Modern Programming Languages

Lecture 27-30

Java Programming Language

An Introduction

Java was developed at Sun in the early 1990s and is based on C++. It looks very similar to C++ but it is significantly simplified as compared to C++. That is why Professor Feldman says that Java is C++--. It supports only OOP and has eliminated multiple inheritance, pointers, structs, enum types, operator overloading, goto statement from C++. It includes support for applets and a form of concurrency.

The First Program

Here is the famous Hello World program in Java.

```
class HelloWorld {
        public static void main( String [ ] args )
 {
                System.out.println("Hello world!");
 }
}
```

It may be noted that as Java supports only OOP, in Java, every variable, constant, and function (including *main*) must be inside some class. Therefore, there are no global variables or functions in Java. In addition, the following may be noted:

- Function main is member of the class.
- Every class may have a main function; there may be more than one main function in a Java program.
- The main must be public static void.
- The main may have one argument: an array of String. This array contains the command-line arguments.
- There is no final semi-colon at the end of the class definition.

Java Files

In Java the source code is written in the .java file. The following restrictions apply:
(1) Each source file can contain at most one *public* class.
(2) If there is a public class, then the class name and file name must match.

Furthermore, every function must be part of a class and every class is part of a package. A *public* class can be used in *any* package and a non-public class can only be used in its own package.

The Java Bytecode is created by the Java compiler and is stored in .class file. This file contains ready to be exected Java bytecode which is executed by the Java Virtual Machine. For each class in a source file (both public and non-public classes), the compiler creates one ".class" file, where the file name is the same as the class name. When compiling a program, you type the full file name, including the ".java" extension;

When running a program, you just type the name of the class whose *main* function you want to run.

Java Types

Java has two "categories" of types: *primitive types* and *reference* types.

Primitive Types

All the primitive types have specified sizes that are machine independent for portability. This includes:

boolean	same as bool in C++
char	holds one 16 bit unicode character
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
float	floating-point number
double	double precision floating-point number

Reference Types

arrays classes

There are no struct, union, enum, unsigned, typedef, or pointers types in Java.

C++ Arrays vs Java Arrays

In C++, when you declare an array, storage for the array is allocated. In Java, when you declare an array, you are really only declaring a pointer to an array; storage for the array itself is not allocated until you use "new". This difference is elaborated as shown below:

C++

```
int A[10];          // A is an array of length 10
A[0] = 5;           // set the 1st element of array A
```

JAVA

```
int [ ] A;          // A is a reference to an array
A = new int [10];   // now A points to an array of length 10
A[0] = 5;           // set the 1st element of the array pointed to by A
```

In both C++ and Java you can initialize an array using values in curly braces. Here's example Java code:

```
int [ ] myArray = {13, 12, 11};
                        //  myArray points to an array of length 3
                        // containing the values 13, 12, and 11
```

In Java, a default initial value is assigned to each element of a newly allocated array if no initial value is specified. The default value depends on the type of the array element as shown below:

Type	**Value**
boolean	false
char	'\u0000'
byte, int, short, long, float, double	0
any reference	null

In Java, array bounds are checked and an out-of-bounds array index always causes a runtime error.

In Java, you can also determine the current length of an array (at runtime) using ".length" operator as shown below:

```
int [ ] A = new int[10];
...
A.length ...          // this expression evaluates to 10
A = new int[20];
...
A.length ...          // now it evaluates to 20
```

In Java, you can copy an array using the *arraycopy* function. Like the output function *println*, arraycopy is provided in java.lang.System, so you must use the name System.arraycopy. The function has five parameters:

src: the source array (the array from which to copy)
srcPos: the starting position in the source array
dst: the destination array (the array into which to copy)
dstPos: the starting position in the destination array
count: how many values to copy

Here is an example:

```
int [ ] A, B;
A = new int[10];

// code to put values into A
B = new int[5];
System.arraycopy(A, 0, B, 0, 5)   // copies first 5 values from A to B
System.arraycopy(A, 9, B, 4, 1)   // copies last value from A into last
                                  // element of B
```

Note that the destination array must already exist (i.e., *new* must already have been used to allocate space for that array), and it must be large enough to hold all copied values

(otherwise you get a runtime error). Furthermore, the source array must have enough values to copy (i.e., the length of the source array must be at least srcPos+count). Also, for arrays of primitive types, the types of the source and destination arrays must be the same. For arrays of non-primitive types, System.arraycopy(A, j, B, k, n) is OK if the assignment B[0] = A[0] would be OK.

