

LECTURE NOTES

ON

Object Oriented Programming using JAVA

II B.TECH I SEMESTER

(JNTUA-R19)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA

Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112

(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

Object Oriented Programming using JAVA

UNIT I

UNIT-I

1

UNIT I

Introduction to Java: The key attributes of object oriented programming, Simple program, The Java keywords, Identifiers, Data types and operators, Program control statements, Arrays, Strings, String Handling

Introduction:

- JAVA is a programming language.
- Computer language innovation and development occurs for two fundamental reasons:
 - To adapt to changing environments and uses
 - To implement refinements and improvements in the art of programming
- Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++.
- Definition: Object-oriented programming (OOP) is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.
- Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. This language was initially called "Oak," but was renamed "Java" in 1995.
- Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems.
- Java contribution to internet:
Java programming had a profound effect on internet.

➤ **Java Applets**

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet

and automatically executed by a Java-compatible web browser.

➤ **Security**

Every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code.

In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

➤ **Portability**

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. Java programming provide portability

Byte code:

The output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. In essence, the original JVM was designed as an *interpreter for bytecode*.

➤ **Servlets: Java on the Server Side**

A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.

➤ **The Java Buzzwords**

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

➤ **Object-Oriented Programming**

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented.

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data.

Some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed.

The first way is called the *process-oriented model*. The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success.

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object oriented program can be characterized as *data controlling access to code*.

The key Attributes of OOP:

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

Encapsulation

- ✓ *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- ✓ In Java, the basis of encapsulation is the class.
- ✓ A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. Objects are sometimes referred to as *instances of a class*.
- ✓ Thus, a class is a logical construct; an object has physical reality.
- ✓ The code and data that constitute a class are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. The code that operates on that data is referred to as *member methods* or just *methods*.
- ✓ Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class.

Inheritance

- ✓ Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.
- ✓ Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization.
- ✓ A new subclass inherits all of the attributes of all of its ancestors.

Polymorphism

- ✓ *Polymorphism* (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions.
- ✓ More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*.

A First Simple Program

```
/*
```

```
    This is a simple Java program.
```

```
    Call this file Example.java.
```

```
*/
```

```
class Example {
```

```
    // A Java program begins with a call to main( ).
```

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

```
        System.out.println("Java drives the Web.");
```

```
    }
```

```
}
```

Entering the program:

The first step in creating a program is to enter its source code into the computer.

The name you give to a source file is very important. In Java, a source file is officially called a *compilation unit*. It is a text file that contains (among other things) one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension

The name of the main class should match the name of the file that holds the program.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. The output of **javac** is not code that can be directly executed.

Running the program

To actually run the program, you must use the Java application launcher called **java**.

To do so, pass the class name **Example** as a command-line argument, as shown here:

```
java Example
```

When the program is run, the following output is displayed:

```
Java drives the Web.
```

First simple program line by line

The program begins with the following lines:

This is a simple Java program. Call this file "Example.java".

```
*/
```

This is a *comment*. The contents of a comment are ignored by the compiler. This is multiline comment

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main( ).
```

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

This line begins the **main()** method. All Java applications begin execution by calling **main()**.

The **public** keyword is an *access modifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. **main()** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main()** to be called without having to instantiate a particular instance of the class. This is necessary since **main()** is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main()** does not return a value.

In **main()**, there is only one parameter, **String args[]** declares a parameter named **args**, which is an array of instances of the class **String**. Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

```
System.out.println("Java drives the Web.");
```

This line outputs the string "Java drives the Web." followed by a new line on the screen. Output is actually accomplished by the built-in **println()** method. In this case, **println()** displays the string which is passed to it. The line begins with **System.out**. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Example2:

/*

This demonstrates a variable.
Call this file Example2.java.

```
class Example2 {
    public static void main(String[] args) {
        int var1; // this declares a variable
        int var2; // this declares another variable
        var1 = 1024; // this assigns 1024 to var1
        System.out.println("var1 contains " + var1);
        var2 = var1 / 2;
        System.out.print("var2 contains var1 / 2: ");
        System.out.println(var2);
    }
}
```

O/P:

var1 contains 1024
var2 contains var1 / 2: 512

Example3:

This program illustrates the differences between int and double.
Call this file Example3.java.

```
Class Example3 {
    public static void main(String[] args) {
        int w; // this declares an int variable
        double x; // this declares a floating-point variable
        w = 10; // assign w the value 10

        x = 10.0; // assign x the value 10.0
        System.out.println("Original value of w: " + w);
        System.out.println("Original value of x: " + x);
        System.out.println(); // print a blank line
        // now, divide both by 4
        w = w / 4;
        x = x / 4;
        System.out.println("w after division: " + w);
        System.out.println("x after division: " + x);
    }
}
```

O/P

Original value of w: 10
Original value of x: 10.0

w after division: 2
x after division: 2.5

Example:

Try This 1-1

This program converts gallons to liters.

Call this program GalToLit.java.

```
class GalToLit {
    public static void main(String[] args) {
        double gallons; // holds the number of gallons
        double liters; // holds conversion to liters

        gallons = 10; // start with 10 gallons
        liters = gallons * 3.7854; // convert to liters
        System.out.println(gallons + " gallons is " + liters + " liters.");
    }
}
```

O/P:

10.0 gallons is 37.854 liters.

The Java Keywords

There are 50 keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.

These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.

The keywords **const** and **goto** are reserved but not used.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are GalToLit, Test, x, y2, maxLoad, my_var.

Invalid identifier names include these: 12x, not/ok.

The Primitive 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.

All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

Name	Width	Range
long	64	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	−2,147,483,648 to 2,147,483,647
short	16	−32,768 to 32,767
byte	8	−128 to 127

Compute distance light travels using long variables. import

java.util.Scanner;

class Light {

public static void main(String args[]) {

int days;

long lightspeed;

long seconds;

long distance;

// approximate speed of light in miles per

second lightspeed = 186000;

Scanner sc=new Scanner(System.in);

System.out.println("Enter number of days");

days=sc.nextInt();

seconds = days * 24 * 60 * 60; // convert to seconds

distance = lightspeed * seconds; // compute

distance System.out.print("In " + days);

System.out.print(" days light will travel about ");

System.out.println(distance + " miles.");

}

}

O/P:

Enter number of days 10

In 10 days light will travel about 160704000000 miles.

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

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

```
// Compute the area of a
circle. import
java.util.Scanner;
class Area {
    public static void main(String args[]) {
        double pi, r, a;
        Scanner input= new
        Scanner(System.in); pi = 3.1416; // pi,
        approximately System.out.println("Enter
        radius "); r=input.nextDouble();
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

O/P:

```
Enter radius 10.8
Area of circle is 366.436224
```

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

In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535. There are no negative **chars**.

```
// Character variables can be handled like integers.
class CharArithDemo {
    public static void main(String[] args) {
        char ch;
        ch = 'X';
        System.out.println("ch contains " + ch);
        ch++; // increment ch
        System.out.println("ch is now " + ch);
        ch = 90; // give ch the value Z
        System.out.println("ch is now " + ch);
    }
}
```

O/P:

```
ch contains X ch is
now Y ch is now Z
```

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

// Demonstrate boolean

```
values. class BoolDemo {  
    public static void main(String[] args) {  
        boolean b;  
        b = false;  
        System.out.println("b is " + b);  
        b = true;  
        System.out.println("b is " + b);  
        // a boolean value can control the if statement  
        if(b) System.out.println("This is executed."); b  
        = false;  
        if(b) System.out.println("This is not executed.");  
        // outcome of a relational operator is a boolean value  
        System.out.println("10 > 9 is " + (10 > 9));  
    }  
}
```

O/P:

b is false b is

true

This is executed.

10 > 9 is true

Literals are also commonly called constants

Java provides special escape sequences sometimes referred to as backslash character constants

Escape Sequence	Character
\n	newline
\t	tab
\b	backspace
\f	form feed
\r	return
\"	" (double quote)
\'	' (single quote)
\\	\ (back slash)
\uDDDD	character from the Unicode character set (DDDD is four hex digits)

Operators:

An operator is a symbol that tells the compiler to perform a specific mathematical, logical, or other manipulation. Java has four general classes of operators: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

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:

Operator	Meaning
+	Addition(also unary plus)
-	Subtraction(also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. We cannot use them on **boolean** types, but we can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

When the division operator is applied to an integer type, there will be no fractional component attached to the result. The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

// Demonstrate the % operator. class

```
ModDemo {
    public static void main(String[] args) {
        int iredult, irem;
        double dresult, drem;
        iredult = 10 / 3;
        irem = 10 % 3;
        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0;
        System.out.println("Result and remainder of 10 / 3: " + iredult + " " + irem);
        System.out.println("Result and remainder of 10.0 / 3.0: " + dresult + " " + drem);
    }
}
```

O/P:

Result and remainder of 10 / 3: 3 1

Result and remainder of 10.0 / 3.0: 3.3333333333333335 1.0

Increment and Decrement

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

`x = x + 1;`

can be rewritten like this by use of the increment operator: `x++;`

Similarly, this statement: `x = x - 1;`

is equivalent to `x--;`

Both increment and decrement operators can either prefix or postfix the operand.

There is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, there is an important difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified

EG:

`x=10;`

`y=++x;`

in this case y will be set to 11

`x=10;`

`y=x++;`

then y will be set to 10

// Demonstrate ++.

```
class IncDec {
    public static void main(String args[ ]) {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b;
        d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

The output of this program follows:

a = 2

b = 3

c = 4

d = 1

Relational and Logical Operators

The *relational operators* determine the relationship that one operand has to the other.

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value.

Java does not define true and false in the same way as C/C++. In C/ C++, true is any nonzero value and false is zero. In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators

Logical operators combine two **boolean** values to form a resultant **Boolean** value.

Operator	Meaning
&	AND
	OR
^	XOR
	Short-circuit OR
&&	Short-circuit AND
!	NOT

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **false == true**. The following table shows the effect of each logical operation:

A	B	A B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

7 Demonstrate the relational and logical operators. class

```
RelLogOps {
    public static void main(String [ ] args) {
        int i, j;
        boolean b1,
        b2; i = 10;
        j = 11;
        if(i < j)
            System.out.println("i < j");
        if(i <= j)
            System.out.println("i <= j");
        if(i != j)
            System.out.println("i != j");
        if(i == j)
            System.out.println("this won't execute");
        if(i >= j)
            System.out.println("this won't execute");
        if(i > j)
            System.out.println("this won't execute");
        b1 = true;
        b2 = false;
        if(b1 & b2)
            System.out.println("this won't execute");
        if(!(b1 & b2))
            System.out.println("!(b1 & b2) is true");
        if(b1 | b2)
            System.out.println("b1 | b2 is true");
        if(b1 ^ b2)
            System.out.println("b1 ^ b2 is true");
    }
}
```

O/P:

```
i < j
i <= j i != j
!(b1 & b2) is true b1 | b2
b1 is true
b1 ^ b2 is true
```


Short-circuit Logical operators:

These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators.

The difference between normal and short-circuit versions is that the normal operands will always evaluate each operand, but short-circuit versions will evaluate the second operand only when necessary.

When the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

```
// Demonstrate the short-circuit
operators. class SCops {
    public static void main(String[] args) {
        int n, d, q;
        n = 10;
        d = 2;
        if(d != 0 && (n % d) == 0)
            System.out.println(d + " is a factor of " + n); d
        = 0; // now, set d to zero
        // Since d is zero, the second operand is not
        evaluated. if(d != 0 && (n % d) == 0)
            System.out.println(d + " is a factor of " + n);
        /* Now, try same thing without short-circuit operator.
           This will cause a divide-by-zero error.
        */
        if(d != 0 & (n % d) == 0) System.out.println(d
            + " is a factor of " + n);
    }
}
```

Assignment operators:

The *assignment operator* is the single equal

sign, =. It has this general form: `var =`

expression;

Here, the type of *var* must be compatible with the type of *expression*.

`int x, y, z;`

`x = y = z = 100; // set x, y, and z to 100`

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement.

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

`a = a + 4;` can rewrite as: `a += 4;`

There are compound assignment operators for all of the arithmetic, binary operators.

Thus, any statement of the form `var = var op expression;` can be rewritten as `var op= expression;`

Operator Precedence

The following table shows the order of precedence for Java operators, from highest to lowest.

Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the `[]`, `()`, and `.` can also act like operators. In that capacity, they would have the highest precedence.

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	--	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
=	op=					
Lowest						

The Precedence of the Java Operators

Using Parentheses

Parentheses raise the precedence of the operations that are inside them.

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

The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation.

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Uppercase letters. class

```
UppCase {
    public static void main(String[] args) {
        char ch;
        for(int i=0; i < 10; i++) {
            ch = (char) ('a' + i);
            System.out.print(ch);
            // This statement turns off the 6th bit.
            ch = (char) ((int) ch & 65503); // ch is now uppercase
            System.out.print(ch + " ");
        }
    }
}
```

O/P:

a bB cC dD eE fF gG hH iI jJ

```

// Lowercase letters. class
LowerCase {
    public static void main(String[] args) {
        char ch;
        for(int i=0; i < 10; i++) {
            ch = (char) ('A' + i);
            System.out.print(ch);
            // This statement turns on the 6th bit.
            ch = (char) ((int) ch | 32); // ch is now lowercase
            System.out.print(ch + " ");
        }
    }
}
O/P:
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

```

The Left Shift

The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << *num* Here, *num* specifies the number of positions to left-shift the value in *value*.

The Right Shift

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times.

Its general form is shown here:

value >> *num* Here, *num* specifies the number of positions to right-shift the value in *value*.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then- else statements. This operator is the **?**.

The **?** has this general form:

expression1 ? *expression2* : *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If

expression1 is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.

```

import java.util.Scanner;
public class Largest
{
    public static void main(String[] args)
    {
        int a, b, c, d;
        Scanner s = new Scanner(System.in);
        System.out.println("Enter all three numbers:");
        a = s.nextInt( );
        b = s.nextInt( );
        c = s.nextInt( );
        d = a>b?(a>c?a:c):(b>c?b:c);
        System.out.println("Largest of "+a+", "+b+", "+c+" is: "+d);
    }
}

```

Control Statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **Selection, Iteration and Jump.**

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion.

Input characters from the keyboard

To read a character from keyboard we can use **System.in.read ()**. The **read()** waits until the user presses a key and then returns the result. The character returned as an integer, so it must be cast into a **char** to assign it to a char variable.

// Read a character from the keyboard.

