}catch(Exception e){ System.out.println(e);}

}
}
```

DriverManager class:

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Commonly used methods of DriverManager class:

1) public static void registerDriver(Driver driver): is used to register the given driver with DriverManager.

2) public static void deregisterDriver(Driver driver):is used to deregister the given driver (drop the driver from the list) with DriverManager.

3) public static Connection getConnection(String url): is used to establish the connection with the specified url.

4) public static Connection getConnection(String url,String userName,String password): is used to establish the connection with the specified url, username and password.

Connection interface:

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(),rollback() etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status.By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
3) public boolean execute(String sql): is used to execute queries that may return multiple results.
4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

Let's see the simple example of Statement interface to insert, update and delete the record.

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where
id=33");
int result=stmt.executeUpdate("delete from emp765 where id=33");

System.out.println(result+" records affected");
con.close();
}}
```


Module 17 Introduction WebServices

Web Services

Web services are open standard (XML, SOAP, HTTP etc.) based Web applications that interact with other web applications for the purpose of exchanging data.

WSDL

WSDL stands for Web Services Description Language. It is an XML-based language for describing Web services.

SOAP

SOAP stands for Simple Object Access Protocol. SOAP is an XML based protocol for accessing Web Services. It is based on XML.

RDF

RDF stands for Resource Description Framework. RDF is a framework for describing resources on the web. It is written in XML

RSS

RSS stands for Really Simple Syndication. RSS allows you to syndicate your site content. It defines an easy way to share and view headlines and content. RSS files can be automatically updated and allows personalized views for different sites. RSS is written in XML

How Does a Web Service Work?

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of:

- XML to tag the data
- SOAP to transfer a message
- WSDL to describe the availability of service.

Example

Consider a simple account-management and order processing system. The accounting personnel use a client application built with Visual Basic or JSP to create new accounts and enter new customer orders.

The processing logic for this system is written in Java and resides on a Solaris machine, which also interacts with a database to store information.

The steps to perform this operation are as follows:

- The client program bundles the account registration information into a SOAP message.
- This SOAP message is sent to the web service as the body of an HTTP POST request.
- The web service unpacks the SOAP request and converts it into a command that the application can understand.
- The application processes the information as required and responds with a new unique account number for that customer.
- Next, the web service packages the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
- The client program unpacks the SOAP message to obtain the results of the account registration process.

Why Web Services:

Exposing the Existing Function on the network

A web service is a unit of managed code that can be remotely invoked using HTTP, that is, it can be activated using HTTP requests. Web services allows you to expose the functionality of your existing code over the network. Once it is exposed on the network, other application can use the functionality of your program.

Interoperability

Web services allow various applications to talk to each other and share data and services among themselves. Other applications can also use the web services. For example, a VB or .NET application can talk to Java web services and vice versa. Web services are used to make the application platform and technology independent.

Standardized Protocol

Web services use standardized industry standard protocol for the communication. All the four layers (Service Transport, XML Messaging, Service Description, and Service Discovery layers) use well-defined protocols in the web services protocol stack. This standardization of protocol stack gives the business many advantages such as a wide range of choices, reduction in the cost due to competition, and increase in the quality.

Low Cost of Communication

Web services use SOAP over HTTP protocol, so you can use your existing low-cost internet for implementing web services. This solution is much less costly compared to proprietary solutions like EDI/B2B. Besides SOAP over HTTP, web services can also be implemented on other reliable transport mechanisms like FTP.

Web Services - Characteristics

Web services have the following special behavioral characteristics:

XML-Based

Web Services uses XML at data representation and data transportation layers. Using XML eliminates any networking, operating system, or platform binding. Web Services based applications are highly interoperable application at their core level.

Loosely Coupled

A consumer of a web service is not tied to that web service directly. The web service interface can change over time without compromising the client's ability to interact with the service. A tightly coupled system implies that the client and server logic are closely tied to one another, implying that if one interface changes, the other must be updated. Adopting a loosely coupled architecture tends to make software systems more manageable and allows simpler integration between different systems.

Coarse-Grained

Object-oriented technologies such as Java expose their services through individual methods. An individual method is too fine an operation to provide any useful capability at a corporate level. Building a Java program from scratch requires the creation of several fine-grained methods that are then composed into a coarse-grained service that is consumed by either a client or another service.

Businesses and the interfaces that they expose should be coarse-grained. Web services technology provides a natural way of defining coarse-grained services that access the right amount of business logic.