The arraycopy function also works when the source and destination arrays are the *same* array; so for example, you can use it to "shift" the values in an array:

```
int [ ] A = {0, 1, 2, 3, 4};
System.arraycopy(A, 0, A, 1, 4);
```

After executing this code, A has the values: [0, 0, 1, 2, 3].

As in C++, Java arrays can be *multidimensional*. For example, a 2-dimensional array is an array of arrays. However, a two-dimensional arrays need not be rectangular. Each row can be a different length. Here's an example:

```
int [ ][ ] A;              // A is a two-dimensional array
A = new int[5][];          // A now has 5 rows, but no columns yet
A[0] = new int [1];        // A's first row has 1 column
A[1] = new int [2];        // A's second row has 2 columns
A[2] = new int [3];        // A's third row has 3 columns
A[3] = new int [5];        // A's fourth row has 5 columns
A[4] = new int [5];        // A's fifth row also has 5 columns
```

C++ Classes vs Java Classes

In C++, when you declare a variable whose type is a class, storage is allocated for an object of that class, and the class's constructor function is called to initialize that instance of the class. In Java, you are really declaring a pointer to a class object; no storage is allocated for the class object, and no constructor function is called until you use "new".

This is elaborated with the help of the following example:

Let us assume that we have the following class:

```
class MyClass {
        ...
}
```

Let us first look at C++

```
MyClass m;   // m is an object of type MyClass;
             //  the constructor function is called to initialize M.
```

```
MyClass  *pm;
                // pm is a pointer to an object of type MyClass
                // no object exists yet, no constructor function
                // has been called

pm = new MyClass;
                // now storage for an object of MyClass has been
                // allocated and the constructor function has been
                // called
```

Now the same thing in Java

```
MyClass m;   // pm is a pointer to an object of type MyClass
             // no object exists yet, no constructor function
             // has been called

m = new MyClass();
             // now storage for an object of MyClass has been allocated
             // and the constructor function has been called. Note
             // that you must use parentheses even when you are not
             // passing any arguments to the constructor function

             // Also note that there is a simple '.' (dot) operator used to
             // access members or send message. Java does not use
             // -> operator.
```

Whereas, as opposed to Java, in C++ use have the following:

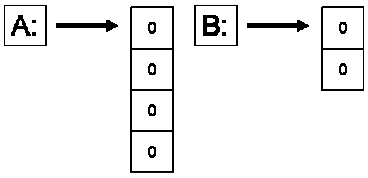
```
MyClass m;          // m is an object of type MyClass;
                    // the constructor function is called to initialize M.

MyClass  *pm;       // pm is a pointer to an object of type MyClass
                    // no object exists yet, no constructor function has
                    // been called

pm = new MyClass; // now storage for an object of MyClass has been
                    // allocated and the constructor function has been
                    // called
```

Aliasing Problems in Java

The fact that arrays and classes are really pointers in Java can lead to some problems. Here is a simple assignment that causes *aliasing*:

```
int [] A = new int [4];
Int [] B = new int [2];
```

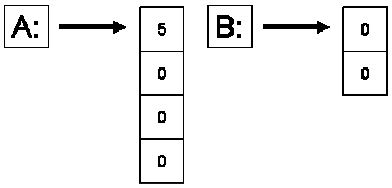
This is depicted as below:



Now, when we say:

```
A[0] = 5;
```

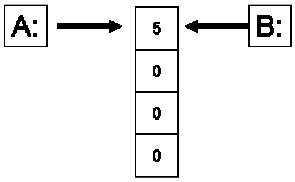
We get the following:



Now when we say:

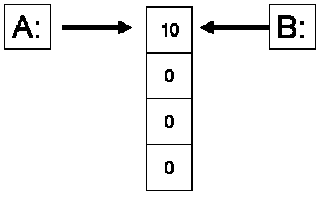
```
B = A;
```

B points to the same array as A and creates an alias. This is shown below:



Now if we make a simple assignment in B, we will also change A as shown below:

```
B[0] = 10;
```



This obviously creates problems. Therefore, as a programmer you have to be very careful when writing programs in Java.

