UNIT 2- JAVA

DATA TYPES:

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple types*. These can be put in four groups:

• Integers This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.

• Floating-point numbers This group includes **float** and **double**, which represent numbers with fractional precision

• Characters This group includes **char**, which represents symbols in a character set, like letters and numbers.

• Boolean This group includes **boolean**, which is a special type for representing true/false values.

INTEGER :

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positiveand negative values. Java does not support unsigned, positive-only integers.

Byte: The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.

Short: short is a signed 16-bit type. It has a range from –32,768 to 32,767.

Int: The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.

Long: long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.

Float: The type **float** specifies a *single-precision* value that uses 32 bits of storage

Name	Width in Bits	Approximate Range
Double	**64**	**4.9e–324 to 1.8e+308**
Float	**32**	**1.4e–045 to 3.4e+038**

Double: Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.

Characters

In Java, the data type used to store characters is **char**. Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**.

Type Conversion and Casting

it is fairly common to assign a value of one type to a variable of another type which is known as casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

• The two types are compatible.

• The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

if you want to assign an **int** value to a **byte** variable. This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion,* since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

```
int a;
byte b;
// ...
b = (byte) a;
```

Arrays: An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. Aspecific element in an array is accessed by its index. *type var-name*[ ];

An Example for Multidimensional Array.

While we define an multidimensional array it is necessary to define the number of row but not the column values.

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

		int twoD[][]= new int[4][];

// Multidimensional array each row has variable column values.
		twoD[0]=new int[1];
		twoD[1]=new int[2];
		twoD[2]=new int[3];
		twoD[3]=new int[4];

		int i,j,k=0;

		for(i=0;i<4;i++)
			for(j=0;j<i+1;j++){
				twoD[i][j]=k;
				k++;
			}
		for(i=0;i<4;i++){
			for(j=0;j<i+1;j++)
			System.out.print(twoD[i][j] + " ");

		System.out.println();
		}

		System.out.printf("the value of k is %d",k);
	}

}
```

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:

+	Addition
–	Subtraction
*	Multiplication
%	Modulus
/	Division
++	Increment
+=	Addition assignment

–=	Subtraction assignment
*=	Multiplication assignment
%=	Modulus assignment
/=	Division assignment
– –	Decrement

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

if (*condition*) *statement1*;

else *statement2*;

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. It looks like this:

if(*condition*)

statement;

else if(*condition*)

statement;

else if(*condition*)

statement;

...

else

statement;

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed.The final **else** acts as a default condition;

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

switch (*expression*) {

case *value1*:

// statement sequence

break;

case *value2*:

// statement sequence

```
break;
...
case valueN:
        // statement sequence
break;
default:
        // default statement sequence }
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. (An enumeration value can also be used to control a **switch** statement.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
        // body of loop  }
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

do-while

if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words,

there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.

The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
// body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

For Loop:

Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new "for-each" form. Both types of **for** loops are discussed here, beginning with the traditional form. Here is the general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
// body   }
```

If only one statement is being repeated, there is no need for the curly braces. The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

For-Each

Beginning with JDK 5, a second form of **for** was defined that implements a "for-each" style loop. As you may know, contemporary language theory has embraced the for-each concept, and it is quickly becoming a standard feature that programmers have come to expect. A foreach

style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained. Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays *type* must be compatible with the base type of the array.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

The General Form of a Class

Aclass is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;

        type methodname1(parameter-list) {
        // body of method      }

        type methodname2(parameter-list) {
        // body of method     }
        // ...
        type methodnameN(parameter-list) {
        // body of method     }   }
```

The data, or variables, defined within a **class** are called *instance variables.* The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Declaring Objects

Obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

First Method:

```
Box mybox = new Box();
```

Second Method:

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.