Ability to be Synchronous or Asynchronous

Synchronicity refers to the binding of the client to the execution of the service. In synchronous invocations, the client blocks and waits for the service to complete its operation before continuing. Asynchronous operations allow a client to invoke a service and then execute other functions.

Asynchronous clients retrieve their result at a later point in time, while synchronous clients receive their result when the service has completed. Asynchronous capability is a key factor in enabling loosely coupled systems.

Supports Remote Procedure Calls (RPCs)

Web services allow clients to invoke procedures, functions, and methods on remote objects using an XML-based protocol. Remote procedures expose input and output parameters that a web service must support.

Component development through Enterprise JavaBeans (EJBs) and .NET Components has increasingly become a part of architectures and enterprise deployments over the past couple of years. Both technologies are distributed and accessible through a variety of RPC mechanisms.

A web service supports RPC by providing services of its own, equivalent to those of a traditional component, or by translating incoming invocations into an invocation of an EJB or a .NET component.

Supports Document Exchange

One of the key advantages of XML is its generic way of representing not only data, but also complex documents. These documents can be as simple as representing a current address, or they can be as complex as representing an entire book or Request for Quotation (RFQ). Web services support the transparent exchange of documents to facilitate business integration.

Over the past few years, three primary technologies have emerged as worldwide standards that make up the core of today's web services technology. These technologies are discussed below.

XML-RPC

This is the simplest XML-based protocol for exchanging information between computers.

- XML-RPC is a simple protocol that uses XML messages to perform RPCs.
- Requests are encoded in XML and sent via HTTP POST.
- XML responses are embedded in the body of the HTTP response.
- XML-RPC is platform-independent.
- XML-RPC allows diverse applications to communicate.
- A Java client can speak XML-RPC to a Perl server.
- XML-RPC is the easiest way to get started with web services.

SOAP

SOAP is an XML-based protocol for exchanging information between computers.

- SOAP is a communication protocol.
- SOAP is for communication between applications.
- SOAP is a format for sending messages.
- SOAP is designed to communicate via Internet.
- SOAP is platform independent.
- SOAP is language independent.
- SOAP is simple and extensible.
- SOAP allows you to get around firewalls.
- SOAP will be developed as a W3C standard.

WSDL

WSDL is an XML-based language for describing web services and how to access them.

- WSDL stands for Web Services Description Language.
- WSDL was developed jointly by Microsoft and IBM.
- WSDL is an XML based protocol for information exchange in decentralized and distributed environments.
- WSDL is the standard format for describing a web service.
- WSDL definition describes how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of UDDI, an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL is pronounced as 'wiz-dull' and spelled out as 'W-S-D-L'.

UDDI

UDDI is an XML-based standard for describing, publishing, and finding web services.

- UDDI stands for Universal Description, Discovery, and Integration.
- UDDI is a specification for a distributed registry of web services.
- UDDI is platform independent, open framework.
- UDDI can communicate via SOAP, CORBA, and Java RMI Protocol.
- UDDI uses WSDL to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.
- UDDI is an open industry initiative enabling businesses to discover each other and define how they interact over the Internet.

Module 18 Web Services Architecture

Web Services – Architecture

There are two ways to view the web service architecture:

The first is to examine the individual roles of each web service actor.

The second is to examine the emerging web service protocol stack.

Web Service Roles

There are three major roles within the web service architecture:

Service Provider

This is the provider of the web service. The service provider implements the service and makes it available on the Internet.

Service Requestor

This is any consumer of the web service. The requestor utilizes an existing web service by opening a network connection and sending an XML request.

Service Registry

This is a logically centralized directory of services. The registry provides a central place where developers can publish new services or find existing ones. It therefore serves as a centralized clearing house for companies and their services.

Web Service Protocol Stack

A second option for viewing the web service architecture is to examine the emerging web service protocol stack. The stack is still evolving, but currently has four main layers.

Service Transport

This layer is responsible for transporting messages between applications. Currently, this layer includes Hyper Text Transport Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), and newer protocols such as Blocks Extensible Exchange Protocol (BEEP).

XML Messaging

This layer is responsible for encoding messages in a common XML format so that messages can be understood at either end. Currently, this layer includes XML-RPC and SOAP.

Service Description

This layer is responsible for describing the public interface to a specific web service. Currently, service description is handled via the Web Service Description Language (WSDL).

Service Discovery

This layer is responsible for centralizing services into a common registry and providing easy publish/find functionality. Currently, service discovery is handled via Universal Description,

Discovery, and Integration (UDDI).

As web services evolve, additional layers may be added and additional technologies may be added to each layer.