In Java, all parameters are passed by value, but for arrays and classes the actual parameter is really a pointer, so changing an array element, or a class field inside the function *does* change the actual parameter's element or field.

This is elaborated with the help of the following example:

```
void f( int [ ] A ) {
        A[0] = 10;      // change an element of parameter A
        A = null;       // change A itself
}

void g() {
        int [ ] B = new int [3];
        B[0] = 5;
        f(B);
                // B is not null here, because B itself was passed by value
                // however, B[0] is now 10, because function f changed the
                // first element of the array
}
```

Note that, in C++, similar problems can arise when a class that has pointer data members is passed by value. This problem is addressed by the use of copy constructors, which can be defined to make copies of the values pointed to, rather than just making copies of the pointers. In Java, the solution is to use the *arraycopy* operation, or to use a class's *clone* operation. Cloning will be discussed later.

Type Conversion

Java is much stronger than C++ in the type conversions that are allowed.
Here we discuss conversions among primitive types. Conversions among class objects will be discussed later.

Booleans cannot be converted to other types. For the other primitive types (char, byte, short, int, long, float, and double), there are two kinds of conversion: *implicit* and *explicit*.

Implicit conversions:

An implicit conversion means that a value of one type is changed to a value of another type without any special directive from the programmer. A char can be implicitly converted to an int, a long, a float, or a double. For example, the following will compile without error:

```
char c = 'a';
int k = c;
long x = c;
float y = c;
double d = c;
```

For the other (numeric) primitive types, the basic rule is that implicit conversions can be done from one type to another if the range of values of the first type is a subset of the range of values of the second type. For example, a byte can be converted to a short, int, long or float; a short can be converted to an int, long, float, or double, etc.

Explicit conversions:

Explicit conversions are done via *casting*: the name of the type to which you want a value converted is given, in parentheses, in front of the value. For example, the following code uses casts to convert a value of type double to a value of type int, and to convert a value of type double to a value of type short:

```
double d = 5.6; int k = (int)d; short s = (short)(d * 2.0);
```

Casting can be used to convert among any of the primitive types except boolean. Note, however, that casting can lose information; for example, floating-point values are truncated when they are cast to integers (e.g., the value of k in the code fragment given above is 5), and casting among integer types can produce wildly different values (because upper bits, possibly including the sign bit, are lost).

JAVA CLASSES

Java classes contain *fields* and *methods*. A field is like a C++ data member, and a method is like a C++ member function. Each field and method has an *access level*:

- private: accessible only in this class
- (package): accessible only in this package
- protected: accessible only in this package and in all subclasses of this class
- public: accessible everywhere this class is available

Similarly, each class has one of two possible access levels:

- (package): class objects can only be declared and manipulated by code in this package
- public: class objects can be declared and manipulated by code in any package

Note: for both fields and classes, package access is the default, and is used when *no* access is specified.

A Simple Example Class

In the following example, a List is defined to be an ordered collection of items of any type:

```
class List {
        // fields
        private Object [ ] items;      // store the items in an array
        private int   numItems;        // the current # of items in the list
                                       // methods

        // constructor function
        public List()
        {
                items = new Object[10];
                numItems = 0;
        }

        // AddToEnd: add a given item to the end of the list
        public void AddToEnd(Object ob)
        {
        ...
        }
}
```

In Java, all classes (built-in or user-defined) are (implicitly) subclasses of the class **Object**. Using an array of Object in the List class allows any kind of Object (an instance of any class) to be stored in the list. However, primitive types (int, char, etc) cannot be stored in the list as they are not inherited from Object.

Constructor function:

As in C++, constructor functions in Java are used to initialize each instance of a class. They have no return type (not even void) and can be overloaded; you can have multiple constructor functions, each with different numbers and/or types of arguments. If you don't write any constructor functions, a default (no-argument) constructor (that doesn't do anything) will be supplied.