```
class Kbln {
    public static void main(String[ ] args)
        throws java.io.IOException {
        char ch;
        System.out.print("Press a key followed by ENTER: ");
        ch = (char) System.in.read( ); // get a char
        System.out.println("Your key is: " + ch);
    }
}
```

O/P:

Press a key followed by ENTER: k Your
key is: k

The above program uses `throws java.io.IOException`. This line is necessary to handle input errors. It is a part of java exception handling mechanism.

if statement:

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;
```

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

```
// Guess the letter game.
class Guess {
    public static void main(String[] args)
        throws java.io.IOException {
        char ch, answer = 'K';
        System.out.println("I'm thinking of a letter between A and Z.");
        System.out.print("Can you guess it: ");
        ch = (char) System.in.read( ); // read a char from the
        keyboard if(ch == answer) System.out.println("*** Right ***");
    }
}
```

The next version uses **else** to print a message when the wrong letter is picked

```
// Guess the letter game, 2nd version. class
Guess2 {
    public static void main(String[] args)
        throws java.io.IOException {
        char ch, answer = 'K';
        System.out.println("I'm thinking of a letter between A and Z.");
        System.out.print("Can you guess it: ");
        ch = (char) System.in.read( ); // get a char
        if(ch == answer) System.out.println("*** Right ***");
        else System.out.println("...Sorry, you're wrong.");
    }
}
```

Nested ifs

A nested **if** is an **if** statement that is the target of another **if** or **else**. When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **ifs** 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;

```
// Demonstrate an if-else-if ladder.
class Ladder {
    public static void main(String[] args) {
        int x;
        for(x=0; x<6; x++) {
            if(x==1)
                System.out.println("x is one");
            else if(x==2)
                System.out.println("x is two");
            else if(x==3)
                System.out.println("x is three");
            else if(x==4)
                System.out.println("x is four");
            else
                System.out.println("x is not between 1 and 4");
        }
    }
}
```

O/P:

```
x is not between 1 and 4
x is one
x is two
x is three
x is four
x is not between 1 and 4
```

switch

The switch provides for a multi-way branch. 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:
}
```

Demonstrate the switch. class

```
switchDemo {  
    public static void main(String[ ] args) {  
        int i;  
        for(i=0; i<10; i++)  
            switch(i) {  
                case 0:  System.out.println("i is zero");  
                        break;  
                case 1:  System.out.println("i is one");  
                        break;  
                case 2:  System.out.println("i is two");  
                        break;  
                case 3:  System.out.println("i is three");  
                        break;  
                case 4:  System.out.println("i is four");  
                        break;  
                default:  System.out.println("i is five or more");  
            }  
        }  
    }  
}
```

The **break** statement is optional. If you omit the **break**, execution will continue on into the next

case.

Nested switch Statements

We can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```
switch(count) {  
    case 1:switch(target) { // nested switch  
        case 0: System.out.println("target is zero");  
                break;  
        case 1: // no conflicts with outer switch  
                System.out.println("target is one");  
                break;  
        }  
        break;  
    case 2: // ...
```


In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **ifs**.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**.

while. 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.

// Demonstrate the while loop. class

```
WhileDemo {
    public static void main(String[ ] args) {
        char ch;
        // print the alphabet using a while loop
        ch = 'a';
        while(ch <= 'z') {
            System.out.print(ch);
            ch++;
        }
    }
}
```

do-while

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);
```

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

for loop

The general form of the traditional for statement: *for(initialization; condition; iteration) {*

body

// Show square roots of 1 to 9.

```
class SqrRoot {
    public static void main(String[] args) {
        double num, sroot;
        for(num = 1.0; num < 10.0; num++) {
            sroot = Math.sqrt(num);
            System.out.println("Square root of " + num +
                               " is " + sroot);
        }
    }
}
```

Some variations on for loop:

- It is possible to declare the variable inside the initialization portion of the **for**.

```
// compute the sum and product of the numbers 1
// through 5 for(int i = 1; i <= 5; i++) {
    sum += i; // i is known throughout the
    loop product *= i;
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does.

- When using multiple loop control variables the initialization and iteration expressions for each variable are separated by commas.

```
for(i=0, j=10; i < j; i++, j--)
    System.out.println("i and j: " + i + " " +
                        j);
```

- It is possible for any or all of the initialization, condition, or iteration portions of the **for** loop to be blank.

```
i = 0; // move initialization out of
loop for( i < 10; ) {
    System.out.println("Pass #" + i);
    i++; // increment loop control var
}
```

NESTED LOOPS:

Java allows loops to be nested. That is, one loop may be inside another.

```
for(i=0; i<=5; i++) {
    for(j=1; j<=i; j++)
        System.out.print("*")
```

```
); System.out.println( );
```

```
}
```

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of goto.

Using break to Exit a Loop: By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

// Using break to exit a loop.

```
class BreakDemo {
    public static void main(String[] args) {
        int num;
        num = 100;
        // loop while i-squared is less than
        num for(int i=0; i < num; i++) {
            if(i*i >= num) break; // terminate loop if i*i >= 100
            System.out.print(i + " ");
        }
        System.out.println("Loop complete.");
    }
}
```

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop

Using break as a Form of Goto: The **break** statement can also be employed by itself to provide a “civilized” form of the goto statement.

The general form of the labeled **break** statement is shown

here: `break label;`

// Using break with a label.

```
class Break4 {
    public static void main(String[] args) {
        int i;
        for(i=1; i<4; i++) { one: {
            two: {
            three: {
                System.out.println("\ni is " + i);
                if(i==1) break one;
                if(i==2) break two; if(i==3)
                break three;
                // this is never reached
                System.out.println("won't print");
            }
            System.out.println("After block three.");
        }
        System.out.println("After block two.");
    }
    System.out.println("After block one.");
}
System.out.println("After for.");
}
```

Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using **continue**. The continue statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop.

```
// Use continue.
class ContDemo {
    public static void main(String[] args) {
        int i;
        // print even numbers between 0 and 100
        for(i = 0; i<=100; i++) {
            if((i%2) != 0) continue; // iterate
            System.out.println(i);
        }
    }
}
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue.

Here is an example program that uses **continue** to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String args[] ) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println( );
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
            System.out.println();
        }
    }
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

ARRAYS:

An *array* is a collection of variables of same type, referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables.

The general form to declare a one-dimensional array:

```
type[ ] array-name=new type[size];
```

Since arrays are implemented as objects, the creation of an array is a two-step process. First declare an array reference variable. Second allocate memory for the array, assigning the reference to that memory to the array. Thus arrays in java are dynamically allocated using **new** operator.

Eg : `int[] sample=new int[10];`

It is possible to break the above declaration.

```
int[ ] sample;
```

```
sample=new int[10];
```

// Demonstrate a one-dimensional array.

```
class ArrayDemo {
    public static void main(String[ ] args) {
        int[ ] sample = new int[10];
        int i;
        for(i = 0; i < 10; i = i+1)
            sample[i] = i;
        for(i = 0; i < 10; i = i+1)
            System.out.println("This is sample[" + i + "]: " + sample[i]);
    }
}
```

O/P:

```
This is sample[0]: 0 This is
sample[1]: 1 This is
sample[2]: 2 This is
sample[3]: 3 This is
sample[4]: 4 This is
sample[5]: 5 This is
sample[6]: 6 This is
sample[7]: 7 This is
sample[8]: 8 This is
sample[9]: 9
```

// Find the minimum and maximum values in an array.

```
class MinMax {
    public static void main(String[] args) {
        int[] nums = new int[10];
        int min, max;
        nums[0] = 99;
        nums[1] = -10;
        nums[2] = 100123;
        nums[3] = 18;
        nums[4] = -978;
        nums[5] = 5623;
        nums[6] = 463;
        nums[7] = -9;
        nums[8] = 287;
        nums[9] = 49;
        min = max = nums[0];
        for(int i=1; i < 10; i++) {
            if(nums[i] < min) min = nums[i];
            if(nums[i] > max) max = nums[i];
        }
        System.out.println("min and max: " + min + " " + max);
    }
}
```

// Use array initializers.

```
class MinMax2 {
    public static void main(String[] args) {
        int[] nums = { 99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49 };
        int min, max;
        min = max = nums[0];
        for(int i=1; i < 10; i++) {
            if(nums[i] < min) min = nums[i];
            if(nums[i] > max) max = nums[i];
        }
        System.out.println("Min and max: " + min + " " + max);
    }
}
```

Multidimensional Arrays:

Two-dimensional arrays:

A two dimensional array is a list of one-dimensional array. A two dimensional array can be thought of as creating a table of data organized by row and column. An individual item of data is accessed by specifying its row and column position.

To declare a two dimensional array, we must specify two dimensions.

```
int[ ][ ] table=new int[10][20];
```

// Demonstrate a two-dimensional array.

```
class TwoD {
    public static void main(String[ ] args) {
        int t, i;
        int[ ][ ] table = new int[3][4];
        for(t=0; t < 3; ++t) {
            for(i=0; i < 4; ++i) {
                table[t][i] = (t*4)+i+1;
                System.out.print(table[t][i] + " ");
            }
            System.out.println( );
        }
    }
}
```

O/P:

1	2	3	4
5	6	7	8
9	10	11	12

Irregular arrays:

When allocating memory for multi dimensional arrays we need to specify only the memory for the first dimension. We can allocate the remaining dimensions separately.

Manually allocate differing size second dimensions. class

```
Tagged {
    public static void main(String[ ] args) {
        int[ ][ ] riders = new int[7][ ];
        riders[0] = new int[10];
        riders[1] = new int[10];
        riders[2] = new int[10];
        riders[3] = new int[10];
        riders[4] = new int[10];
        riders[5] = new int[2];
        riders[6] = new int[2];
        int i, j;
```



```

// fabricate some
data for(i=0; i < 5;
i++) for(j=0; j < 10;
j++)
    riders[i][j] = i + j + 10;
for(i=5; i < 7; i++) for(j=0;
j < 2; j++)
    riders[i][j] = i + j + 10;
System.out.println("Riders per trip during the week:");
for(i=0; i < 5; i++) {
    for(j=0; j < 10; j++)
        System.out.print(riders[i][j] + " ");
    System.out.println( );
}
System.out.println( );
System.out.println("Riders per trip on the weekend:");
for(i=5; i < 7; i++) {
    for(j=0; j < 2; j++)
        System.out.print(riders[i][j] + " ");
    System.out.println( );
}
}
}

```

Initializing multi dimensional array:

A multidimensional array can be initialized by enclosing each dimension's initialize list within its own set of braces.

// Initialize a two-dimensional array.

```

class Squares {
    public static void main(String[ ] args) {
        int[ ][ ] sqrs = {
            { 1, 1 },
            { 2, 4 },
            { 3, 9 },
            { 4, 16 },
            { 5, 25 },
            { 6, 36 },
            { 7, 49 },
            { 8, 64 },
            { 9, 81 },
            { 10, 100 }
        };
        int i, j;
        for(i=0; i < 10; i++) {
            for(j=0; j < 2; j++)
                System.out.print(sqrs[i][j] + " ");
            System.out.println( );
        }
    }
}

```

Using the length member:

Because arrays are implemented as objects, each array has associated with it a **length** instance variable that contains the number of elements that the array can hold. In other words **length** contains the size of the array.

// Use the length array member.

```
class LengthDemo {  
    public static void main(String[ ] args) {  
        int[ ] list = new int[10];  
        int[ ] nums = { 1, 2, 3 };  
        int[ ][ ] table = { // a variable-length table  
            {1, 2, 3},  
            {4, 5},  
            {6, 7, 8, 9}  
        };  
        System.out.println("length of list is " + list.length);  
        System.out.println("length of nums is " + nums.length);  
        System.out.println("length of table is " + table.length);  
        System.out.println("length of table[0] is " + table[0].length);  
        System.out.println("length of table[1] is " + table[1].length);  
        System.out.println("length of table[2] is " + table[2].length);  
        System.out.println();  
        // use length to initialize list  
        for(int i=0; i < list.length; i++)  
            list[i] = i * i;  
        System.out.print("Here is list: ");  
        // now use length to display list  
        for(int i=0; i < list.length; i++)  
            System.out.print(list[i] + " ");  
        System.out.println( );  
    }  
}
```

The for-each style for loop:

A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

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:

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*.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.

EG: compute the sum of the values in an array:

```
Int[] nums= { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
int sum = 0;
```

```
for(int i=0; i < 10; i++) sum += nums[i];
```

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;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on.

// Use a for-each style for loop.

```
class ForEach {
    public static void main(String[] args) {
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;
        // Use for-each style for to display and sum the
        values. for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
        }
        System.out.println("Summation: " + sum);
    }
}
```

There is one important point to understand about the for-each style loop. Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array.

// The for-each loop is essentially read-only.

```
class NoChange {
    public static void main(String[] args) {
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        for(int x : nums) {
            System.out.print(x + " ");
            x = x * 10; // no effect on nums
        }
        System.out.println();
        for(int x : nums)
            System.out.print(x + " ");
        System.out.println( );
    }
}
```

2 3 4 5 6 7 8 9 10

2 3 4 5 6 7 8 9 10

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays.

Use for-each style for on a two-dimensional array. class

```
ForEach2 {
    public static void main(String[] args) {
        int sum = 0;
        int[][] nums = new int[3][5];
        // give nums some values
        for(int i = 0; i < 3; i++) for(int
        j=0; j < 5; j++)
            nums[i][j] = (i+1)*(j+1);
        // Use for-each for loop to display and sum the values.
        for(int[] x : nums) {
            for(int y : x) {
                System.out.println("Value is: " + y);
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

O/P:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

// Search an array using for-each style

for. import java.util.Scanner;

public class Search {

public static void main(String[] args) {

int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };

int val;

 Scanner a=**new** Scanner(System.in);

 System.out.println("Enter a number to search");

 val=a.nextInt();

boolean found = **false**;

 // Use for-each style for to search nums for val.

for(**int** x : nums)

 { **if**(x == val) {

 found = **true**;

break;

 }

 }

if(found)

 System.out.println("Value found!");

else

 System.out.println("Value not found!");

 }

}

O/P:

```
Enter a number to search 8
Value found!
```

```
Enter a number to search 10
Value not found!
```

STRINGS:

One of the most important data type in java is String. String defines and supports character sting. In java strings are objects

Constructing strings:

```
String str= new String("HELLO");
```

This creates a String object called str that contains the character string "HELLO".

A string can be constructed from another string

```
String str2= new String(str);
```

Another easy way to create a string is

```
String str="Java strings are powerful";
```

// Introduce String.

```
class StringDemo {
    public static void main(String[ ] args) {
        // declare strings in various ways
        String str1 = new String("Java strings are objects.");
        String str2 = "They are constructed various ways.";
        String str3 = new String(str2);
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

Operating on strings:

The **String** class contains several methods that operate on strings.

boolean equals(str)	Returns true if the invoking string contains the same character sequence as str
int length()	Returns the number of characters in the string
char charAt(index)	Returns the character at the index specified by index
int compareTo(str)	Returns less than zero if the invoking string is less than str, greater than zero if the the invoking string is greater than str, and zero if the strings are equal
int indexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure
int lastIndexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure

// Some String operations.

```
class StrOps {  
    public static void main(String[] args) {  
        String str1 = "When it comes to Web programming, Java is #1.";  
        String str2 = new String(str1);  
        String str3 = "Java strings are powerful.";  
        int result, idx;  
        char ch;  
        System.out.println("Length of str1: " + str1.length());  
        for(int i=0; i < str1.length(); i++)           // display str1, one char at a  
            time. System.out.print(str1.charAt(i));  
        System.out.println();  
        if(str1.equals(str2))  
            System.out.println("str1 equals str2");  
        else  
            System.out.println("str1 does not equal str2");  
        if(str1.equals(str3))  
            System.out.println("str1 equals str3");  
        else  
            System.out.println("str1 does not equal str3");  
        result = str1.compareTo(str3);  
        if(result == 0)  
            System.out.println("str1 and str3 are equal");  
        else if(result < 0)  
            System.out.println("str1 is less than str3");  
        else  
            System.out.println("str1 is greater than str3");  
        // assign a new string to str2  
        str2 = "One Two Three One";  
        idx = str2.indexOf("One");  
        System.out.println("Index of first occurrence of One: " + idx);  
        idx = str2.lastIndexOf("One");  
        System.out.println("Index of last occurrence of One: " + idx);  
    }  
}
```

Array of strings:

/ Demonstrate String arrays. class

```
StringArrays {  
    public static void main(String[] args) {  
        String[] strs = { "This", "is", "a", "test." };  
        System.out.println("Original array: ");  
        for(String s : strs)  
            System.out.print(s + " ");  
        System.out.println("\n");  
        // change a string in the  
        array strs[1] = "was";  
        strs[3] = "test, too!";  
        System.out.println("Modified array: ");  
        for(String s : strs)  
            System.out.print(s + " ");  
    }  
}
```

/ Use substring(). class

```
SubStr {  
    public static void main(String[] args) {  
        String orgstr = "Java makes the Web move.";  
        // construct a substring  
        String substr = orgstr.substring(5, 18);  
        System.out.println("orgstr: " + orgstr);  
        System.out.println("substr: " + substr);  
    }  
}
```


Use a string to control a switch statement. class

```
StringSwitch {  
    public static void main(String[ ] args) {  
        String command = "cancel";  
        switch(command) {  
            case "connect":    System.out.println("Connecting");  
                               // ...  
                               break;  
            case "cancel":    System.out.println("Canceling");  
                               // ...  
                               break;  
            case "disconnect": System.out.println("Disconnecting");  
                               // ...  
                               break;  
            default:          System.out.println("Command Error!");  
                               break;  
        }  
    }  
}
```

Using command line arguments:

Display all command-line information. class

```
CLDemo {  
    public static void main(String[ ] args) {  
        System.out.println("There are " + args.length + " command-line arguments.");  
        System.out.println("They are: ");  
        for(int i=0; i<args.length; i++)  
            System.out.println("arg[" + i + "]: " + args[i]);  
    }  
}
```

STRING HANDLING:

The **String** class is packaged in **java.lang**. Thus it is automatically available to all programs.

String objects can be constructed in a number of ways, making it easy to obtain a string when needed.

String Constructors:

// Demonstrate several String constructors.