Constructor:

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

Example for Constructor:

```
class rect{
	int Ilength,Ibreadth;
	// Here rect() is an Constructor
	rect(){
		// here the member variables are initialized
		Ilength=10;
		Ibreadth=20;
		}
	int fun_area()
	{
	return Ilength*Ibreadth;
	}
}
public class cons {
		public static void main(String args[])
		{
			rect op=new rect();

			int Iarea;

			Iarea=op.fun_area();

			System.out.println("Area of rectangle is" + Iarea);
```

```
		}
	}
```

Parameterized Constructors:

We can pass the initialization values to the constructor.it is know as parameterized constructor.

```
import java.util.Scanner;

class cube{
	int ISide;
	cube(int x){
		// this is constructor
		ISide=x;
		}
	int fun_volume()
	{
	return ISide*ISide*ISide;
	}
}

public class paramconst {
	public static void main(String args[])
	{
		Scanner sr= new Scanner(System.in);

		System.out.println("Enter the side value");

		int side=sr.nextInt();

		// Parameterised Constructor
		cube op=new cube(side);

		int IVolume;

		IVolume=op.fun_volume();

		System.out.println("Volume of Cube is" + IVolume );
}
}
```

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked.

```
class box{
	double height;
	double depth;
```

```
        double width;
        // this operator & constructor when 3 dimensions are known
        box(double w, double d, double h){
                this.width=w;
                this.height=h;
                this.depth=d;
        }
        //default constructor
        box (){
                height=width=depth=2;
        }

        double fun_volume(){
                return width*height*depth;
        }
}

}


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

        double volume;

        box mybox1=new boxweight(10,10,10,10);

        volume=mybox1.fun_volume();
        System.out.println("volume of mybox1 object is " + volume);
        }
}
```

The finalize( ) Method:

Sometimes an object will need to perform some action when it is destroyed.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
        // finalization code here        }
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.

Overloading Methods :

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism.

Overloading Constructors: In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception**.**

```
class box2{
	double height;
	double depth;
	 double width;
	// this operator & constructor when 3 dimensions are known
	box2(double w, double d, double h){
		this.width=w;
		this.height=h;
		this.depth=d;
	}
	//default constructor
	box2(){
		height=width=depth=2;
	}

	double fun_volume_box(){
		return width*height*depth;
	}
}


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

		double volume;

		box2 mybox1=new box2(10,10,10);//calls parametrized constructor

		box2 mybox2=new mybox2();//calls default constructor

		volume=mybox1.fun_volume_box();
		System.out.println("volume of mybox1 object is " + volume);

		volume=mybox2.fun_volume_box();
		System.out.println("Volume of mybox2 obj is " + volume);
		}
}
```

We can pass the value to the member variable through the object. And also we can call the member function with the help of object. Here is an example for the above said statement.

```
class ree{
	int x;
	int square2(){
		return x*x;
		}
}
public class metho {
```

```
	public static void main(String args[])
	{
		int number,squaredvalue;
		Scanner sr= new Scanner(System.in);

		ree op=new ree();

		System.out.println("enter the number");
		number=sr.nextInt();

		op.x=number;

		squaredvalue=op.square2();

		System.out.println("squared value is" + squaredvalue);
}
}
```

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**.

Instance variables declared as **static** are, essentially, global variables.

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

```
// when the member is static it can be accessed
//before any object can be created
public class supercla {

	static int a=3;
	static int b;
	/*static method  access only static variable
	 * call ststic method.
	 * cant be used by this & super keyword*/
	static void meth(int x){
		System.out.println("X=" +x);
		System.out.println("a="+a);
		System.out.println("b="+b);
```

```
	}
	//Static block loaded exactly once when the
	//class is first loaded
	static{
		System.out.println("Static block");
		b=a*10;
	}
public static void main(String args[]){
	meth(50);
}
}
```

Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

Using super

There will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. **super** has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

```
class box2{
	// this is the base class , here its member variables are private.hece its
//visibility is with in this class alone.
	private double height;
	private double depth;
	private double width;

	//  use of this operator & this constructor will be called when 3 dimensions
//are known
	box2(double w, double d, double h){
		this.width=w;
		this.height=h;
		this.depth=d;
	}

//default constructor
	box2(){
		height=width=depth=2;
	}

	double fun_volume_box(){
		return width*height*depth;
```

```
	}
}

//the class "boxweight2" inherits the the properties of base class "BOX2" by the
//keyword extend

class boxweight2 extends box2{
	double weight;
	boxweight2(double w,double h,double d,double we){
		//here super keyword is used to access the private members of base class
		super(w,h,d);
		weight=we;
	}
//default constructor
boxweight2(){
		//here super() is used to call the default constructor in the base class
		super();
		weight=2;
	}
}

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

		double volume;

		boxweight2 mybox1=new boxweight2(10,10,10,10);

		boxweight2 mybox2=new boxweight2();

		volume=mybox1.fun_volume_box();
		System.out.println("volume of mybox1 object is " + volume);

		volume=mybox2.fun_volume_box();
		System.out.println("Volume of mybox2 obj is " + volume);
		}
}
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish

to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version.

```
class A{
	int i,j;
	A(int a, int b){
		i=a;
		j=b;
	}
	void show(){
		System.out.println("i & j values are " + i + " "+ j);
	}
	}
class B extends A{
	int k;
	B(int a, int b,int c){
		super(a,b);
		k=c;
	}
	void show(){
		//the following super.show()Is used to call the base class method.
		super.show();
		System.out.println("k value is  are " + k);
	}
	}

public class overrideclass2 {
 public static void main(String args[]){
	B suboj= new B(1,2,3);
	suboj.show();
 }
}
```

Dynamic Method Dispatch or Runtime Polymorphism:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

A superclass reference variable can refer to a subclass object.

Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this

determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

```
//method overriding
class shape{
	double dimension1;
	shape(double a){
		dimension1=a;
	}
	double volume(){
		System.out.println("volume for the shape is not defined");
		return 0;
	}
}

class sphere extends shape{
	sphere(double a){
		super(a);
	}
	double volume(){
		System.out.println("in the computation of volume of sphere");
		return (4/3)*Math.PI*Math.pow(dimension1, 3);
	}
}
class hemisphere extends shape{
	hemisphere(double a ){
		super(a);
	}
	double volume(){
		System.out.println("in the computation of volume of hemisphere");
		return (2.0/3.0)*Math.PI*Math.pow(dimension1, 3);
	}
}
public class overrideclas {
 public static void main(String args[]){
	shape f = new shape(2);
	sphere s=new sphere(3);
	hemisphere hs=new hemisphere(4);

	shape ref;

	ref=s;
	System.out.println("the volume is "+ ref.volume());

	ref=hs;
	System.out.println("the volume is "+ ref.volume());

	ref=f;
	System.out.println("the volume is "+ ref.volume());
 }

}
```

Abstract Class:

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclass responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

abstract *type name(parameter-list)*;

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. An abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

```
abstract class A5{
	abstract void callme();

	void callmetoo()
	{
		System.out.println("this is an example for abstract class ");
	}
}

class B5 extends A5{
	void callme(){
		System.out.println("this ia an implementation of abstract function");
	}
}

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

```
        B5 bobj=new B5();

        bobj.callme();
        bobj.callmetoo();
 } }
```

Uses of Final Keyword:

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too.

```
//  Named Constant
final int x=4;


//to prevent overriding
class A3{
        final void meth2() {
                System.out.println("This is final method");
        }
}

class B3 extends A3{
        void meth(){
                System.out.println("this is not possible");
        }
}
```

```
// to prevent inheritance
final class A4{
        ----------------------  }
class A4 extends B4{
        // this is also not possible
}
```

Package:

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package *pkg*;

Here name of the package.