If you write a constructor that takes one or more arguments, no default constructor will be supplied (so an attempt to create a new object without passing any arguments will cause a compile-time error). It is often useful to have one constructor call another (for example, a constructor with no arguments might call a constructor with one argument, passing a default value). The call must be the *first* statement in the constructor. It is performed using *this* as if it were the name of the method. For example:

```
this( 10 );
```

is a call to a constructor that expects one integer argument.

Initialization of fields:

If you don't initialize a field (i.e., either you don't write any constructor function, or your constructor function just doesn't assign a value to that field), the field will be given a default value, depending on its type. The values are the same as those used to initialize newly created arrays (see the "Java vs C++" notes).

Access Control:

Note that the access control must be specified for every field and every method; there is no grouping as in C++. For example, given these declarations:

```
public
        int x;
        int y;
```

only x is public; y gets the default, package access.

Static Fields and Methods

Fields and methods can be declared *static*. If a field is static, there is only one copy for the entire class, rather than one copy for each instance of the class. (In fact, there is a copy of the field even if there are *no* instances of the class.) A method should be made static when it does not access any of the non-static fields of the class, and does not call any non-static methods. (In fact, a static method *cannot* access non-static fields or call non-static methods.) Methods that would be "free" functions in C++ (i.e., not members of any class) should be static methods in Java. A public static field or method can be accessed from outside the class.

Final Fields and Methods

Fields and methods can also be declared *final*. A final method cannot be overridden in a subclass. A final field is like a constant: once it has been given a value, it cannot be assigned to again.

Some Useful Built-in Classes

Following is a list of some useful classes in Java. These classes are not really part of the language; they are provided in the package *java.lang*

1. String

String Creation:

```
String S1 = "hello",          // initialize from a string literal
S2 = new String("bye"),       // use new and the String constructor
S3 = new String(S1);          // use new and a different constructor
```

String concatenation

```
String S1 = “hello ” + “world”;
String S2 = S1 + “!”;
String S3 = S1 + 10;
```

2. Object

Object is the Base class for all Java Classes.

3. Classes for primitive types

In Java, we have classes also for primitive types. These are: Boolean, Integer, Double, etc. Note that variable of bool, int, etc types are not objects whereas variable of type Boolean, Integer, etc are objects.

Packages

Every class in Java is part of some *package*. All classes in a file are part of the same package. You can specify the package using a *package declaration*:

package *name* ;

as the first (non-comment) line in the file. Multiple files can specify the same package name. If no package is specified, the classes in the file go into a special unnamed package (the same unnamed package for all files). If package *name* is specified, the file must be in a subdirectory called *name* (i.e., the directory name must match the package name).

You can access public classes in another (named) package using:

package-name.class-name

You can access the public fields and methods of such classes using:

package-name.class-name.field-or-method-name

You can avoid having to include the *package-name* using import *package-name*.* or import *package-name.class-name* at the beginning of the file (after the package declaration). The former imports all of the classes in the package, and the second imports just the named class.

You must still use the *class-name* to access the classes in the packages, and *class-name.field-or-method-name* to access the fields and methods of the class; the only thing you can leave off is the package name.

Inheritance

Java directly supports single inheritance. To support concept of a **interface** class is used. Inheritance is achieved as shown below:

```
class SuperClass {
        …
}

class SubClass extends SuperClass {
        …
}
```

When a class is inherited, all fields and methods are inherited. When the **final** reserved word is specified on a class specification, it means that class cannot be the parent of any class. *Private* fields are inherited, but cannot be accessed by the methods of the subclass. fields can be defined to be *protected*, which means that subclass methods can access them

Each superclass method (except its constructors) can be inherited, overloaded, or overridden. These are elaborated in the following paragraphs:

inherited: If no method with the same name is (re)defined in the subclass, then the subclass has that method with the same implementation as in the superclass.

overloaded: If the subclass defines a method with the same name, but with a different number of arguments or different argument types, then the subclass has *two* methods with that name: the old one defined by the superclass, and the new one it defined.

overridden: If the subclass defines a method with the same name, and the same number and types of arguments, then the subclass has only *one* method with that name: the new one it defined. A method cannot be overridden if it is defined as final in the superclass

Dynamic Binding

In C++, a method must be defined to be virtual to allow dynamic binding. In Java all method calls are dynamically bound unless the called method has been defined to be **final**, in which case it cannot be overridden and all bindings are static.