```
class StringConsDemo {
    public static void main(String[] args) {
        char[] digits = new char[16];
        // Create an array that contains the digits 0 through 9
        // plus the hexadecimal values A through F.
        for(int i=0; i < 16; i++) {
            if(i < 10) digits[i] = (char) ('0'+i);
            else digits[i] = (char) ('A' + i - 10);
        }
        // Create a string that contains all of the array.
        String digitsStr = new String(digits);
        System.out.println(digitsStr);
        // Create a string the contains a portion of the array.
        String nineToTwelve = new String(digits, 9, 4);
        System.out.println(nineToTwelve);
        // Construct a string from a string.
        String digitsStr2 = new String(digitsStr);
        System.out.println(digitsStr2);
        // Now, create an empty string.
        String empty = new String();
        // This will display nothing:
        System.out.println("Empty string: " + empty);
    }
}
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result.

```
String age = "9";
String s = "He is " + age + " years
old."; System.out.println (s);
This displays the string "He is 9 years old."
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years
old."; System.out.println (s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object.

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the **+** to concatenate them. Here is an example:

```
// Using concatenation to prevent long
lines. class ConCat {
public static void main(String args[ ]) {
String longStr = "This could have been " +
"a very long line that would have " +
"wrapped around. But string concatenation " +
"prevents this.";
System.out.println(longStr);
}
}
```

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object

charAt()

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt()** method. It has this general form:

char charAt(int *index*)

// Demonstrate charAt() and length().

```
class CharAtAndLength {
public static void main(String[] args) {
String str = "Programming is both art and science.";
// Cycle through all characters in the string.
for(int i=0; i < str.length(); i++)
System.out.print(str.charAt(i) + " ");
System.out.println();
}
}
```

getChars()

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target[]*, int *targetStart*)

// getChars()

```
class GetCharsDemo {
public static void main(String[] args) {
String str = "Programming is both art and science.";
int start = 15;
int end = 23;
char[ ] buf = new char[end - start];
str.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings.

equals() and **equalsIgnoreCase()**

To compare two strings for equality, use **equals()**. It has this general form:

```
boolean equals(Object str)
```

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase()**.

When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true**

if the strings contain the same characters in the same order, and **false** otherwise.

// Demonstrate equals() and equalsIgnoreCase().

```
class EqualityDemo {
    public static void main(String[] args) {
        String str1 = "table";
        String str2 = "table";
        String str3 = "chair";
        String str4 = "TABLE";
        if(str1.equals(str2))
            System.out.println(str1 + " equals " + str2);
        else
            System.out.println(str1 + " does not equal " + str2);
        if(str1.equals(str3))
            System.out.println(str1 + " equals " + str3);
        else
            System.out.println(str1 + " does not equal " + str3);
        if(str1.equals(str4))
            System.out.println(str1 + " equals " + str4);
        else
            System.out.println(str1 + " does not equal " + str4);
        if(str1.equalsIgnoreCase(str4))
            System.out.println("Ignoring case differences, " + str1 + " equals " + str4);
        else
            System.out.println(str1 + " does not equal " + str4);
    }
}
```

O/P:

```
table equals table
table does not equal chair
table does not equal TABLE
Ignoring case differences, table equals TABLE
```

equals() Versus ==

It is important to understand that the equals() method and the == operator perform two different operations. The equals() method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance.

Equals() vs ==

```
class EqualsNotEqualTo {
    public static void main(String args[ ]) {
        String s1 = "Hello";
        String s2 = new String(s1);
        System.out.println(s1 + " equals " + s2 + " -> " +
            s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

The variable s1 refers to the String instance created by "Hello". The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

regionMatches()

The **regionMatches()** method compares a specific region inside a string with another specific region in another string. Here are the general forms for two methods:

```
boolean regionMatches(int startIndex, String str2,
                      int str2StartIndex, int
numChars) boolean regionMatches(boolean
ignoreCase,
                              int startIndex, String str2,
                              int str2StartIndex, int numChars)
```

// Demonstrate

RegionMatches. class

```
CompareRegions {
    public static void main(String[] args) {
        String str1 = "Standing at river's edge.";
        String str2 = "Running at river's edge.";
        if(str1.regionMatches(9, str2, 8, 12))
            System.out.println("Regions match.");
        if(!str1.regionMatches(0, str2, 0, 12))
            System.out.println("Regions do not match.");
    }
}
```

O/P:

Regions match. Regions do
not match.

startsWith() and endsWith()

The **startsWith()** method determines whether a given **String** begins with a specified string. Conversely, **endsWith()** determines whether the **String** in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str)
```

```
boolean endsWith(String str)
```

A second form of **startsWith()**, shown here, lets you specify a starting

```
point: boolean startsWith(String str, int startIndex)
```

compareTo() and compareToIgnoreCase()

It has this general form:

```
int compareTo(String str)
```

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than str
Greater than zero	The invoking string is greater than str
Zero	The two strings are equal.

If you want to ignore case differences when comparing two strings, use

compareToIgnoreCase(), as shown here:

```
int compareToIgnoreCase(String str)
```

substring()

You can extract a substring using **substring()**. It has two forms. The first

```
is String substring(int startIndex)
```

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring()** allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex)
```

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point

replace()

The **replace()** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```
String replace(char original, char replacement)
```

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

The second form of **replace()** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

trim()

The **trim()** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim()

Eg: String str=" gamma ";

After str=str.trim();

Str will contain only the string"gamma"

Changing the Case of Characters Within a String

The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase. The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

String toLowerCase()
String toUpperCase()

// Demonstrate toUpperCase() and toLowerCase().

```
class ChangeCase {
    public static void main(String[] args)
    {
        String str = "This is a test.";
        System.out.println("Original: " + str);
        String upper = str.toUpperCase(); String
        lower = str.toLowerCase();
        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}
```

O/P:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

String represents fixed-length, immutable character sequences.

In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.

StringBuilder is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance.

UNIT II

Classes: Classes, Objects, Methods, Parameters, Constructors, Garbage Collection, Access modifiers, Pass Objects and arguments, Method and Constructor Overloading, Understanding static, Nested and inner classes.

Inheritance – Basics, Member Access, Usage of Super, Multi level hierarchy, Method overriding, Abstract class, Final keyword.

Interfaces –Creating, Implementing, Using, Extending, and Nesting of interfaces.

Packages – Defining, Finding, Member Access, Importing

CLASSES AND OBJECTS:

- ✓ A class is a *template* for an object, and an object is an *instance* of a class.

The General Form of a Class

- ✓ When a class is defined, its exact form and nature is declared by specifying the data that it contains and the code that operates on that data.
- ✓ A class is declared by use of the **class** keyword.

```
class class-name{
    // declare instance variables
    type var1;
    type var2;
    //.....
    type varN;
    // declare methods
    type method1( parameters){
        //body of method
    }
    type method2( parameters){
        //body of method
    }
    //.....
    type methodN( parameters){
        //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.
- ✓ Each instance of the class (that is, each object of the class) contains its own copy of these variables

Defining a class:

```
class Vehicle {
    int passengers; // number of passengers
    int fuelCap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon
}
```

class declaration is only a type description: it does not create an actual object To actually create a **Vehicle** object, use a statement like the following

Vehicle minivan = new Vehicle(); // create a vehicle object called minivan

After this statement executes, **minivan** will be an instance of **Vehicle**. Thus, it will have “physical” reality.

- ✓ Thus, every **Vehicle** object will contain its own copies of the instance variables passengers, fuelCap, mpg. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable.

object.member;

minivan.fuelcap=16;

/* A program that uses the Vehicle class.

Call this file VehicleDemo.java

***/**

```
class Vehicle {  
    int passengers; // number of passengers  
    int fuelCap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
}
```

// This class declares an object of type Vehicle.

```
class VehicleDemo {  
    public static void main(String[ ] args) {  
        Vehicle minivan = new Vehicle();  
        int range;  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelCap = 16;  
        minivan.mpg = 21;  
        // compute the range assuming a full tank of gas  
        range = minivan.fuelCap * minivan.mpg;  
        System.out.println("Minivan can carry " + minivan.passengers +  
            "with a range of " + range);  
    }  
}
```

- ✓ When this program is compiled, two **.class** files have been created. The Java compiler automatically puts each class into its own **.class** file.
- ✓ To run this program we must run VehicleDemo.class.

// This program creates two Vehicle objects.

```
class Vehicle {  
    int passengers; // number of passengers  
    int fuelCap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
}
```

// This class declares an object of type Vehicle.

```
class TwoVehicles {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportscar = new Vehicle();  
        int range1, range2;  
  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelCap = 16;  
        minivan.mpg = 21;  
  
        // assign values to fields in sportscar  
        sportscar.passengers = 2;  
        sportscar.fuelCap = 14;  
        sportscar.mpg = 12;  
  
        // compute the ranges assuming a full tank of gas  
        range1 = minivan.fuelCap * minivan.mpg;  
        range2 = sportscar.fuelCap * sportscar.mpg;  
        System.out.println("Minivan can carry " + minivan.passengers + " with a range of " + range1);  
        System.out.println("Sportscar can carry " + sportscar.passengers + " with a range of " + range2);  
    }  
}
```

}

}

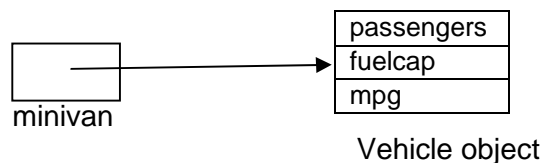
DECLARING OBJECTS

- ✓ 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. This can be done by 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.
- ✓ This reference is, more or less, the address in memory of the object allocated by **new**.

Vehicle minivan; // declare reference to object

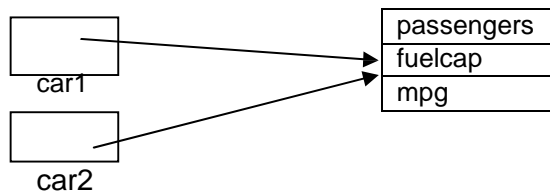


minivan = new Vehicle(); // allocate a Vehicle object



REFERENCE VARIABLES AND ASSIGNMENTS

- ✓ Object reference variables act differently when an assignment takes place.
- ✓ Consider the following fragment:
Vehicle car1= new Vehicle();
Vehicle car2= car1;



After this fragment executes, **car1** and **car2** will both refer to the *same* object. The assignment of **car1** to **car2** did not allocate any memory or copy any part of the original object. It simply makes **car2** refer to the same object as does **car1**. Thus, any changes made to the object through **car2** will affect the object to which **car1** is referring, since they are the same object.

car1.mpg=26;

- ✓ When the following statements are executed display the same value 26
System.out.println(car1.mpg);

System.out.println(car2.mpg);

- ✓ Although **car1** and **car2** both refer to the same object, they are not linked in any other way. Vehicle
car1= new Vehicle();

Vehicle car2= car1;

Vehicle car3= new Vehicle();

car2=car3;

- ✓ After this statement executes **car2** refers to the same object as **car3**. The object referred to by **car1** is exchanged.

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Methods

- ✓ A method contains the statements that define its actions. This is the general form of a method:

```
type name(parameter-list) {  
    // body of method
```

```
}
```

- ✓ Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**.
- ✓ The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Adding a Method to the Vehicle Class

- ✓ Most of the time, methods are used to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions.

// Add range to Vehicle.

```
class Vehicle {  
    int passengers; // number of passengers  
    int fuelCap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
    // Display the range.  
    void range( ) {  
        System.out.println("Range is " + fuelCap * mpg);  
    }  
}  
  
class AddMeth {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportscar = new Vehicle();  
        int range1, range2;  
  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelCap = 16;  
        minivan.mpg = 21;  
  
        // assign values to fields in sportscar  
        sportscar.passengers = 2;  
        sportscar.fuelCap = 14;  
        sportscar.mpg = 12;  
        System.out.print("Minivan can carry " + minivan.passengers + ". ");  
        minivan.range( ); // display range of minivan  
        System.out.print("Sportscar can carry " + sportscar.passengers + ". ");  
        sportscar.range( ); // display range of sportscar.  
    }  
}
```

- ✓ When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call.
- ✓ When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.

Returning from a method:

In general, two conditions can cause a method to return- first, when the method's closing brace is encountered. The second is when a return statement is executed. There are two forms of return – one for use in void methods and one for returning values.

Returning a value:

- ✓ Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

// Use a return value.

```
class Vehicle {
    int passengers; // number of passengers
    int fuelCap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon
    // Return the range.
    int range( ) {
        return mpg * fuelCap;
    }
}

class RetMeth {
    public static void main(String[ ] args) {
        Vehicle minivan = new Vehicle( );
        Vehicle sportscar = new Vehicle( );
        int range1, range2;
        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelCap = 16;
        minivan.mpg = 21;
        // assign values to fields in sportscar
        sportscar.passengers = 2;
        sportscar.fuelCap = 14;
        sportscar.mpg = 12;
        // get the ranges
        range1 = minivan.range();
        range2 =
        sportscar.range();
        System.out.println("Minivan can carry " + minivan.passengers + "with range of " + range1 + " miles");
        System.out.println("Sportscar can carry " + sportscar.passengers + "with range of " + range2 + " miles");
    }
}
```

Using parameters:

- ✓ It is possible to pass one or more values to a method when the method is called.
- ✓ Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.
- ✓ There are two important things to understand about returning values:
 - The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
 - The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

// A simple example that uses a parameter.

```
class ChkNum {
    // Return true if x is even.
    boolean isEven(int x) {
        if((x% 2) == 0) return true;
        else return false;
    }
}

class ParmDemo {
    public static void main(String[ ] args) {
        ChkNum e = new ChkNum( );
        if(e.isEven(10)) System.out.println("10 is
        even."); if(e.isEven(9)) System.out.println("9 is
        even."); if(e.isEven(8)) System.out.println("8 is
        even.");
    }
}
```

} }

A method can have more than one parameter. Simply declare each parameter, separating one from the next with a comma.

```
class Factor {
    // Return true if a is a factor of b.
    boolean isFactor(int a, int b) {
        if( (b % a) == 0) return true;
        else return false;
    }
}

class IsFact {
    public static void main(String[] args) {
        Factor x = new Factor( );
        if(x.isFactor(2, 20)) System.out.println("2 is factor");
        if(x.isFactor(3, 20)) System.out.println("this won't be displayed");
    }
}
```

Adding a parameterized method to Vehicle:

```
/*
    Add a parameterized method that computes the    fuel required for a given distance.
*/
class Vehicle {
    int passengers; // number of passengers
    int fuelCap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon
    // Return the range.
    int range( ) {
        return mpg * fuelCap;
    }
    // Compute fuel needed for a given distance.
    double fuelNeeded(int miles) {
        return (double) miles / mpg;
    }
}

class CompFuel {
    public static void main(String[ ] args) {
        Vehicle minivan = new Vehicle( );
        Vehicle sportscar = new Vehicle( );
        double gallons;
        int dist = 252;
        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelCap = 16;
        minivan.mpg = 21;
        // assign values to fields in sportscar
        sportscar.passengers = 2;
        sportscar.fuelCap = 14;
        sportscar.mpg = 12;
        gallons = minivan.fuelNeeded(dist);
        System.out.println("To go " + dist + " miles minivan needs " + gallons + " gallons of fuel.");
        gallons = sportscar.fuelNeeded(dist);
        System.out.println("To go " + dist + " miles sportscar needs " + gallons + " gallons of fuel.");
    }
}
```


CONSTRUCTORS:

- ✓ Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a 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.

// A simple constructor.