INTERFACE

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

To implement an interface, a class must create the complete set of methods defined by the interface

This is the general form of an interface:

access interface *name* {

return-type method-name1(*parameter-list*)*;*

return-type method-name2(*parameter-list*)*;*
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(*parameter-list*)*;*
type final-varnameN = value; }

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

```
interface Callback {
    void callback(int param);
        }
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {
// class-body }

*When you implement an interface method, it must be declared as **public***

```
package day1;
```

```
interface callback{
	void callmenow();
	}

class client implements callback{
	public void callmenow(){
		System.out.println("this is an implementation of interface");
	}


}
public class interfaceexample {

	public static void main(String args[]){
		// creatinf an obj for interface through the method class
		//implements its method
		callback c=new client();
		c.callmenow();
	}
}
```

Multiple Inheritance In JAVA

```
interface callback1{
	void callmenow1();
	}

interface callback2{
	void callmenow2(int x);
	}

class client2 implements callback1,callback2{
	public void callmenow1(){
		System.out.println("this is an implementation of interface");
	}

	public void callmenow2(int x){
		System.out.println("Intrger passed in the second implementaion is" + x);
	}

}

public class multipleinheritance {
public static void main(String args[]){
	callback1 firstobj=new client2();
	callback2 secondobj=new client2();

	firstobj.callmenow1();
	secondobj.callmenow2(23);
}
}
```

EXCEPTIONS:

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

A general form of an exception-handling block:

```
try {
        // block of code to monitor for errors
        }

catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
        }
// ...
finally {
        // block of code to be executed after try block ends
        }
```

Here, *ExceptionType* is the type of exception that has occurred.

It is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause, or creating one with the **new** operator.

```
public class throwexample {
        static void add(){
                //int a=0,b=25;

                try{
                        throw new NullPointerException("demo");
                } catch(NullPointerException e){
                        System.out.println(" Caught inside demoproc() ");
                        throw e;
                }
        }

        public static void main(String args[])
        {
        try{
                add();
        }
        catch (NullPointerException e){
                System.out.println("Recaught"+e);
        }
        }
        }
```

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. A**throws** clause lists the types

of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause

type method-name(*parameter-list*) throws *exception-list*

{

// body of method

}

```
package day1;

public class throwsexceptionexample {

	static void throwone() throws IllegalAccessException {
		System.out.println("Inside throws");
		throw new IllegalAccessException("Demo");

	}

	public static void main(String args[])
	{
	try{
		throwone();
	}
	catch (IllegalAccessException e){
		System.out.println("caught"+e);
	}

}
}
```

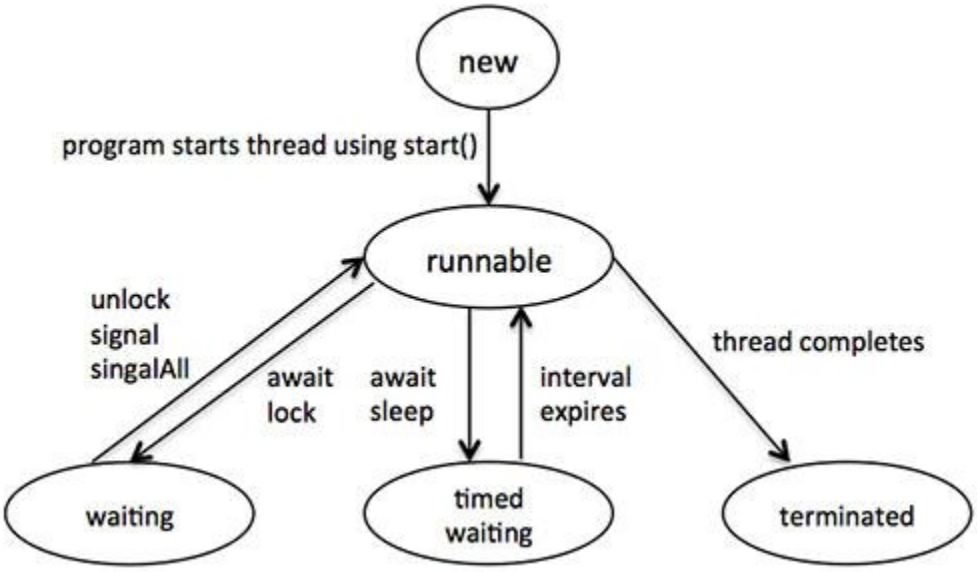

Multithreading

Java is a multithreaded programming language which means we can develop multi Threaded program using Java. A multi threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multi threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.

Multi threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:



A thread goes through various stages in its life cycle. For example, a thread is created, started, runs, and then destroyed. The above diagram shows complete life cycle of a thread.

Above- mentioned stages are explained here:

New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable: After a newly created thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting: Sometimes, a thread transitions to the Waiting state while the thread waits for another thread to perforMa task. A thread transitions back to the runnable state only when another thread signals the Waiting thread to continue executing.

Timed Waiting: A runnable thread can enter the Timed Waiting state for a specified interval of time. A thread, in this state transitions back to the runnable state when that time interval expires or when the event it is Waiting for occurs.

Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower- priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependant.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing Runnable interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax ofrun() method:

public void run( )

Step 2:

At second step you will instantiate a Thread object using the following constructor:

Thread(Runnable threadObj, String threadName);

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start( );

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
private Thread t;
private String threadName;

RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
 Thread.sleep(50);
}
```

```
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}

public void start () {
System.out.println("Starting " + threadName );
if(t== null)
{
t= new Thread (this, threadName);
t.start ();
} } }

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

RunnableDemo R1 = new RunnableDemo( "Thread-1"); R1.start();

RunnableDemo R2 = new RunnableDemo( "Thread-2"); R2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
```

Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting. Thread Thread-2 exiting.

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override run( ) method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

public void run( )

Step 2

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start( );

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {

private Thread t;
private String threadName;
```

```
ThreadDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}

public void start ()
{
System.out.println("Starting " + threadName );
if(t== null)
{
t= new Thread (this, threadName);
t.start ();
} } }

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

ThreadDemo T1 = new ThreadDemo( "Thread-1"); T1.start();
```

```
ThreadDemo T2 = new ThreadDemo( "Thread-2"); T2.start();
} }
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting. Thread Thread-2 exiting.
```

Thread Methods:

public void start(): Starts the thread in a separate path ofexecution, then invokes the run() method on this Thread object.

public void run(): If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

public final void setName(String name): Changes the name ofthe Thread object. There is also a getName() method for retrieving the name.

public final void setPriority(int priority): Sets the priority of this Thread object. The possible values are between 1 and 10.

public final void join(long millisec): The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

public void interrupt(): Interrupts this thread, causing it to continue execution ifit was blocked for any reason.

public final boolean isAlive(): Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

I/O Fundamentals

The Java language provides a simple model for input and output (I/O). All I/O is performed by writing to and reading from streams of data. The data may exist in a file or an array, be piped from another stream, or even come from a port on another computer. The flexibility of this model makes it a powerful abstraction of any required input and output.

The File Class

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

Creating a File Object

You create a File object by passing in a String that represents the name of a file, and possibly a Stringor another Fileobject. For example,

```
File a = new File("/usr/local/bin/IPLAB");
```

defines an abstract file name for the smurf file in directory /usr/local/bin. This is an *absolute* abstract file name. It gives *all* path information necessary to find the file.

You could also create a file object as follows:

```
File b = new File("bin/IPLAB");
```

This is a *relative* abstract file name, because it leaves out some necessary path information, which will be filled in by the VM. By default, the VM will use the directory in which the application was executed as the "current path".

File Attribute Methods

The Fileobject has several methods that provide information on the current state of the file.

boolean canRead()	Returns trueif the file is readable
Boolean canWrite()	Returns trueif the file is writeable
Boolean exists()	Returns trueif the file exists
boolean isAbsolute()	Returns trueif the file name is an *absolute* path name
boolean isDirectory()	Returns trueif the file name is a directory
boolean isFile()	Returns trueif the file name is a "normal" file (depends on OS)
boolean isHidden()	Returns trueif the file is marked "hidden"
long lastModified()	Returns a longindicating the last time the file was modified
long length()	Returns the length of the contents of the file

Text I/O Versus Binary I/O

Java's I/O classes are divided into two main groups, based on whether you want text or binary I/O. Readerand Writerclasses handle text I/O. InputStreamand OutputStream classes handle binary I/O.

Reader and InputStream

Java supplies Readers and InputStreams to read data; their use is similar. The following table shows the most commonly used methods in these classes. See the javadocs for the other methods available. Note that these two classes are *abstract*; you won't ever create an instance of either, but they provide the base implementation details for all other input classes.

Reader Class Methods:

void close()	Closes the input. Always call this method when you are finished reading . it allows the VM to release locks on the file.
int read()	Reads a single *item* from the file. In Reader, an *item* is a char, while in InputStreamit's a byte. The return value will either be the *item* or -1 if there is no more data
int read(***type***[])	Attempts to fill the array with as much data as possible. If enough data is available, the ***type***[] (char[]for Reader, byte[]for InputStream) will be filled with the data and the length of the array will be returned. If there's not enough data available, it will wait until the data is available or end-of-file is reached.
int read(***type***[], int offset, int length)	Similar to read(*datum[]*)but allows you to start at a specified offset in the input and read a limited number of bytes
int skip(int n)	Skips past the next *n* bytes or characters in the file, returning the actual number that were skipped. (If for example, end-of-file was reached, it might skip fewer than requested).

Writer and OutputStream

Java supplies Writerand OutputStream to write data; their use is similar. The following table shows the methods provided in these classes. Note that these two classes are *abstract*; you won't ever create an instance of either, but they provide the base implementation details for all other output classes.

void write(type[], int offset, int length)	Similar to write(type[]), but only *length* units of data will written from ***type***[], starting at the *offset*.
void write(int)	Writes a single item (charfor Writer, bytefor OutputStream) to the file.
void write(String) (Writeronly!)	Writes the contents of a java.lang.Stringto the file.
void write(String, int offset, int length) (Writer only!)	Writes the substring starting at *offset* and *length* characters lo to the file.

Reading and Writing Files

To read and write from files on a disk, use the following classes:

- FileInputStream
- FileOutputStream
- FileReader
- FileWriter

Each of these has a few constructors, where *class* is the name of one of the above classes:

•*class*(File) - create an input or output file based on the abstract path name passed in

•*class*(String)- create an input or output file based on the Stringpath name

•*class*(FileDescriptor)- create an input or output file based on a FileDescriptor (you generally won't use this and this class will not discuss it)

•*class*(String, boolean)- [**for output classes only**] create an input or output file based on the path name passed in, and if the boolean parameter is true, *append* to the file rather than *overwrite* it

For example, we could copy one file to another by using:

```
import java.io.*;

public class FileCopy {
```

```
public static void main(String args[]) {
try {
// Create an input file
FileInputStream inFile = new FileInputStream(args[0]);

// Create an output file
FileOutputStream outFile = new FileOutputStream(args[1]);

// Copy each byte from the input to output
 int byteRead;
while((byteRead = inFile.read()) != -1)
outFile.write(byteRead);

// Close the files!!!
inFile.close();
outFile.close();
}

// If something went wrong, report it!
catch(IOException e) {
System.err.println("Could not copy "+
args[0] + " to " + args[1]);
System.err.println("Reason:");
System.err.println(e);}
```

String:

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings:

The most direct way to create a string is to write **String greeting = "Hello world!";**

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

```
public class StringDemo{
public static void main(String args[]){
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
String helloString = new String(helloArray);
System.out.println( helloString );
}
}
```

String Length: Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object .

```
public class StringDemo {
public static void main(String args[]) {
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
System.out.println( "String Length is : " + len );
}
}
```

Concatenating Strings: The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat () method with string literals.

Some String Methods:

char charAt (int index): Returns the character at the specified index.

int compareTo(Object o): Compares this String to another Object .

int compareTo(String anotherString): Compares two strings lexicographically.

int compareToIgnoreCase(String str): Compares two strings lexicographically, ignoring case differences.

String concat (String str): Concatenates the specified string to the end of this string.

boolean equals(Object anObject): Compares this string to the specified object .

int indexOf (int ch): Returns the index within this string of the first occurrence of the specified character.

int length(): Returns the length of this string.

String replace(char oldChar, char newChar):Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

String[] split (String regex): Splits this string around matches of the given regular expression.

String toLowerCase(): Converts all of the characters in this String to lower case using the rules of the default locale.