Constructors

A subclass's constructors *always* call a super class constructor, either explicitly or implicitly. If there is no explicit call (and no call to another of the subclass constructors), then the no-argument version of the superclass constructor is called before executing any statements.

Abstract Classes

An *abstract method* is a method that is declared in a class, but not defined. In order to be instantiated, a subclass must provide the definition. An *abstract class* is any class that includes an abstract method. It is similar to Pure virtual in C++.

If a class includes an abstract method, the class *must* be declared abstract, too. For example:

```
abstract class AbstractClass {
        abstract public void Print();
                // no body, just the function header
}
class MyConcreteClass extends AbstractClass {
        public void Print() { // actual code goes here } }
```

An abstract class cannot be instantiated. A subclass of an abstract class that does not provide bodies for all abstract methods must also be declared abstract. A subclass of a

non-abstract class can override a (non-abstract) method of its superclass, and declare it abstract. In that case, the subclass must be declared abstract.

Interface

As mentioned earlier, multiple inheritance is achieved through Interface in Java. Inheritance implements the "is-a" relationship whereas an interface is similar to a class, but can only contain public, static, final fields (i.e., constants) and public, abstract methods (i.e., just method headers, no bodies).

An interface is declared as shown below:

```
public interface Employee {
        void RaiseSalary( double d );
        double GetSalary();
}
```

Note that both methods are implicitly public and abstract (those keywords can be provided, but are not necessary).

A class can *implement* one or more interfaces (in addition to extending one class). It must provide bodies for all of the methods declared in the interface, or else it must be abstract. For example:

```
public class TA implements Employee {
        void RaiseSalary( double d ) {
                // actual code here
        }
        double GetSalary() {
                // actual code here
        }
}
```

Public interfaces (like public classes) must be in a file with the same name. Many classes can implement the same interface thus achieving multiple inheritance. An interface can be a "marker" with *no* fields or methods, is used only to "mark" a class as having a property, and is testable via the *instanceof* operator.

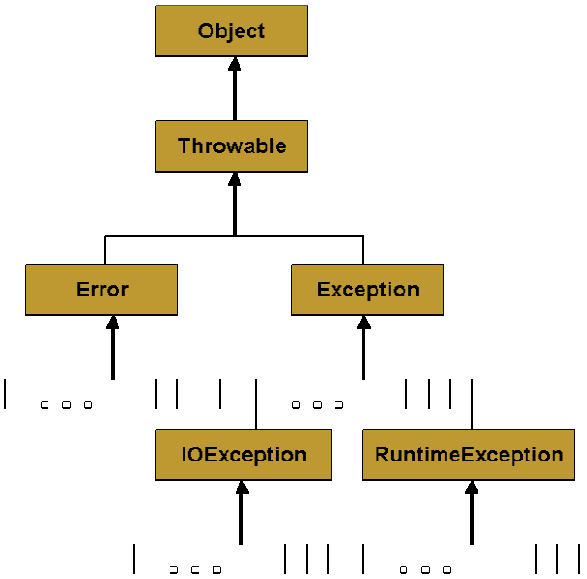
Exception Handling in Java

Exception handling in Java is based on C++ but is designed to be more in line with OOP. It includes a collection of predefined exceptions that are implicitly raised by the JVM. All java exceptions are objects of classes that are descendents of Throwable class. There are two predefined subclasses of Throwable: Error and Exception.

Error and its descendents are related to errors thrown by JVM. Examples include out of heap memory. Such an exception is never thrown by the user programs and should not be handled by the user.

User programs can define their own exception classes. Convention in Java is that such classes are subclasses of Exception. There are two predefined descendents of Exception: IOException and RuntimeException. IOException deals with errors in I/O operation.
In the case of RuntimeException there are some predefined exceptions which are, in many cases, thrown by JVM for errors such as out of bounds, and Null pointer.

The following diagram shows the exception hierarchy in Java.