```
class MyClass
{ int x;
  MyClass( ) {
    x = 10;
  }
}

class ConsDemo {
  public static void main(String[ ] args) {
    MyClass t1 = new MyClass( );
    MyClass t2 = new MyClass( );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

Parameterized Constructors

- ✓ Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parenthesis after the constructor's name.

// A parameterized constructor.

```
class MyClass {
  int x;
  MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  public static void main(String[ ] args) {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);
    System.out.println(t1.x + " " + t2.x);
  }
}
```

Adding a Constructor to a Vehicle class

// Add a constructor.

```
class Vehicle {
  int passengers; // number of passengers
  int fuelCap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
  // This is a constructor for Vehicle.
  Vehicle(int p, int f, int m) {
    passengers =
    p; fuelCap = f;
    mpg = m;
  }
  // Return the range.
  int range( ) {
    return mpg * fuelCap;
  }
  // Compute fuel needed for a given distance.
  double fuelNeeded(int miles) {
    return (double) miles / mpg;
  }
}
```

}

```

class VehConsDemo {
    public static void main(String[] args) {
        // construct complete vehicles
        Vehicle minivan = new Vehicle(7, 16, 21);
        Vehicle sportscar = new Vehicle(2, 14,
        12); double gallons;
        int dist = 252;
        gallons = minivan.fuelNeeded(dist);
        System.out.println("To go " + dist + " miles minivan needs " +
            gallons + " gallons of fuel.");
        gallons = sportscar.fuelNeeded(dist);
        System.out.println("To go " + dist + " miles sportscar needs " +
            gallons + " gallons of fuel.");
    }
}

```

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 MyClass {
    int x;
    MyClass( int i) {
        this.x = i;
    }
}

class ConsDemo {
    public static void main(String[] args) {
        MyClass t1 = new MyClass(10);
        MyClass t2 = new MyClass(88);
        System.out.println(t1.x + " " + t2.x);
    }
}

```

Output of the above code is 10 88

- ✓ **this** has some important uses. For example java syntax permits the name of a parameter or a local variable to be the same of an instance variable. When this happens, the local name hides the instance variable. The hidden instance variable can gain access by referring to it through **this**.

```

class MyClass {
    int x;
    MyClass( int x) {
        x = x;
    }
}

class ConsDemo {
    public static void main(String[] args) {
        MyClass t1 = new MyClass(10);
        MyClass t2 = new MyClass(88);
        System.out.println(t1.x + " " + t2.x);
    }
}

```

Output is 0 0

If we use this key word we can gain access to the hidden instance variables

```
class MyClass {  
    int x;  
    MyClass( int x) {  
        this.x = x;  
    }  
}
```

```
class ConsDemo {  
    public static void main(String[] args) {  
        MyClass t1 = new MyClass(10);  
        MyClass t2 = new MyClass(88);  
        System.out.println(t1.x + " " + t2.x);  
    }  
}
```

O/P is 10 88

new OPERATOR REVISITED

- ✓ When you allocate an object, you use the following general form:
`class-var = new classname ();`
- ✓ Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called.
- ✓ When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- ✓ The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones.
- ✓ Once you define your own constructor, the default constructor is no longer used.

Garbage Collection

- ✓ Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.
- ✓ Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called *garbage collection*.
- ✓ When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- ✓ Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize() Method

- ✓ Sometimes an object will need to perform some action when it is destroyed.
- ✓ To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- ✓ To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method, you will specify those actions that must be performed before an object 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.

Access Modifiers:

- ✓ Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*
- ✓ How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers.
- ✓ Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.
- ✓ When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- ✓ When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- ✓ An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;  
private double j;  
private int myMethod(int a, char b) { //...
```

- ✓ To understand the effects of public and private access, consider the following program:

/* This program demonstrates the difference between public and private. */

```
class Test {  
    int a; // default access public  
    int b; // public access  
    private int c; // private access  
    // methods to access c  
    void setc(int i) { // set c's value  
        c = i;  
    }  
    int getc() { // get c's value  
        return c;  
    }  
}  
  
class AccessTest {  
    public static void main(String args[ ]) {  
        Test ob = new Test();  
        // These are OK, a and b may be accessed directly  
        ob.a = 10;  
        ob.b = 20;  
        // This is not OK and will cause an error  
        // ob.c = 100; // Error!  
        // You must access c through its methods  
        ob.setc(100); // OK  
        System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());  
    }  
}
```

Pass objects to methods:

- ✓ It is possible to pass objects to a methods

// Objects can be passed to methods.

```
class Block {
    int a, b, c;
    int volume;
    Block(int i, int j, int k) {
        a = i;
        b = j;
        c = k;
        volume = a * b * c;
    }
    // Return true if ob defines same block.
    boolean sameBlock(Block ob) {
        if((ob.a == a) & (ob.b == b) & (ob.c == c)) return
        true; else return false;
    }
    // Return true if ob has same volume.
    boolean sameVolume(Block ob) {
        if(ob.volume == volume) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String[] args) {
        Block ob1 = new Block(10, 2, 5);
        Block ob2 = new Block(10, 2, 5);
        Block ob3 = new Block(4, 5, 5);
        System.out.println("ob1 same dimensions as ob2: " + ob1.sameBlock(ob2));
        System.out.println("ob1 same dimensions as ob3: " + ob1.sameBlock(ob3));
        System.out.println("ob1 same volume as ob3: " + ob1.sameVolume(ob3));
    }
}
```

How Arguments are passed:

- ✓ There are two ways to pass an argument to a subroutine.
- ✓ The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- ✓ The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
- ✓ When you pass a primitive type to a method, it is passed by value.

// Primitive types are passed by value.

```
class Test {  
    /* This method causes no change to the arguments  
       used in the call. */  
    void noChange(int i, int j) {  
        i = i + j;  
        j = -j;  
    }  
}  
  
class CallByValue {  
    public static void main(String[] args) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
        System.out.println("a and b before call: " + a + " " + b);  
        ob.noChange(a, b);  
        System.out.println("a and b after call: " + a + " " + b);  
    }  
}
```

- ✓ When you pass an object to a method, objects are passed by call-by-reference.

// Objects are passed through their references.

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    /* Pass an object. Now, ob.a and ob.b in object  
       used in the call will be changed. */  
    void change(Test ob)  
    { ob.a = ob.a + ob.b;  
      ob.b = -ob.b;  
    }  
}  
  
class PassObjRef {  
    public static void main(String[] args) {  
        Test ob = new Test(15, 20);  
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);  
        ob.change(ob);  
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);  
    }  
}
```

Returning objects:

✓ A method can return any type of data, including class types that you create.

// Return a String object.

```
class ErrorMsg {
    String[] msgs = { "Output Error", "Input Error", "Disk Full", "Index Out-Of-Bounds"};
    // Return the error message.
    String getErrorMsg(int i) {
        if(i >=0 & i < msgs.length)
            return msgs[i];
        else
            return "Invalid Error Code";
    }
}

class ErrMsgDemo {
    public static void main(String[] args) {
        ErrorMsg err = new ErrorMsg();
        System.out.println(err.getErrorMsg(2));
        System.out.println(err.getErrorMsg(19));
    }
}
```

O/P:

Disk Full

Invalid Error Code

✓ We can also return objects of classes that we create.

// Return a programmer-defined object. class

```
Err {
    String msg; // error message
    int severity; // code indicating severity of error
    Err(String m, int s) {
        msg = m;
        severity = s;
    }
}

class ErrorInfo {
    String[] msgs = {
        "Output Error",
        "Input Error",
        "Disk Full",
        "Index Out-Of-Bounds"
    };
    int[] howbad = { 3, 3, 2, 4 };
    Err getErrorInfo(int i) {
        if(i >= 0 & i < msgs.length)
            return new Err(msgs[i], howbad[i]);
        else
            return new Err("Invalid Error Code", 0);
    }
}

class ErrInfoDemo {
    public static void main(String[] args) {
        ErrorInfo err = new ErrorInfo( );
        Err e;
        e = err.getErrorInfo(2);
        System.out.println(e.msg + " severity: " + e.severity);
        e = err.getErrorInfo(19);
        System.out.println(e.msg + " severity: " + e.severity);
    }
}
```


}

METHOD OVERLOADING:

- ✓ 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.
- ✓ Overloaded methods must differ in the type and/or number of their parameters.
- ✓ While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- ✓ When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

// **Demonstrate method overloading. class**

```
Overload {  
    void          ovIDemo()          {  
        System.out.println("No parameters");  
    }  
    // Overload ovIDemo for one integer parameter.  
    void ovIDemo(int a) {  
        System.out.println("One parameter: " + a);  
    }  
    // Overload ovIDemo for two integer parameters.  
    int ovIDemo(int a, int b) {  
        System.out.println("Two parameters: " + a + " " + b);  
        return a + b;  
    }  
    // Overload ovIDemo for two double parameters.  
    double ovIDemo(double a, double b) {  
        System.out.println("Two double parameters: " + a + " " + b);  
        return a + b;  
    }  
}  
  
class OverloadDemo {  
    public static void main(String[] args) {  
        Overload ob = new Overload();  
        int resI;  
        double resD;  
        // call all versions of ovIDemo()  
        ob.ovIDemo();  
        System.out.println();  
        ob.ovIDemo(2);  
        System.out.println();  
        resI = ob.ovIDemo(4, 6);  
        System.out.println("Result of ob.ovIDemo(4, 6): "  
        +resI); System.out.println();  
        resD = ob.ovIDemo(1.1, 2.32);  
        System.out.println("Result of ob.ovIDemo(1.1, 2.32): " +  
            resD);  
    }  
}
```

O/P:

No parameters

One
parameter: 2

Two parameters: 4 6

Result of ob.ovIDemo(4, 6): 10

Two double parameters: 1.1 2.32

Result of ob.ovlDemo(1.1, 2.32): 3.42

- ✓ The difference in their return types is insufficient for the purpose of overloading.

// one ovlDemo(int a) is ok

```
void ovlDemo(int a) {  
    System.out.println("One parameter: " + a);  
}  
// Error. two ovlDemo(int a) are not ok even though their return types are different  
int ovlDemo(int a) {  
    System.out.println("One parameter: " + a);  
    return a * a;  
}
```

- ✓ Java provides certain automatic type conversions. These conversions also apply to parameters of overloaded methods. For example consider the following:

/* Automatic type conversions can affect overloaded method resolution. */

```
class Overload2  
{ void f(int x) {  
    System.out.println("Inside f(int): " + x);  
}  
  
void f(double x) {  
    System.out.println("Inside f(double): " + x);  
}  
}  
  
class TypeConv {  
    public static void main(String[] args) {  
        Overload2 ob = new Overload2();  
        int i = 10;  
        double d = 10.1;  
  
        byte b = 99;  
        short s = 10;  
        float f = 11.5F;  
  
        ob.f(i); // calls ob.f(int)  
        ob.f(d); // calls ob.f(double)  
  
        ob.f(b); // calls ob.f(int) - type conversion  
        ob.f(s); // calls ob.f(int) - type conversion  
        ob.f(f); // calls ob.f(double) - type conversion  
    }  
}
```

O/P

```
Inside f(int) : 10  
Inside f(double) : 10.1  
Inside f(int) : 99  
Inside f(int) : 10  
Inside f(double) : 11.5
```

In the case of byte and short java automatically converts them to int. In the case of float the value is converted to double and f(double) is called.

The automatic type conversions apply only if there is no direct match between a parameter and an argument.

OVERLOADING CONSTRUCTORS:

✓ Like methods constructors can also be overloaded. This allows to construct objects in a variety of ways.

// Demonstrate an overloaded constructor.

class

```
MyClass{ int  
x; MyClass()  
{  
    System.out.println("Inside MyClass().");  
    x = 0;  
}  
MyClass(int i) {  
    System.out.println("Inside MyClass(int).");  
    x = i;  
}  
MyClass(double d) {  
    System.out.println("Inside MyClass(double).");  
    x = (int) d;  
}  
MyClass(int i, int j) { System.out.println("Inside  
    MyClass(int, int)."); x = i * j;  
}  
}
```

class OverloadConsDemo {

```
    public static void main(String[] args) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass(88);  
        MyClass t3 = new MyClass(17.23);  
        MyClass t4 = new MyClass(2, 4);  
        System.out.println("t1.x: " + t1.x);  
        System.out.println("t2.x: " + t2.x);  
        System.out.println("t3.x: " + t3.x);  
        System.out.println("t4.x: " + t4.x);  
    }  
}
```

O/P:

```
Inside MyClass(). Inside  
MyClass(int).  
Inside MyClass(double).  
Inside MyClass(int, int). t1.x: 0  
t2.x: 88  
t3.x: 17  
t4.x: 8
```

}

// Initialize one object with another.

```
class Summation {  
    int sum;  
    // Construct from an int.  
    Summation(int num) {  
        sum = 0;  
        for(int i=1; i <= num; i++)  
            sum += i;  
    }  
    // Construct from another object.  
    Summation(Summation ob) {  
        sum = ob.sum;  
    }  
}
```

}

class SumDemo {

```
    public static void main(String[] args) {  
        Summation s1 = new Summation(5);  
        Summation s2 = new Summation(s1);  
        System.out.println("s1.sum: " + s1.sum);  
        System.out.println("s2.sum: " + s2.sum);  
    }  
}
```

}

O/P:

s1.sum: 15
s2.sum: 15

UNDERSTANDING static:

- ✓ 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**.
- ✓ When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- ✓ You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

static variables:

- ✓ Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

// Use a static variable.