Hyper Text Transfer Protocol (HTTP)

Currently, HTTP is the most popular option for service transport. HTTP is simple, stable, and widely deployed. Furthermore, most firewalls allow HTTP traffic. This allows XML-RPC or SOAP messages to masquerade as HTTP messages. This is good if you want to integrate remote applications, but it does raise a number of security concerns.

Blocks Extensible Exchange Protocol (BEEP)

This is a promising alternative to HTTP. BEEP is a new Internet Engineering Task Force (IETF) framework for building new protocols. BEEP is layered directly on TCP and includes a number of built-in features, including an initial handshake protocol, authentication, security, and error

handling. Using BEEP, one can create new protocols for a variety of applications, including instant messaging, file transfer, content syndication, and network management.

SOAP is not tied to any specific transport protocol. In fact, you can use SOAP via HTTP, SMTP, or FTP. One promising idea is therefore to use SOAP over BEEP.

What is REST?

REST stands for **RE**presentational **S**tate **T**ransfer. REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Now a days JSON is the most popular format being used in web services.

HTTP Methods

Following well known HTTP methods are commonly used in REST based architecture.

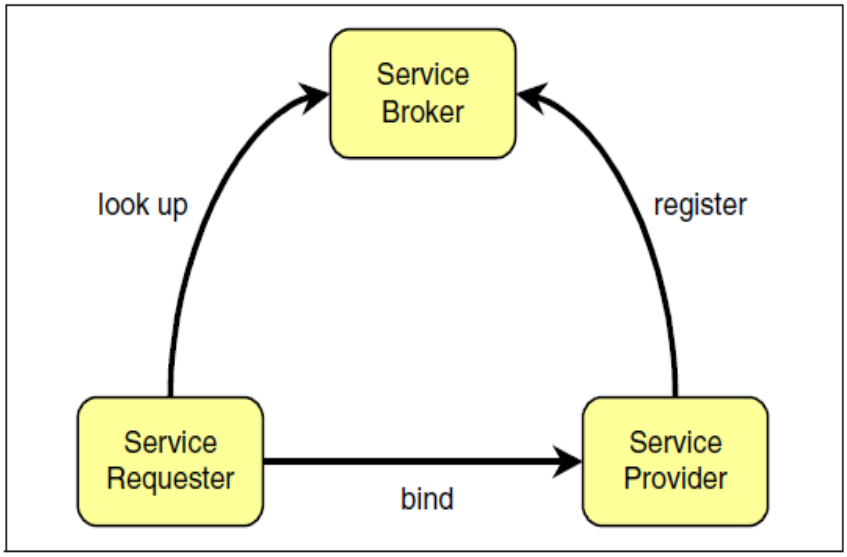
- **GET** - Provides a read only access to a resource.
- **PUT** - Used to create a new resource.
- **DELETE** - Used to remove a resource.
- **POST** - Used to update a existing resource or create a new resource.
- **OPTIONS** - Used to get the supported operations on a resource.

RESTFul Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like

the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.



SOAP vs RESTful Web Services

SOA - Introduction:

Service-Oriented Architecture (SOA) is a new way of thinking about enterprise IT Architecture. SOA is about associating business process with IT. SOA represents a natural evolution of proven software architectural principles or design patterns commonly implemented in object-oriented systems.

No.	SOAP	REST
1)	SOAP is a protocol.	REST is an architectural style.
2)	SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
3)	SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
4)	SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
5)	JAX-WS is the java API for SOAP web services.	JAX-RS is the java API for RESTful web services.
6)	SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
7)	SOAP requires more bandwidth and resource than REST.	REST requires less bandwidth and resource than SOAP.
8)	SOAP defines its own security.	RESTful web services inherits security measures from the underlying transport.
9)	SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
10)	SOAP is less preferred than REST.	REST more preferred than SOAP.

In short SOA is the concept that defines an architecture which gives a conduit to deliver business processes as services and increase the growth for business applications.

Service

A service is well-defined, self-contained function that represents unit of functionality. A service can exchange information from another service. It is not dependent on the state of another service.

Over the past few years, three primary technologies have emerged as worldwide standards that make up the core of today's web services technology. These technologies are discussed below.

XML-RPC

This is the simplest XML-based protocol for exchanging information between computers.

- XML-RPC is a simple protocol that uses XML messages to perform RPCs.
- Requests are encoded in XML and sent via HTTP POST.
- XML responses are embedded in the body of the HTTP response.
- XML-RPC is platform-independent.
- XML-RPC allows diverse applications to communicate.
- A Java client can speak XML-RPC to a Perl server.
- XML-RPC is the easiest way to get started with web services.