Checked and Unchecked Exceptions

Exceptions of class Error and RuntimeException are called unchecked exceptions. They are never a concern of the compiler. A program can catch unchecked exceptions but it is not required. All other are checked exceptions. Compiler ensures that all the checked exceptions a method can throw are either listed in its throws clause or are handled in the method.

Exception Handlers

Exception handler in Java is similar to C++ except that the parameter of every **catch** must be present and its class must be descendent of Thrwoable. The syntax of **try** is exactly same as C++ except that there is **finally** clause as well. For example:

```
class MyException extends Exception {
```

```
        public MyException() { }
        public MyException(String message) {
                super (message);
        }
}
```

This exception can be thrown with

```
throw new MyException();
```

Or

```
MyException myExceptionObject = new MyException();
…
throw myExceptionObject;
```

Binding of exception is also similar to C++. If an exception is thrown in the compound statement of try construct, it is bound to the first handler (catch function) immediately following the try clause whose parameter is the same class as the thrown object or an ancestor of it. Exceptions can be handled and then re-thrown by including a throw statement without an operand at the end of the handler. To ensure that exceptions that can be thrown in a try clause are always handled in a method, a special handler can be written that matches all exceptions. For example:

```
catch (Exception anyException) {
        …
}
```

Other Design Choices

The exception handler parameter in C++ has a limited purpose. During program execution, the Java runtime system stores the class name of every object. getClass can be used to get an object that stores the class name. It has a getName method. The name of the class of the actual parameter of the throw statement can be retrieved in the handler as shown below.

```
anyException.getClass().getName();
```

In the case of a user defined exception, the thrown object could include any number of data fields that might be useful in the handler.

throws Clause

throws clause is overloaded in C++ and conveys two different meanings: one as specification and the other as command. Java is similar in syntax but different in semantics. The appearance of an exception class name in the **throws** clause of Java method specifies that the exception class or any of its descendents can be thrown by the method.

A C++ program unit that does not include a throw clause can throw any exceptions. A Java method that does not include a throws cannot throw any checked exception it does not handle. A method cannot declare more exceptions in its throws clause than the methods it overrides, though it may declare fewer. A method that does not throw a particular exception, but calls another method that could throw the exception, must list the exception in its throws clause.

The finally clause

A Java exception handler may have a finally clause. A finally clause *always* executes when its try block executes (whether or not there is an exception). The finally clause is written as shown below:

```
try {
        …
}
catch (…) {
        …
}
…
finally {
        …
}
```

A finally clause is usually included to make sure that some clean-up (e.g., closing opened files) is done. If the finally clause includes a transfer of control statement (return, break, continue, throw) then that statement overrides any transfer of control initiated in the try or in a catch clause.

First, let's assume that the finally clause does not include any transfer of control. Here are the situations that can arise:

- No exception occurs during execution of the try, and no transfer of control is executed in the try. In this case the finally clause executes, then the statement following the try block.
- No exception occurs during execution of the try, but it *does* execute a transfer of control. In this case the finally clause executes, then the transfer of control takes place.
- An exception *does* occur during execution of the try, and there is no catch clause for that exception. Now the finally clause executes, then the uncaught exception is "passed up" to the next enclosing try block, possibly in a calling function.
- An exception *does* occur during execution of the try, and there *is* a catch clause for that exception. The catch clause does not execute a transfer of control. In this case the catch clause executes, then the finally clause, then the statement following the try block.

- An exception *does* occur during execution of the try, there *is* a catch clause for that exception, and the catch clause *does* execute a transfer of control. Here, the catch clause executes, then the finally clause, then the transfer of control takes place.

If the finally block *does* include a transfer of control, then that takes precedence over any transfer of control executed in the try or in an executed catch clause. So for all of the cases listed above, the finally clause would execute, then *its* transfer of control would take place. Here's one example:

```
try {
        return 0;
} finally {
        return 2;
}
```

The result of executing this code is that 2 is returned. Note that this is rather confusing! The moral is that you probably do *not* want to include transfer-of-control statements in both the try statements and the finally clause, or in both a catch clause and the finally clause.

Java Threads

Java supports concurrency through threads which are lightweight processes. A thread is similar to a real process in that a thread and a running program are threads of execution. A thread takes advantage of the resources allocated for that program instead of having to allocate those resources again. A thread has its own stack and program counter. The code running within the thread works only within the context implied by the stack and PC so also called an execution context.