```
class StaticDemo {
    int x; // a normal instance variable
    static int y; // a static variable
    // Return the sum of the instance variable x and the static variable y.
    int sum() {
        return x + y;
    }
}

class SDemo {
    public static void main(String[] args) {
        StaticDemo ob1 = new StaticDemo();
        StaticDemo ob2 = new StaticDemo();
        // Each object has its own copy of an instance variable.
        ob1.x = 10;
        ob2.x = 20;
        System.out.println("ob1.x: " + ob1.x + "\nob2.x: " + ob2.x);
        System.out.println();
        StaticDemo.y = 19;
        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();
        StaticDemo.y = 100;
        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();
    }
}
```

O/P:

ob1.x: 10
ob2.x: 20

ob1.sum(): 29
ob2.sum(): 39

ob1.sum(): 110
ob2.sum(): 120

static Methods:

- ✓ Methods declared static are, essentially, global methods. They are called independently of any object. Instead a static method is called through its class name.
- ✓ Methods declared as **static** have several restrictions:
 - They can only directly call other **static** methods.
 - They can only directly access **static** data.
 - They cannot refer to **this** or **super** in any way.

// Use a static method. class

```
StaticMeth {  
    static int val = 1024; // a static variable  
    // A static method.  
    static int valDiv2() {  
        return val/2;  
    }  
}  
  
class SDemo2 {  
    public static void main(String[] args) {  
        System.out.println("val is " + StaticMeth.val);  
        System.out.println("StaticMeth.valDiv2(): " +StaticMeth.valDiv2());  
        StaticMeth.val = 4;  
        System.out.println("val is " + StaticMeth.val);  
        System.out.println("StaticMeth.valDiv2(): " + StaticMeth.valDiv2());  
    }  
}
```

O/P:

val is 1024

StaticMeth.valDiv2(): 512

val is 4

StaticMeth.valDiv2(): 2

static Blocks:

- ✓ A static block is executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose.

// Use a static block

```
class StaticBlock { static  
    double rootOf2; static  
    double rootOf3;  
static {  
    System.out.println("Inside static block.");  
    rootOf2 = Math.sqrt(2.0);  
    rootOf3 = Math.sqrt(3.0);  
}  
    StaticBlock(String msg) {  
        System.out.println(msg);  
    }  
}  
  
class SDemo3 {  
    public static void main(String[] args) {  
        StaticBlock ob = new StaticBlock("Inside Constructor");  
        System.out.println("Square root of 2 is " +StaticBlock.rootOf2);  
        System.out.println("Square root of 3 is " +StaticBlock.rootOf3);  
    }  
}
```

O/P:

Inside static block.

Inside Constructor

Square root of 2 is 1.4142135623730951

Square root of 3 is 1.7320508075688772

}

NESTED AND INNER CLASSES:

- ✓ It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
- ✓ A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- ✓ A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.
- ✓ There are two types of nested classes: *static* and *non-static*.
- ✓ A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.
- ✓ The most important type of nested class is the *inner* class. An inner class is a non-static nested class.

// Use an inner class. class

```
Outer {
    int[]    nums;
    Outer(int[] n) {
        nums = n;
    }
    void analyze() {
        Inner inOb = new Inner();
        System.out.println("Minimum: " + inOb.min());
        System.out.println("Maximum: " + inOb.max());
        System.out.println("Average: " + inOb.avg());
    }
    // This is an inner class.
    class Inner {
        // Return the minimum value.
        int min() {
            int m = nums[0];
            for(int i=1; i < nums.length; i++)
                if(nums[i] < m) m = nums[i];
            return m;
        }
        // Return the maximum value.
        int max() {
            int m = nums[0];
            for(int i=1; i < nums.length; i++)
                if(nums[i] > m) m = nums[i];
            return m;
        }
        // Return the average.
        int avg() {
            int a = 0;
            for(int i=0; i < nums.length; i++)
                a += nums[i];
            return a / nums.length;
        }
    }
}

class NestedClassDemo {
    public static void main(String[] args) {
        int[] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
        Outer outOb = new Outer(x);
        outOb.analyze();
    }
}
```

O/P: Minimum: 1 Maximum: 9 Average: 5

INHERITANCE:

Basics:

- ✓ Java supports inheritance by allowing one class to incorporate another class into its declaration. This is done by using **extends** keyword.
- ✓ In java a class that is inherited is called a superclass. The class that does the inheriting is called a subclass.
- ✓ The general form of a **class** declaration that inherits a superclass is shown here: class
subclass-name extends superclass-name {
 // body of class
}
- ✓ We can only specify one superclass for any subclass that we create. Java does not support the inheritance of multiple superclasses into a single subclass.
- ✓ A major advantage of inheritance is that once we had created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- ✓ The following program creates a superclass called TwoDShape and subclass called Triangle

// A class for two-dimensional objects. class

```
TwoDShape {  
    double width;  
    double height;  
    void showDim()  
    {  
        System.out.println("Width and height are " + width + " and " + height);  
    }  
}
```

// A subclass of TwoDShape for triangles.

```
class Triangle extends TwoDShape  
{ String style;  
    double area() {  
        return width * height / 2;  
    }  
    void showStyle() {  
        System.out.println("Triangle is " + style);  
    }  
}
```

```
class Shapes {  
    public static void main(String[] args) {  
        Triangle t1 = new Triangle();  
        Triangle t2 = new  
        Triangle(); t1.width = 4.0;  
        t1.height = 4.0;  
        t1.style = "filled";  
        t2.width = 8.0;  
        t2.height = 12.0;  
        t2.style = "outlined";  
        System.out.println("Info for t1: ");  
        t1.showStyle();  
        t1.showDim();  
        System.out.println("Area is " + t1.area());  
        System.out.println();  
        System.out.println("Info for t2: ");  
        t2.showStyle();  
        t2.showDim();  
        System.out.println("Area is " + t2.area());  
    }  
}
```

O/P:

Info for t1:
Triangle is filled
Width and height are 4.0 and 4.0
Area is 8.0

Info for t2:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

}
}

MEMBER ACCESS AND INHERITANCE:

- ✓ Inheriting a class does not overrule the private access restriction. Thus even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

// Private members of a superclass are not accessible by a subclass.

// This example will not compile.

// A class for two-dimensional objects. class

```
TwoDShape {
    private double width; // these are
    private double height; // now private
    void showDim() {
        System.out.println("Width and height are " + width + " and " + height);
    }
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    String style;
    double area()
    {
        return width * height / 2; // Error! can't access
    }
    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

- ✓ The Triangle class will not compile because the reference to width and height inside the area() method causes an access violation. Since width and height are declared private in TwoDShape, they are accessible only by the other members of TwoDShape.
- ✓ To access private members of superclass we can use accessor methods.

USAGE OF super:

- ✓ Both the superclass and subclass have their own constructors.
- ✓ Constructors for the superclass construct the superclass portion of the object, and the constructor for the subclass constructs the subclass part.
- ✓ When both the superclass and the subclass define constructors, the process is a bit complicated because both the superclass and subclass constructors must be executed. In this case we need to use **super** keyword.
- ✓ **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.

Using super to Call Superclass Constructors

- ✓ A subclass can call a constructor defined by its superclass by use of the following form of

super:

```
super(arg-list);
```

- ✓ **super()** must always be the first statement executed inside a subclass constructor.

// Add constructors to TwoDShape.

```
class TwoDShape {
    private double width;
    private double height;
    // Parameterized constructor.
    TwoDShape(double w, double h)
    {
        width = w;    height = h;
    }
    // Accessor methods for width and height.
    double getWidth() { return width; }
```

```
double getHeight() { return height; }  
void setWidth(double w) { width = w; }  
void setHeight(double h) { height = h;  
}
```

```

        void showDim() {
            System.out.println("Width and height are " + width + " and " + height);
        }
    }
    // A subclass of TwoDShape for triangles. class
    Triangle extends TwoDShape {
        private String style;
        Triangle(String s, double w, double h) {
            super(w, h); // call superclass constructor
            style = s;
        }
        double area() {
            return getWidth() * getHeight() / 2;
        }
        void showStyle() {
            System.out.println("Triangle is " + style);
        }
    }
    class Shapes4 {
        public static void main(String[] args) {
            Triangle t1 = new Triangle("filled", 4.0, 4.0);
            Triangle t2 = new Triangle("outlined", 8.0, 12.0);
            System.out.println("Info for t1: "); t1.showStyle();
            t1.showDim();
            System.out.println("Area is " + t1.area());
            System.out.println();
            System.out.println("Info for t2: ");
            t2.showStyle();
            t2.showDim();
            System.out.println("Area is " + t2.area());
        }
    }
}

```

Using super to Access Superclass Members:

- ✓ The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super. member

- ✓ Here, *member* can be either a method or an instance variable

// Using super to overcome name hiding.

```

class A
{ int i;
}
// Create a subclass by extending class A. class
B extends A {
    int i; // this i hides the i in A
    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }
    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper {
    public static void main(String[] args) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}

```

}

Creating a Multilevel Hierarchy

- ✓ We can build hierarchies that contain as many layers of inheritance. It is perfectly acceptable to use a subclass as a superclass of another.
- ✓ For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.
- ✓ To see how a multilevel hierarchy can be useful, consider the following program.

```
// A multilevel hierarchy. class
TwoDShape {
    private double width;
    private double height;
    // A default constructor.
    TwoDShape() {
        width = height = 0.0;
    }
    // Parameterized constructor.
    TwoDShape(double w, double h)
    {
        width = w;
        height = h;
    }
    // Construct object with equal width and height.
    TwoDShape(double x) {
        width = height = x;
    }
    // Accessor methods for width and height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
    void showDim() {
        System.out.println("Width and height are " +
            width + " and " + height);
    }
}

// Extend TwoDShape.
class Triangle extends TwoDShape
{ private String style;
  // A default constructor.
  Triangle() {
      super();
      style = "none";
  }
  Triangle(String s, double w, double h) {
      super(w, h); // call superclass constructor
      style = s;
  }
  // One argument constructor.
  Triangle(double x) {
      super(x); // call superclass constructor
      // default style to filled
      style = "filled";
  }
  double area() {
      return getWidth() * getHeight() / 2;
  }
  void showStyle() {
```

```
        System.out.println("Triangle is " + style);  
    }  
}
```

```

// Extend Triangle.
class ColorTriangle extends Triangle
{ private String color;
  ColorTriangle(String c, String s, double w, double h) {
    super(s, w, h);
    color = c;
  }
  String getColor() { return color; }
  void showColor() {
    System.out.println("Color is " + color);
  }
}
class Shapes6 {
  public static void main(String[] args) {
    ColorTriangle t1 =new ColorTriangle("Blue", "outlined", 8.0, 12.0);
    ColorTriangle t2 =new ColorTriangle("Red", "filled", 2.0, 2.0);
    System.out.println("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    System.out.println("Area is " + t1.area());
    System.out.println();
    System.out.println("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    System.out.println("Area is " + t2.area());
  }
}

```

When Constructors Are Called

- ✓ When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?
- ✓ In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- ✓ Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed.
- ✓ The following program illustrates when constructors are executed:

// Demonstrate when constructors are executed. class

```

A {
  A() {
    System.out.println("Constructing A.");
  }
}

class B extends A
{ B() {
  System.out.println("Constructing B.");
}
}

class C extends B
{ C() {
  System.out.println("Constructing C.");
}
}

class OrderOfConstruction {
  public static void main(String[] args) {
    C c = new C();
  }
}

```

}

Superclass references and subclass objects:

- ✓ A reference variable for one class type cannot normally refer to an object of another class type.

// This will not compile. class

```
X {
    int a;
    X(int i) { a = i; }
}

class Y
{ int a;
  Y(int i) { a = i; }
}

class IncompatibleRef {
    public static void main(String[] args) {
        X x = new X(10);
        X x2;
        Y y = new Y(5);
        x2 = x; // OK, both of same type
        x2 = y; // Error, not of same type
    }
}
```

- ✓ A reference variable of a superclass can be assigned a reference to an object of any subclass derived from that superclass.

// A superclass reference can refer to a subclass object. class

```
X {
    int a;
    X(int i) { a = i; }
}

class Y extends X
{ int b;
  Y(int i, int j) {
      super(j);
      b = i;
  }
}

class SupSubRef {
    public static void main(String[] args) {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);
        x2 = x; // OK, both of same type
        System.out.println("x2.a: " + x2.a);
        x2 = y; // still OK because y is derived from X
        System.out.println("x2.a: " + x2.a);
        // X references know only about X members
        x2.a = 19; // OK
        // x2.b = 27; // Error, X doesn't have a b member
    }
}
```

- ✓ When a reference to a subclass object is assigned to a superclass reference variable we can only access to those parts of the object defined by the superclass.
- ✓ When constructors are called in a class hierarchy, subclass references can be assigned to a superclass variable.

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 its 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.
- ✓ Consider the following:

```
class A1{
    int i, j;
    A1(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show(){
        System.out.println("i and j: " + i + " " + j);
    }
}

class B1 extends A1 {
    int k;
    B1(int a, int b, int c) {
        super(a,
            b); k = c;
    }
    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class MetOver{
    public static void main(String[] args) {
        B1 subOb = new B1(1,2,3);
        subOb.show(); // this calls show() in B
    }
}
```

O/P:
k: 3

- ✓ When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used.
- ✓ If you wish to access the superclass version of an overridden method, you can do so by using **super**.

```
class B extends A
{
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    void show() {
        super.show(); // this calls A's show()
        System.out.println("k: " + k);
    }
}
```

O/P:
i and j: 1 2
k: 3

- ✓ Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

/* Methods with differing signatures are overloaded and not overridden. */ class A

```
{
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

// Create a subclass by extending class A. class

```
B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // overload show() void
    show(String msg) {
        System.out.println(msg + k);
    }
}
```

```
class Overload {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);
        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

O/P:

this is k: 3 i and

j: 1 2

DYNAMIC METHOD DISPATCH:

- ✓ Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. 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.
- ✓ 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. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

// Demonstrate dynamic method dispatch.

```
class Sup {
    void who() {
        System.out.println("who() in Sup");
    }
}

class Sub1 extends Sup {
    void who() {
        System.out.println("who() in Sub1");
    }
}
```

}


```

class Sub2 extends Sup {
    void who() {
        System.out.println("who() in Sub2");
    }
}

class DynDispDemo {
    public static void main(String[] args) {
        Sup superOb = new Sup();
        Sub1 subOb1 = new Sub1();
        Sub2 subOb2 = new Sub2();
        Sup supRef;
        supRef =
        superOb;
        supRef.who();
        supRef = subOb1;
        supRef.who();
        supRef = subOb2;
        supRef.who();
    }
}

```

O/P:

who() in Sup who()

in Sub1 who() in

Sub2

Why Overridden Methods?

- ✓ Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- ✓ Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

USING ABSTRACT CLASSES:

- ✓ A class which contains the **abstract** keyword in its declaration is known as abstract class.
 - ✓ Abstract classes may or may not contain *abstract methods* ie., methods without body (public void get();)
 - ✓ But, if a class has at least one abstract method, then the class **must** be declared abstract.
 - ✓ If a class is declared abstract it cannot be instantiated.
 - ✓ To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.
 - ✓ If you inherit an abstract class you have to provide implementations to all the abstract methods in it.

Abstract Methods:

- ✓ If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
 - ✓ **abstract** keyword is used to declare the method as abstract.
 - ✓ You have to place the **abstract** keyword before the method name in the method declaration.
 - ✓ An abstract method contains a method signature, but no method body.
 - ✓ Instead of curly braces an abstract method will have a semi colon (;) at the end.
- ✓
- ✓

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}
class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

Using final

- ✓ To prevent a method from being overridden or a class from being inherited by using the keyword **final**.

final prevents overriding:

- ✓ To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Using final to Prevent Inheritance

- ✓ To prevent a class from being inherited, precede the class declaration with **final**.
- ✓ Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- ✓ It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
    // ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    // ...
}
```

final with data members:

- ✓ **final** can also be applied to member variables . If a class variable's name precede with **final**, its value cannot be changed throughout the lifetime of the program.

INTERFACES:

- ✓ Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- ✓ Java Interface also **represents IS-A relationship**.
- ✓ Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- ✓ Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- ✓ To implement an **interface**, a class must create the complete set of methods defined by the interface.
- ✓ By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- ✓ Interfaces are designed to support dynamic method resolution at run time.
- ✓ **Defining an Interface**
- ✓ An interface is defined much like a class. This is a simplified general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    //...  
    return-type method-nameN(parameter-list);  
}
```
- ✓ The java compiler adds **public** and **abstract** keywords before the interface method and **public**, **static** and **final** keywords before data members.
- ✓ An interface is different from a class in several ways, including:
 - You cannot instantiate an interface.
 - An interface does not contain any constructors.
 - All of the methods in an interface are abstract.
 - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
 - An interface is not extended by a class; it is implemented by a class.
 - An interface can extend multiple interfaces.