SOAP

SOAP is an XML-based protocol for exchanging information between computers.

- SOAP is a communication protocol.
- SOAP is for communication between applications.
- SOAP is a format for sending messages.
- SOAP is designed to communicate via Internet.
- SOAP is platform independent.

- SOAP is language independent.
- SOAP is simple and extensible.
- SOAP allows you to get around firewalls.
- SOAP will be developed as a W3C standard.

WSDL

WSDL is an XML-based language for describing web services and how to access them.

- WSDL stands for Web Services Description Language.
- WSDL was developed jointly by Microsoft and IBM.
- WSDL is an XML based protocol for information exchange in decentralized and distributed environments.
- WSDL is the standard format for describing a web service.
- WSDL definition describes how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of UDDI, an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL is pronounced as 'wiz-dull' and spelled out as 'W-S-D-L'.

UDDI

UDDI is an XML-based standard for describing, publishing, and finding web services.

- UDDI stands for Universal Description, Discovery, and Integration.
- UDDI is a specification for a distributed registry of web services.
- UDDI is platform independent, open framework.
- UDDI can communicate via SOAP, CORBA, and Java RMI Protocol.
- UDDI uses WSDL to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.

- UDDI is an open industry initiative enabling businesses to discover each other and define how they interact over the Internet.

Module 19 Java Web Services

Web Services with JAX-WS

Java API for XML Web Services (JAX-WS) is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as Remote Procedure Call-oriented (RPC-oriented) web services.

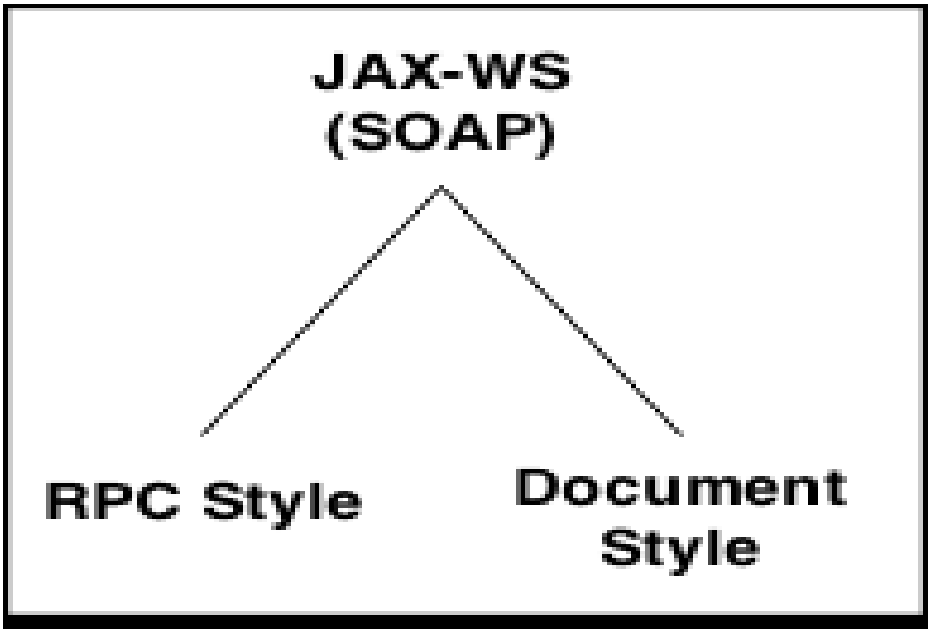
In JAX-WS, a web service operation invocation is represented by an XML-based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

JAX-WS provides concepts and examples of JAX-WS API.

This JAX-WS is designed for beginners and professionals.

There are two ways to develop JAX-WS example.

- RPC style
- Document style.



Difference between RPC and Document web services:

Difference between RPC and Document web services

There are many differences between RPC and Document web services. The important differences between RPC and Document are given below:

RPC Style

1) RPC style web services use method name and parameters to generate XML structure.

2) The generated WSDL is difficult to be validated against schema.

3) In RPC style, SOAP message is sent as many elements.

4) RPC style message is tightly coupled.

5) In RPC style, SOAP message keeps the operation name.

6) In RPC style, parameters are sent as discrete values.

Document Style

1) Document style web services can be validated against predefined schema.

2) In document style, SOAP message is sent as a single document.

3) Document style message is loosely coupled.

4) In Document style, SOAP message loses the operation name.

5) In Document style, parameters are sent in XML format.