Creating Java Threads

There are two ways to create our own **Thread** object:

1. Subclassing the **Thread** class and instantiating a new object of that class
2. Implementing the **Runnable** interface

In both cases the **run()** method should be implemented. This is elaborated with the help of following examples:

Example of Extending Thread

```
public class ThreadExample extends Thread {
        public void run () {
        for (int i = 1; i <= 100; i++) {
```

```
                    System.out.println("Thread: " + i);
                }
            }
        }
```

Example of Implementing Runnable

```
    public class RunnableExample implements Runnable {
        public void run () {
            for (int i = 1; i <= 100; i++) {
            System.out.println ("Runnable: " + i);
        }
        }
    }
```

It may be noted that both of these are very similar to each other with minor syntactic and semantic differences.

Starting the Threads

A thread is started by simply sending the **start** message to the thread. This is shown in the following example:

```
public class ThreadsStartExample {
    public static void main (String argv[]) {
        new ThreadExample ().start ();
            new Thread(new RunnableExample ()).start ();
    }
}
```

Thread Methods

Some of the common thread methods are listed below:

- start()
- sleep()
- yield()
- run()
- wait()
- notify()
- notifyAll()
- setPriority()

Thread Synchronization - Wait and Notify

Threads are based upon the concept of a Monitor. The **wait** and **notify** methods are used just like **wait** and **signal** in a Monitor. They allow two threads to cooperate and based on a single shared lock object. The following example of a producer-consumer problem elaborates this concept.

```
class Buffer {
        private int [ ] buf;
        private int head, tail, max, size;

        public Buffer (int buf_size) {
                buf = new int [buf_size];
                head =0; tail = 0; size = 0;
                max = buf_size;
        }

        public synchronized void deposit (int item) {
                try {
                        while (size ==max) wait();
                        buf [tail] = item;
                        tail = (tail + 1) % max;
                        size++;
                        notify();
                }
                catch
                (interruptedException e) {
                        // exception handler
                }
        }

        public synchronized int fetch() {
                int item = 0;
                try {
                        while (size == 0) wait();
                        item = buf [head] ;
                        head = (head + 1) % max;
                        size--;
                        notify();
                }
                catch
                   (interruptedException e) {
                        // exception handler
                }
                return item;
        }
```

```
class Producer extends Thread {
        private Buffer buffer;


        public Producer (Buffer buf) {
                buffer = buf;
        }

        public void run () {
                int item;
                while (true) {
                    // create a new item
                    buffer.deposit(item);
                }
        }
}

class Consumer implements Runnable  {
        private Buffer buffer;

        public Consumer (Buffer buf) {
                buffer = buf;
        }

        public void run () {
                int item;
                while (true) {
                    item = buffer.fetch();
                    // consume item
                }
        }
}
```

We can now instantiate these threads as shown below:

```
Buffer buffer = new Buffer(100);
Producer producer1 = new Producer(buffer);
Consumer comsumer = new Consumer(buffer);
Thread consumer1 = new Thread(consumer);
producer1.start();
consumer1.start();
```

In this case the **buffer** is a shared variable used by both producer and consumer.

notify and notifyAll

There is a slight difference between **notify** and **notifyAll**. As the name suggest, **notify()** wakes up a single thread which is waiting on the object's lock. If there is more than one thread waiting, the choice is arbitrary i.e. there is no way to specify which waiting thread should be re-awakened. On the other hand, **notifyAll()** wakes up ALL waiting threads; the scheduler decides which one will run.

Applet

Java also has support for web applications through Applets. An applet is a stand alone Java *application.* Java *applet* runs in a Java-enabled Web browser.

Java versus C++

Generally, Java is more robust than C++. Some of the reasons are:
- Object handles are initialized to **null** (a keyword)
- Handles are always checked and exceptions are thrown for failures
- All array accesses are checked for bounds violations
- Automatic garbage collection prevents memory leaks and dangling pointers
- Type conversion is safer
- Clean, relatively fool-proof exception handling
- Simple language support for multithreading
- Bytecode verification of network applets