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 is:

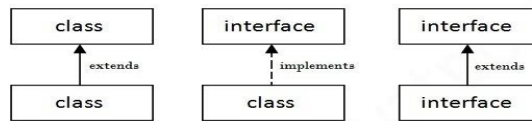
```
class classname extends superclass implements interface {  
    // class-body  
}
```
- ✓ If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**.
- ✓ Differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can have static methods, main method and constructor .	Interface can't have static methods, main method or constructor .
5) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
6) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
7) Example: public abstract class Shape{ public abstract void draw(); }	Example: public interface Drawable{ void draw(); }

- ✓ Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Understanding relationship between classes and interfaces

- ✓ As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



```

public interface Series {
    int getNext(); // return next number in series
    void reset(); // restart
    void setStart(int x); // set starting value
}
// Implement Series.
class ByTwos implements Series {
    int start;
    int val;
    ByTwos()
    {
        start = 0;
        val = 0;
    }
    // Implement the methods specified by Series.
    public int getNext()
    { val += 2;
      return val;
    }
    public void reset()
    { val = start;
    }
    public void setStart(int x) {
        start = x;
        val = x;
    }
}
class SeriesDemo {
    public static void main(String[] args) {
        ByTwos ob = new ByTwos();
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " +
            ob.getNext()); System.out.println("\nResetting");
        ob.reset();
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " +
            ob.getNext()); System.out.println("\nStarting at
            100"); ob.setStart(100);
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " + ob.getNext());
    }
}
  
```

O/P:

Next value is 2 Next
value is 4 Next value
is 6 Next value is 8

Next value is 10

Resetting
Next value is 2 Next
value is 4 Next value
is 6 Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110

- ✓ The classes that implement an interface not only limited to those methods in an interface. The class can provide whatever additional functionality is desired.
- ✓ Any number of classes can implement an interface.

```
// Implement Series a different way.
class ByThrees implements Series {
    int start; int
    val;
    ByThrees()
    {
        start = 0;
        val = 0;
    }
    // Implement the methods specified by Series.
    public int getNext() {
        val += 3;
        return val;
    }
    public void reset() {
        val = start;
    }
    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

Using interface reference:

- ✓ An interface declaration creates a new reference type. When a class implements an interface, it is adding that interface's type to its type.
- ✓ Interface reference variable can refer to any object that implements the interface.

```
class SeriesDemo2 {
    public static void main(String[] args) {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new
        ByThrees();

        Series iRef; // an interface reference

        for(int i=0; i < 5; i++) {
            iRef = twoOb; // refers to a ByTwos object
            System.out.println("Next ByTwos value is " + iRef.getNext());
            iRef = threeOb; // refers to a ByThrees object
            System.out.println("Next ByThrees value is " +
            iRef.getNext());
        }
    }
}
```

```
}  
}  
}
```

Implementing multiple interfaces:

- ✓ A class can implement more than one interface.
- ✓ Multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

```
interface IfA {
    void doSomething();
}
interface IfB {
    void doSomethingElse();
}
// Implement both IfA and IfB. class
MyClass implements IfA, IfB {
    public void doSomething() {
        System.out.println("Doing something.");
    }
    public void doSomethingElse() {
        System.out.println("Doing something else.");
    }
}
```

- ✓ If a class implements two interfaces that declare the same method, then the same method implementation will be used for both interfaces. This means that only one version of the method is defined by the class.

// Both IfA and IfB declare the method doSomething().

```
interface IfA {
    void doSomething();
}
interface IfB {
    void doSomething();
}
// Implement both IfA and IfB class
MyClass implements IfA, IfB {
    // This method implements both IfA and IfB.
    public void doSomething() {
        System.out.println("Doing something.");
    }
}
class MultiImpDemo {
    public static void main(String[] args) {
        IfA aRef;
        IfB bRef;
        MyClass obj = new MyClass();

        // Both interfaces use the same doSomething().
        aRef = obj;
        aRef.doSomething()
        ; bRef = obj;
        bRef.doSomething()
        ;
    }
}
```

Constants in Interfaces:

- ✓ The primary purpose of an interface is to declare methods that provide a well defined interface to functionality. An interface can also include variables, but these are not instance variables instead they are implicitly **public, final, static** and must be initialized.
- ✓ To define a set of shared constants, simply create an interface that contains only those constants without any methods. Each class that needs to access the constants simply "implements" the interface.


```
// An interface that contains constants.
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Boundary Error";
}

// Gain access to the constants by implementing IConst. class
IConstDemo implements IConst {
    public static void main(String[] args) {
        int[] nums = new int[MAX];
        for(int i=MIN; i < (MAX + 1); i++)
        {
            if(i >= MAX)
                System.out.println(ERRORMSG); else
            {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}
```

INTERFACES CAN BE EXTENDED:

- ✓ One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes.
- ✓ When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B inherits meth1() and meth2() - it adds meth3().
interface B extends A {
    void meth3();
}

// This class must implement all of A and B. class
MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }
    public void meth2() {
        System.out.println("Implement meth2().");
    }
    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String[] args) {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}

O/P:
Implement    meth1().
```

Implement meth2().
Implement meth3().

Nested Interfaces

- ✓ An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- ✓ A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level.
- ✓ When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

// A nested interface example.

// This interface contains a nested interface.

```
interface A {  
    // this is a nested interface  
    public interface NestedIF {  
        boolean isNotNegative(int x);  
    }  
    void doSomething();  
}
```

// This class implements the nested interface.

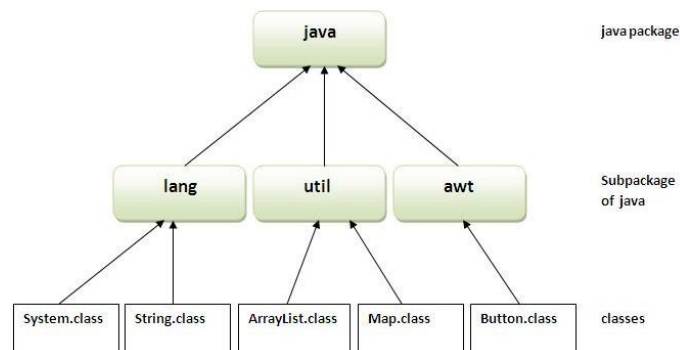
```
class B implements A.NestedIF {  
    public boolean isNotNegative(int x)  
    { return x < 0 ? false: true;  
    }  
}  
class NestedIFDemo {  
    public static void main(String[] args) {  
        // use a nested interface reference  
        A.NestedIF nif = new B( );  
        if(nif.isNotNegative(10))  
            System.out.println("10 is not negative");  
        if(nif.isNotNegative(-12))  
            System.out.println("this won't be displayed");  
    }  
}
```

PACKAGE

- ✓ A **java package** is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form, built-in package and user-defined package.
- ✓ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



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, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:
`package MyPackage;`
- ✓ More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong.
- ✓ You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
`package pkg1[.pkg2[.pkg3]];`
- ✓ A package hierarchy must be reflected in the file system of your Java development system.
For example, a package declared
as `package java.awt.image;`

Finding Packages and CLASSPATH

- ✓ The Java run-time system looks for packages in three ways.
- ✓ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- ✓ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- ✓ Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- ✓ When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is
`C:\MyPrograms\Java\MyPack`
- ✓ Then the class path to **MyPack** is
`C:\MyPrograms\Java`

- ✓ **A simple package example.**
- ✓ Save the following file as A.java in a folder called **pack**.
package pack;

```
public class A {
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

- ✓ Save the following file as B.java in a folder called **mypack**.
package mypack;

```
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

- ✓ Assume that **pack** and **mypack** folders are in the directory **E:/java**.
- ✓ **compile:**

E:/java>javac mypack/B.java

Running:

E:/java>java mypack/B Hello

PACKAGES AND MEMBER ACCESS:

- ✓ Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- ✓ Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- ✓ Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- ✓ The class is Java's smallest unit of abstraction.
- ✓ Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
 - Subclasses in the same package
 - Non-subclasses in the same package
 - Subclasses in different packages
 - Classes that are neither in the same package nor subclasses

CLASS MEMBER ACCESS

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	YES	YES	YES	YES
Visible within same package by subclass	NO	YES	YES	YES
Visible within same package by non-subclass	NO	YES	YES	YES
Visible within different package by subclass	NO	NO	YES	YES
Visible within different package by non-subclass	NO	NO	NO	YES

A package access example:

The following is saved as **Protection.java** in package p1

```
package p1;
public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;
    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Derived.java in package p1

```
package p1;
class Derived extends Protection
{ Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);
    // class only
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
}
}
```

SamePackage.java in package

```
p1 package p1;
class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Protection2.java in package p2

```
package p2;
class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package constructor");
        // class or package only
        // System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

OtherPackage.java in package

p2 package p2;

```
class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");
        // class or package only
        // System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

// Demo package p1.

package p1;

// Instantiate the various classes in p1.

```
public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new
        SamePackage();
    }
}
```

// Demo package p2.

package p2;

// Instantiate the various classes in p2.

```
public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new
        OtherPackage();
    }
}
```

O/P for Demo in p1
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived constructor
n = 1
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
same
package
e constructor
n = 1
n_pro = 3

O/P for Demo in p2
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived other package constructor
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
other package constructor
n_pub = 4

IMPORTING PACKAGES:

- ✓ Java includes the import statement to bring certain classes, or entire packages, into visibility.
- ✓ Once imported, a class can be referred to directly, using only its name.
- ✓ In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.
- ✓ This is the general form of the **import** statement:
import *pkg1* .*pkg2* .*classname* | *;
- ✓ Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.).

Eg: **import** mypack.MyClass;
import mypack.*;

- ✓ * indicates that the Java compiler should import the entire package.
- ✓ All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- ✓ It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

- ✓ The same example without the **import** statement looks like this: class
MyDate extends java.util.Date {
}

JAVA'S Standard packages:

Sub package	Description
java.lang	Contains a large number of general –purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit
java.util	Contains various utility classes, plus the Collections Framework

STATIC IMPORT:

- ✓ Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following import with the keyword static, an import statement can be used to import the static members of a class or interface.
- ✓ When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class.

// Compute the hypotenuse of a right triangle. import

```
java.lang.Math.sqrt;  
import java.lang.Math.pow;  
class Hypot {  
    public static void main(String args[]) {  
        double side1, side2;  
        double hypot;  
        side1 = 3.0;  
        side2 = 4.0;  
        // Notice how sqrt() and pow() must be qualified by  
        // their class name, which is Math.  
        hypot = Math.sqrt(Math.pow(side1, 2) +Math.pow(side2, 2));  
        System.out.println("Given sides of lengths " +side1 + " and "  
            + side2 +" the hypotenuse is " +hypot);  
    }  
}
```

}

Given sides of lengths 3.0 and 4.0 the hypotenuse is 5.0

- ✓ Because **pow()** and **sqrt()** are static methods, they must be called through the use of their class' name, **Math**. This results in a somewhat unwieldy hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

- ✓ We can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

// Compute the hypotenuse of a right triangle.

```
import static java.lang.Math.sqrt; import static
java.lang.Math.pow; class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot; side1 = 3.0;
        side2 = 4.0;
        // Notice how sqrt() and pow() must be qualified by
        // their class name, which is Math.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2)); System.out.println("Given
        sides of lengths " +side1 + " and "
            + side2 +" the hypotenuse is " +hypot);
    }
}
```

- ✓ The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```
import static pkg.type-name.*;
```

- ✓ One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

UNIT III

Exception handling: Hierarchy, Fundamentals, Multiple catch clauses, Subclass exceptions, Nesting try blocks, Throwing an exception, Using Finally and Throws, Built-in exceptions, User-defined exceptions.

I/O: Byte streams and Classes, Character streams and Classes, Predefined streams, Using byte streams, Reading and Writing files using byte streams, Reading and writing binary data, Random-access files. File I/O using character streams. Wrappers

What is exception?

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO, SQL, Remote etc.

Advantage of Exception Handling

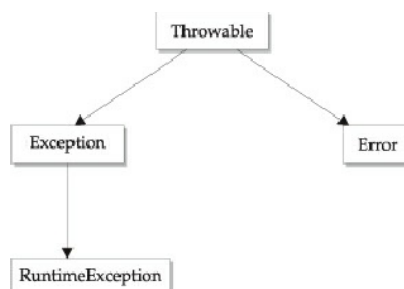
The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; //exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Exception Hierarchy:

- ✓ In Java, all exceptions are represented by classes. All exception classes are derived from a class called **Throwable**.
- ✓ When an exception occurs in a program, an object of some type of exception class is generated.
- ✓ There are two direct subclasses of **Throwable**: **Exception** and **Error**.
- ✓ Exceptions of type **Error** are related to errors that are beyond our control, such as those that occur in the Java Virtual Machine itself.
- ✓ Errors that results from program activity are represented by subclasses of **Exception**. For example, divide-by-zero, array boundary, and I/O errors. An important subclass of **Exception** is **RuntimeException**, which is used to represent various common types of run-time errors.



Exception handling Fundamentals:

- ✓ 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.
- ✓ 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.
- ✓ The code can catch this exception using **catch** and handle it in some rational manner.
- ✓ System-generated exceptions are automatically thrown by the Java runtime 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.
- ✓ The general form of **try/catch** exception handling blocks:


```
try{
    // block of code to monitor for errors
}
catch(ExceptionType1 exOb){
    // handle for ExceptionType1
}
catch(ExceptionType2 exOb){
    // handle for ExceptionType2
}
.
.
```
- ✓ When an exception is thrown, it is caught by its corresponding **catch** clause, which then processes the execution. When an exception is caught *exOb* will receive its value.
- ✓ If no exception is thrown, then a **try** block ends normally, and all of its **catch** blocks are bypassed. Execution resumes with the first statement following the last **catch**.

/*A simple Exception example */

// Demonstrate exception handling.

```
class ExcDemo1 {
    public static void main(String[] args) {
        int[] nums = new int[4];
        try {
            System.out.println("Before exception is generated.");
            // generate an index out-of-bounds exception
            nums[7] = 10;
            System.out.println("this won't be displayed");
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("Index out-of-bounds!");
        }
        System.out.println("After catch.");
    }
}
```

O/P:

Before exception is generated.

Index out-of-bounds!

After catch.

- ✓ Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "this won't be displayed" is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch** mechanism.

Uncaught Exceptions

- ✓ When an exception is thrown, it must be caught by some piece of code; Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- ✓ Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
    public static void main(String args[]){
        int data=50/0;//may throw exception
        System.out.println("rest of the code...");
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

- ✓ As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).
- ✓ Solution by exception handling
- ✓ It is important to handle exceptions by the program itself rather than rely on JVM Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
    public static void main(String args[]){
        try{
            int data=50/0;
        }catch(ArithmeticException e){System.out.println(e);}
        System.out.println("rest of the code...");
    }
}
```

- ✓ Output:

Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...

- ✓ Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.
- ✓ The type of exception must match the type specified in a catch. If it does not, the exception would not be caught.

// This won't work! class

```
ExcTypeMismatch {
    public static void main(String[] args) {
        int[] nums = new int[4];

        try {
            System.out.println("Before exception is generated.");

            //generate an index out-of-bounds exception
            nums[7] = 10;
            System.out.println("this won't be displayed");
        }

        /* Can't catch an array boundary error with an ArithmeticException. */
        catch (ArithmeticException exc) {
            // catch the exception
            System.out.println("Index out-of-bounds!");
        }
    }
}
```

Multiple catch Clauses

- ✓ We can specify two or more **catch** clauses, each catching a different type of exception.
- ✓ When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block.

```
public class TestMultipleCatchBlock{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;

        }
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
        catch(Exception e){System.out.println("common task completed");}
        System.out.println("rest of the code...");
    }
}
```

Output:

task1 completed
rest of the code...

- ✓ **Rule: At a time only one Exception is occurred and at a time only one catch block is executed.**

Catching subclass Exceptions

- ✓ When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.
- ✓ **Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.**

class TestMultipleCatchBlock1{

```
public static void main(String args[]){
    try{
        int a[]=new int[5];
        a[5]=30/0;

    }
    catch(Exception e){System.out.println("common task completed");}
    catch(ArithmeticException e){System.out.println("task1 is completed");}
    catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
    System.out.println("rest of the code...");
}
}
```

Output:

Compile-time error

Nesting try blocks

- ✓ The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- ✓ Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- ✓ If no **catch** statement matches, then the Java run-time system will handle the exception.

// Use a nested try block.

```
class nestTrys {
    public static void main(String[] args) {
        // Here, number is longer than denom.
        int[] number = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        try { // outer try
            for(int i=0; i<number.length; i++) {
                try { // nested try
                    System.out.println(number[i] + " / " +
                                         denom[i] + " is " +
                                         number[i]/denom[i]);
                }
                catch (ArithmeticException exc) {
                    // catch the exception
                    System.out.println("Can't divide by Zero!");
                }
            }
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("No matching element found.");
            System.out.println("Fatal error - program terminated.");
        }
    }
}
```

OUTPUT:

```
4 / 2 is 2
Can't divide by Zero! 16 /
4 is 4
32 / 4 is 8
Can't divide by Zero! 128 /
8 is 16
No matching element found. Fatal error
- program terminated.
```


THROWING AN EXCEPTION:

- ✓ It is possible to manually throw an exception explicitly, by using the **throw** statement.
- ✓ The general form of **throw** is shown here:
`throw exceptionobject;`

Here, *exceptionobject* 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.

- ✓ The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.

// Manually throw an exception. class

```
ThrowDemo {
    public static void main(String[] args) {
        try {
            System.out.println("Before throw.");
            throw new ArithmeticException();
        }
        catch (ArithmeticException exc) {
            System.out.println("Exception caught.");
        }
        System.out.println("After try/catch block.");
    }
}
```

OUTPUT:

Before throw. Exception
 caught. After try/catch
 block.

Rethrowing an Exception:

- ✓ An exception caught by one catch can be rethrown so that it can be caught by an outer catch. The most likely reason for rethrowing this way is to allow multiple handlers access to the exception.
- ✓ To rethrow an exception use a **throw** statement inside a **catch** clause, throwing the exception passed as an argument.

// Rethrow an exception.

```
class Rethrow {
    public static void genException() {
        // here, numer is longer than denom
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +denom[i] + " is " + numer[i]/denom[i]);
            }
            catch (ArithmeticException exc) {
                // catch the exception
                System.out.println("Can't divide by Zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("No matching element found.");
                throw exc; // rethrow the exception
            }
        }
    }
}
```

```

class RethrowDemo {
    public static void main(String[] args) {
        try {
            Rethrow.genException();
        }
        catch(ArrayIndexOutOfBoundsException exc) {
            // rethrow exception
            System.out.println("Fatal error - " +
                               "program terminated.");
        }
    }
}

```

OUTPUT:

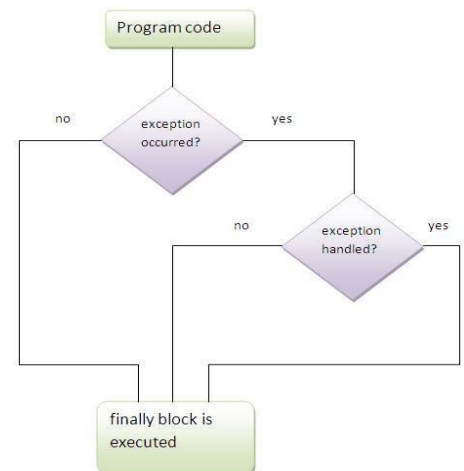
```

4 / 2 is 2
Can't divide by Zero! 16 /
4 is 4
32 / 4 is 8
Can't divide by Zero! 128 /
8 is 16
No matching element found. Fatal error
- program terminated.

```

Using finally:

- ✓ **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- ✓ The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- ✓ Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- ✓ The **finally** clause is optional.
- ✓ **finally** block in java can be used to put "cleanup" code such as closing a file, closing connection etc
- ✓ **Rule: For each try block there can be zero or more catch blocks, but only one finally block.**
- ✓ **Note: The finally block will not be executed if program exits(either by calling System.exit()) or by causing a fatal error that causes the process to abort).**



Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```

class TestFinallyBlock{
    public static void main(String args[]){
        try{
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}

```

Output:

5

finally block is always executed
rest of the code...

Case 2

Let's see the java finally example where **exception occurs and not handled**. **class** TestFinallyBlock1{

```

    public static void main(String args[]){
    try{
        int data=25/0;
        System.out.println(data);
    }
    catch(NullPointerException e){System.out.println(e);}
    finally{System.out.println("finally block is always executed");}
    System.out.println("rest of the code...");
    }
}

```

Output:

finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```

public class TestFinallyBlock2{
    public static void main(String args[]){
    try{
        int data=25/0;
        System.out.println(data);
    }
    catch(ArithmeticException e){System.out.println(e);}
    finally{System.out.println("finally block is always executed");}
    System.out.println("rest of the code...");
    }
}

```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

Using Throws:

- ✓ If a method generates an exception that it does not handle, it must specify that exception in a throws clause.
- ✓ 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. If they are not, a compile-time error will result.
- ✓ This is the general form of a method declaration that includes a **throws** clause:
- ✓ *type method-name(parameter-list) throws exception-list*

```

{
    // body of method
}

```

- ✓ Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.
- ✓ Which exception should be declared

Ans) checked exception only, because:

unchecked Exception: under your control so correct your code.

error: beyond your control e.g. you are unable to do anything if there occurs

VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

- ✓ Now Checked Exception can be propagated (forwarded in call stack). It provides information to the caller

of the method about the exception.

Java throws example

- ✓ Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n()throws
        IOException{ m();
    }
    void
        p(){ try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:
exception handled
normal flow...

- ✓ **Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.**

There are two cases:

- ✓ **Case1:** You caught the exception i.e. handle the exception using try/catch.
- ✓ **Case2:** You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- ✓ In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
public class Testthrows2{
    public static void main(String args[]){
        try{
            M m=new
            M();
            m.method();
        }catch(Exception e){System.out.println("exception handled");}

        System.out.println("normal flow...");
    }
}
```

Output:
exception handled

normal flow...

Case2: You declare the exception

- ✓ A) In case you declare the exception, if exception does not occur, the code will be executed fine.
- ✓ B) In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A) Program if exception does not occur

```
import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
```

Output:

device operation
performed normal flow...

B) Program if exception occurs

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
```

Output:

Runtime Exception

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

throw	throws
1) Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2) Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3) Throw is followed by an instance.	Throws is followed by class.
4) Throw is used within the method.	Throws is used with the method signature.
5) You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java's Built-in Exceptions

- ✓ Inside the standard package **java.lang**, Java defines several exception classes.
- ✓ The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- ✓ In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table.
- ✓ Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

- ✓ Following is the list of Java Checked Exceptions Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

User-Defined Exceptions:

- ✓ We can create our own exceptions easily by making them a subclass of an existing exception, such as Exception, the superclass of all exceptions. In a subclass of Exception, there are only two methods you might want to override: Exception () with no arguments and Exception () with a String as an argument. In the latter, the string should be a message describing the error that has occurred.

```
import java.util.Scanner;
class InvalidAgeException extends Exception{
    InvalidAgeException(String s){
        super(s);
    }
    public String toString(){
        return "age is less than 18. not eligible to vote";
    }
}
class Validateage {
    static void validate(int age)throws InvalidAgeException{
        if(age<18)
            throw new InvalidAgeException("not valid");
        else
            System.out.println("welcome to vote");
    }
}

public class UDExceptiton {
    public static void main(String args[]){
        Scanner s=new
        Scanner(System.in);
        System.out.println("Enter age");
        int age=s.nextInt();
        try{
            Validateage.validate(age);
        }
        catch(Exception m){
            System.out.println("Exception occured: "+m);
        }
    }
}
```

I/O:

- ✓ Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- ✓ All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.
- ✓ Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

- ✓ Java defines two types of streams: byte and character.
- ✓ *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.
- ✓ *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.
- ✓ At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes

- ✓ Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in **java.io** are shown in Table

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Character Stream Classes

- ✓ Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams.
- ✓ Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in Table .

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print() and println()
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams

- ✓ **java.lang** package defines a class called **System**, which encapsulates several aspects of the runtime environment.
- ✓ **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.
- ✓ **System.out** refers to the standard output stream. By default, this is the console.
- ✓ **System.in** refers to standard input, which is the keyboard by default.
- ✓ **System.err** refers to the standard error stream, which also is the console by default.
- ✓ **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console.

Using Byte streams:

Reading Console Input

- ✓ **InputStream**, defines only one input method, **read()**, which reads bytes. There are three versions of **read()**

int read() throws **IOException**

int read(byte[] buffer) throws **IOException**

int read(byte[] buffer, int offset, int numBytes) throws **IOException**

- ✓ The first version read a single character from the keyboard. It returns -1 when the end of stream is encountered.
- ✓ The second version reads bytes from the input stream and puts them into buffer until either the array is full, the end of stream is reached or an error occurs. It returns the number of bytes read or -1 when the end of stream is encountered.
- ✓ The third version reads input into buffer beginning at location specified by offset upto numBytes bytes are stored. It returns the number of bytes read or -1 when the end of stream is encountered.

```
import java.io.*; class
```

```
ReadBytes {
    public static void main(String[] args) throws IOException {
        byte[] data = new byte[10];

        System.out.println("Enter some characters.");
        int numRead = System.in.read(data);
        System.out.print("You entered: ");
        for(int i=0; i < numRead; i++)
            System.out.print((char) data[i]);
    }
}
```

Writing Console Output

- ✓ Console output is most easily accomplished with **print()** and **println()**. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**).
- ✓ Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write()**. Thus, **write()** can be used to write to the console. The simplest form of **write()** defined by **PrintStream** is shown here:

```
void write(int b)
```

- ✓ This method writes the byte specified by **b**. Although **b** is declared as an integer, only the low- order eight bits are written.

```
// Demonstrate System.out.write().
class WriteDemo {
    public static void main(String[] args) {
        int b;

        b = 'X';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

READING AND WRITING FILES USING BYTE STREAMS:

- ✓ Java provides a number of classes and methods that allow you to read and write files.
- ✓ Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. `FileInputStream(String fileName)` throws `FileNotFoundException`

`FileOutputStream(String fileName)` throws `FileNotFoundException`

- ✓ Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.
- ✓ **FileNotFoundException** is a subclass of **IOException**.
- ✓ When an output file is opened, any preexisting file by the same name is destroyed.
- ✓ To read from a file, you can use a version of **read()** that is defined within **FileInputStream**. `int read()` throws `IOException`
- ✓ Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read()** returns `-1` when the end of the file is encountered. It can throw an **IOException**.
- ✓ When you are done with a file, you must close it. This is done by calling the **close()** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:
`void close()` throws `IOException`
- ✓ Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

/* Display a text file.

To use this program, specify the name of the file that you want to see. For example, to see a file called TEST.TXT, use the following command line. `java ShowFile TEST.TXT*/`

```
import java.io.*;
class ShowFile
{
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin;
        // First make sure that a file has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile File");
            return;
        }
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException exc) {
            System.out.println("File Not Found");
            return;
        }
        try {
            // read bytes until EOF is encountered
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        } catch(IOException exc) {
            System.out.println("Error reading file.");
        }
        try {
            fin.close();
        } catch(IOException exc) {
            System.out.println("Error closing file.");
        }
    }
}
```

- ✓ In the above code **close()** can be written within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file.
- ✓ Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file.

/* This variation wraps the code that opens and accesses the file within a single try block.
The file is closed by the finally block. */

```
import java.io.*;

class ShowFile {
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin = null;

        // First, confirm that a file name has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code opens a file, reads characters until EOF
        // is encountered, and then closes the file via a finally block.
        try {
            fin = new FileInputStream(args[0]);

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException exc) {
            System.out.println("File Not Found.");
        } catch(IOException exc) {
            System.out.println("An I/O Error Occurred");
        } finally {
            // Close file in all cases.
            try {
                if(fin != null) fin.close();
            } catch(IOException exc) {
                System.out.println("Error Closing File");
            }
        }
    }
}
```

Writing to a file:

- ✓ To write to a file, you can use the **write()** method defined by **FileOutputStream**. void write(int b) throws IOException
 - ✓ This method writes the byte specified by *b* to the file. Although *b* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown.
- /* Copy a text file. To use this program, specify the name of the source file and the destination file. For example, to copy a file called FIRST.TXT to a file called SECOND.TXT, use the following command line. java CopyFile FIRST.TXT SECOND.TXT */

```
import java.io.*;

class CopyFile {
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // First, make sure that both files has been specified.
        if(args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }

        // Copy a File.
        try {
            // Attempt to open the files.
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);

            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);

        } catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        } finally {
            try {
                if(fin != null) fin.close();
            } catch(IOException exc) {
                System.out.println("Error Closing Input File");
            }
            try {
                if(fout != null) fout.close();
            } catch(IOException exc) {
                System.out.println("Error Closing Output File");
            }
        }
    }
}
```


AUTOMATICALLY CLOSING A FILE:

- ✓ JDK 7 adds a new feature to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short.
- ✓ Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {
    // use the resource
}
```

- ✓ Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream.
- ✓ It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released.
- ✓ The **try-with-resources** statement can be used only with those resources that implement the

AutoCloseable interface defined by **java.lang**. This interface defines the **close()** method.

- ✓ **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try-with-resources** can be used when working with streams, including file streams.

/* This version of the ShowFile program uses a try-with-resources statement to automatically close a file when it is no longer needed.

Note: This code requires JDK 7 or later.

*/

```
import java.io.*; class
```

```
ShowFile {
    public static void main(String[] args)
    {
        int i;

        // First, make sure that a file name has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code uses try-with-resources to open a file
        // and then automatically close it when the try block is left.
        try(FileInputStream fin = new FileInputStream(args[0])) {

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        }
        catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

- ✓ We can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon.

```
try (FileInputStream fin = new FileInputStream(args[0]);
    FileOutputStream fout = new FileOutputStream(args[1]))
```

Reading and writing binary data

- ✓ To read and write binary values of java primitive types we can use **DataInputStream** and **DataOutputStream**.
- ✓ **DataOutputStream** implements the **DataOutput** interface. This interface defines methods that write all of Java's primitive types to a file. It is important to understand that this data is written using its internal, binary format.
- ✓ Output methods defined by **DataOutputStream**.

Output Method	Purpose
void writeBoolean(boolean val)	Writes the boolean specified by val
void writeByte(int val)	Writes the lower-order byte specified by val
void writeChar(int val)	Writes the value specified by val as a char
void writeDouble(double val)	Writes the double specified by val
void writeFloat(float val)	Writes the float specified by val
void writeInt(int val)	Writes the int specified by val
void writeLong(long val)	Writes the long specified by val
void writeShort(int val)	Writes the value specified by val as a short

- ✓ The constructor for **DataOutputStream** is :
DataOutputStream(**OutputStream** outputstream)
 Here outputstream is the stream to which data is written. To write output to a file we can use the object created by **FileOutputStream**.
- ✓ **DataInputStream** implements the **DataInput** interface which provides methods for reading all of java's primitives

Input Method	Purpose
boolean readBoolean()	Reads a boolean
byte readByte()	Reads a byte
char readChar()	Reads a char
double readDouble()	Reads a double
float readFloat()	Reads a float
int readInt()	Reads an int
long readLong()	Reads a long
short readShort()	Reads a short

- ✓ The constructor for **DataInputStream** is:
DataInputStream(**InputStream** inputstream)
 Here inputstream is the stream that is linked to the instance of **DataInputStream** being created. To read input from a file we can use the object created by **FileInputStream**

// Write and then read back binary data.
 // This code requires JDK 7 or later.

```
import java.io.*; class
RWData {
    public static void main(String[] args)
    {
        int i = 10;
        double d = 1023.56;
        boolean b = true;
        // Write some values.
        try (DataOutputStream dataOut =
            new DataOutputStream(new FileOutputStream("testdata")))
        {
            System.out.println("Writing " + i);
            dataOut.writeInt(i);

            System.out.println("Writing " + d);
```

```
dataOut.writeDouble(d);
```

```
        System.out.println("Writing " + b);
        dataOut.
        writeBoolean(b);

        System.out.println("Writing " +12.2 * 7.4);
        dataOut.writeDouble(12.2 * 7.4);
    }
    catch(IOException exc) {
        System.out.println("Write error.");
        return;
    }
    System.out.println();
    // Now, read them back.
    try (DataInputStream dataIn =
        new DataInputStream(new FileInputStream("testdata")))
    {
        i = dataIn.readInt();
        System.out.println("Reading " + i);
        d = dataIn.readDouble();
        System.out.println("Reading " + d);

        b = dataIn.readBoolean();
        System.out.println("Reading " + b);

        d = dataIn.readDouble();
        System.out.println("Reading " + d);
    }
    catch(IOException exc) {
        System.out.println("Read error.");
    }
}
```

OUTPUT:

Writing 10
Writing 1023.56
Writing true Writing
90.28

Reading 10
Reading 1023.56
Reading true Reading
90.28

RANDOM ACCESS FILES:

- ✓ To access the contents of a file in random order we can use **RandomAccessFile**.
- ✓ **RandomAccessFile** implements the interface **DataInput** and **DataOutput**.
RandomAccessFile(String filename, String access) throws FileNotFoundException
- ✓ Here, the name of the file is passed in filename, and access determines what type of file access is permitted. "r"- the file can be read but not written;
- "rw"- the file is opened in read-write mode
- ✓ The method seek() is used to set current position of the file pointer within the file:
void seek(long newPos) throws IOException
- ✓ Here newPos specifies the new position, in bytes, of the file pointer from the beginning of the file.

```
import java.io.*; class
```

```
Random{
    public static void main(String[] args)
    {
        int d;
        // Open and use a random access file.
        try (RandomAccessFile raf = new RandomAccessFile("knr.txt", "rw"))
        {
            System.out.println("Every fifth characters in the file is : ");
            int i=0;
            do {
                raf.seek(i);
                d = raf.read();
                if(d!=-1)
                    System.out.print((char)d + " ");
                i=i+5;
            }while(d!=-1);
        }
        catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

For execution of this program assume that there exists a file called knr.txt which contains alphabets a-z.

When the above program is executed we get the following output;

```
E:\java>javac Random.java
```

```
E:\java>java Random
```

```
Every fifth characters in the file is : a f k p
u z
```

FILE I/O USING CHARACTER STREAMS:

- ✓ To perform character based file I/O we can use **FileReader** and **FileWriter** classes.

Using FileWriter

- ✓ **FileWriter** creates a **Writer** that you can use to write to a file. Two of its most commonly used constructors are shown here:

FileWriter(String *filePath*) throws **IOException** **FileWriter**(String *filePath*, boolean *append*) throws **IOException**

- ✓ **FileWriter** is derived from **OutputStreamWriter** and **Writer**. Thus it has access to the methods defined by those classes.
- ✓ Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object.

```
import java.io.*; class
KtoD{
    public static void main(String[] args){
        String str;
        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in)); System.out.println("enter test.stop to quit");
        try(FileWriter fw=new FileWriter("test.txt")){
            do{
                System.out.print(":");
                str=br.readLine();
                if (str.compareTo("stop")==0) break;
                str=str+"\r\n";
                fw.write(str);
            }while(str.compareTo("stop")!=0);
        }
        catch(IOException exe){
            System.out.println("I/O Error: " +exe);
        }
    }
}
```

Using a FileReader:

- ✓ The **FileReader** class creates a **Reader** that you can use to read the contents of a file. A commonly used constructors is shown here:
FileReader(String *filePath*) throws **FileNotFoundException**
- ✓ **FileReader** is derived from **InputStreamReader** and **Reader**. Thus, it has access to the methods defined by those classes.

```
import java.io.*; class
Dtos{
    public static void main(String[] args){
        String str;
        try(BufferedReader br=new BufferedReader(new FileReader("test.txt"))){
            while((str=br.readLine())!=null){
                System.out.println(str);
            }
        }
        catch(IOException exe){
            System.out.println("I/O Error: " +exe);
        }
    }
}
```

File:

- ✓ **File** class deals directly with files and file systems. The **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.
- ✓ A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies. File (String path) File (String directory path, String filename)

Obtaining a File's properties:

- ✓ **File** defines many methods that obtain the standard properties of a **File** object.

Method	Description
boolean canRead()	Returns true if the file can be read
boolean canWrite()	Returns true if the file can be written
boolean exists()	Returns true if the file exists
String getAbsolutePath()	Returns the absolute path to the file
String getName()	Returns the file's name
String getParent()	Returns the name of the file's parent directory, or null if no parent exists
boolean isAbsolute()	Returns true if the path is absolute. It returns false if the path is relative
boolean isDirectory()	Returns true if the file is directory
boolean isFile()	Returns true if the file is a normal file. It returns false if the file is a directory or some other nonfile object.
boolean isHidden()	Returns true if the invoking file is hidden. Returns false otherwise
long length()	Returns the length of the file, in bytes.

// Obtain information about a file.

```
import java.io.*;
class FileDemo
{
    public static void main(String[] args) {
        File myFile = new File("/pack/k.txt");
        System.out.println("File Name: " + myFile.getName());
        System.out.println("Path: " + myFile.getPath());
        System.out.println("Abs Path: " + myFile.getAbsolutePath());
        System.out.println("Parent: " + myFile.getParent());
        System.out.println(myFile.exists() ? "exists" : "does not exist");
        System.out.println(myFile.isHidden() ? "is hidden": "is not hidden");
        System.out.println(myFile.canWrite() ? "is writeable": "is not writeable");
        System.out.println(myFile.canRead() ? "is readable": "is not readable");
        System.out.println("is " + (myFile.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(myFile.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(myFile.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File size: " + myFile.length() + " Bytes");
    }
}
```

Obtaining a Directory Listing:

- ✓ A directory is a file that contains a list of other files and directories. When we create a File object that is a directory, the **isDirectory()** method will return true.
- ✓ The list of the files in the directory can be obtained by calling list() on the object. String[] list()

```
// Using directories. import
java.io.*; class DirList {
    public static void main(String[] args) {
        String dirname = "/java";
        File myDir = new File(dirname);
        if (myDir.isDirectory()) {
            System.out.println("Directory of " + dirname);
            String[] s = myDir.list();
        }
    }
}
```

```

    for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
            System.out.println(s[i] + " is a directory");
        } else {
            System.out.println(s[i] + " is a file");
        }
    }
} else {
    System.out.println(dirname + " is not a directory");
}
}
}

```

The listFiles() Alternative

- ✓ There is a variation to the **list()** method, called **listFiles()**. The signatures for **listFiles()** are shown here:

```
File[ ] listFiles( )
```

```
File[ ] listFiles(FileNameFilter FFObj) File[ ]
```

```
listFiles(FileFilter FFObj)
```

- ✓ These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FileNameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles()** work like their equivalent **list()** methods.

The third version of **listFiles()** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept()**, which is called once for each file in a list. Its general form is given here:

```
boolean accept(File path)
```

The **accept()** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

Various File Utility methods:

- ✓ **File** includes several other utility methods.

Method	Description
boolean delete()	Deletes the file specified by the invoking object. returns true if the file was deleted and false if the file cannot be removed
void deleteOnExit()	removes the file associated with the invoking object when the java virtual machine terminates
long getFreeSpace()	Returns the number of free bytes of storage available on the partition associated with the invoking object.
boolean mkdir()	Creates the directory specified by the invoking object. Returns true if the directory was created and false if the directory could not be created.
boolean mkdirs()	Creates the directory and all required parent directories specified by the invoking object. Returns true if the entire path was created and false otherwise.
boolean setReadOnly()	set the file read-only
boolean setWritable(boolean how)	If how is true, the file is set to writable. If how is false, the file is set to read only. Returns true if the status of the file was modified and false if the write status cannot be changed.

WRAPPERS

- ✓ Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object.
- ✓ The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Wrapper	Conversion Method
Double	static double parseDouble(string str) throws NumberFormatException
Float	static float parseFloat(String str) throws NumberFormatException
Long	static long parseLong(String str) throws NumberFormatException
Wrapper	Conversion Method
Integer	static int parseInt(String str) throws NumberFormatException
Short	static short parseShort(String str) throws NumberFormatException
Byte	static byte parseByte(String str) throws NumberFormatException

// This program averages a list of numbers entered by the user.

```
import java.io.*; class
AvgNums {
    public static void main(String[] args) throws IOException {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); String
        str;
        int n;
        double sum = 0.0; double
        avg, t;

        System.out.print("How many numbers will you enter: "); str =
        br.readLine();
        try {
            n = Integer.parseInt(str);
        }
        catch(NumberFormatException exc) {
            System.out.println("Invalid format"); n = 0;
        }

        System.out.println("Enter " + n + " values."); for(int
        i=0; i < n ; i++) {
            System.out.print(": "); str =
            br.readLine(); try {
                t = Double.parseDouble(str);
            } catch(NumberFormatException exc) {
                System.out.println("Invalid format"); t = 0.0;
            }
            sum += t;
        }
        avg = sum / n; System.out.println("Average is "
        + avg);
    }
}
```

Object Oriented Programming using JAVA

UNIT IV

Multithreaded Programming

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a **thread**, and each **thread** defines a separate path of execution.
- A thread introduces **Asynchronous** behaviour.

Multitasking

Two types:

1. Process based multitasking
2. Thread based multitasking

Process based multitasking	Thread based multitasking
Process is a program under execution	Thread is one part of a program.
Two or more programs run concurrently	Two or more parts of a single program run concurrently.
Heavyweight process.	Lightweight process.
Programs requires separate address spaces	Same address space is shared by threads.
Interprocess communication is expensive and limited.	Interthread communication is inexpensive.
Context switching from one process to another is also costly.	Context switching from one thread to the next is lower in cost.
May create more idle time.	Reduces the idle time.
Ex: Running a Java compiler and downloading a file from a web site at the same time	Ex: We can format a text using a Text editor and printing the data at the same time.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

a) The Thread class and the Runnable Interface

Java's multithreading system is built upon the **Thread class**, its methods, and its companion interface, **Runnable**.

Thread class methods

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
setName()	Give a name to a thread
setPriority()	Set the priority to a thread
isAlive()	Determine if a thread is still running.
Join()	Wait for a thread to terminate.
Run()	Entry point for the thread.
Sleep()	Suspend a thread for a period of time.
Start()	Start a thread by calling its run method.
currentThread()	returns a reference to the thread in which it is called

b) The Main thread

- When a Java program starts up, one thread begins running immediately. This is usually called the **main thread of the program**.
- It is the thread from which other "child" threads will be spawned.

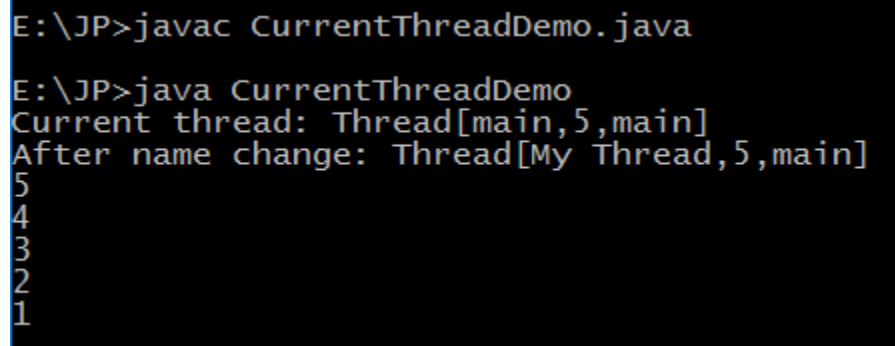
// Controlling the main Thread.

```
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
    }
}
```

```
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);

try {
    for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}
}
```

Output:



```
E:\JP>javac CurrentThreadDemo.java
E:\JP>java CurrentThreadDemo
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Thread Group:

A **thread group** is a data structure that controls the state of a collection of threads as a whole.

Creating and Starting a Thread

Two ways to create a thread –

1. By extending Thread class
2. By implementing Runnable interface

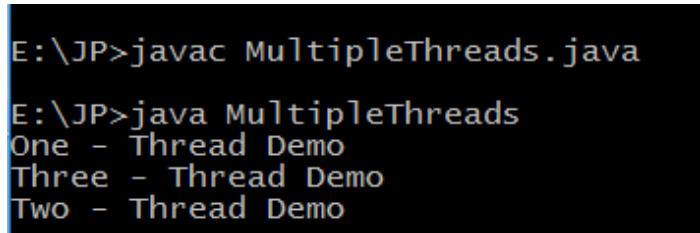
→ **run()** method is the entry point for another concurrent thread of execution in the program.

1. By extending Thread class

- The first way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run() method, which is the entry point for the new thread.
- It must also call start() to begin execution of the new thread.

Example

```
class MultipleThreads extends Thread
{
    MultipleThreads(String name)
    {
        super(name);
        start();
    }
    public void run()
    {
        System.out.print(Thread.currentThread().getName());
        System.out.println(" - Thread Demo");
    }
    public static void main(String ar[])
    {
        MultipleThreads t1 = new MultipleThreads("One");
        MultipleThreads t2 = new MultipleThreads("Two");
        MultipleThreads t3 = new MultipleThreads("Three");
    }
}
```



```
E:\JP>javac MultipleThreads.java
E:\JP>java MultipleThreads
One - Thread Demo
Three - Thread Demo
Two - Thread Demo
```

2. By implementing Runnable interface

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

public void run()

- Inside run(), we will define the code that constitutes the new thread.
- It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run() establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run() returns.

Example

```
class RMultipleThreads implements Runnable
{
    String tname;
    Thread t;
    RMultipleThreads(String name)
    {
        tname = name;
        t = new Thread(this,tname);
        t.start();
    }
    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"- Thread Demo");
    }
    public static void main(String ar[])
    {
        RMultipleThreads t1 = new RMultipleThreads("One");
        RMultipleThreads t2 = new RMultipleThreads("Two");
        RMultipleThreads t3 = new RMultipleThreads("Three");
    }
}
```

```
E:\JP>javac RMultipleThreads.java
E:\JP>java RMultipleThreads
Three- Thread Demo
One- Thread Demo
Two- Thread Demo
```

Which of the above two approaches is better?

- The first approach is easier to use in simple applications, but is limited by the fact that our class must be a subclass of Thread. In this approach, our class can't extend any other class.
- The second approach, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

Thread Priorities, isAlive() and join() methods

Thread Priorities

- 1) Every thread has a priority that helps the operating system to determine the order in which threads are scheduled for execution.
- 2) Thread priorities are integers that ranges between, 1 to 10.

MIN-PRIORITY (a constant of 1)

MAX-PRIORITY (a constant of 10)

- 3) By default every thread is given a **NORM-PRIORITY(5)**. The **main** thread always have **NORM-PRIORITY**.
- 4) Thread's priority is used in **Context Switch**.

Context Switch

Switching from one running thread to the next thread is called as **Context Switch**.

Rules for Context Switch

- 1) **A thread can voluntarily relinquish control:** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- 2) **A thread can be preempted by a higher-priority thread:** In this case, a lower-priority thread preempted by a higher-priority thread. This is called **preemptive multitasking**.

Note:

- In *windows*, **threads of equal priority** are time-sliced automatically in round-robin fashion.
- For other types of operating systems, **threads of equal priority** must voluntarily yield control to their peers. If they don't, the other threads will not run.

final void setPriority(int level)

To set a thread's priority, use the setPriority() method, which is a member of Thread.

final int getPriority()

By using getPriority(), we can obtain the current priority.

isAlive() & join()

isAlive() -> The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise.

final boolean isAlive()

join() -> This method waits until the thread on which it is called terminates.

final void join() throws InterruptedException**Example:**

```
class ThreadDemo implements Runnable
{
    public void run()
    {
        try
        {
            for(int i=0;i<3;i++)
            {
                System.out.println("Thread
Demo:"+Thread.currentThread().getName());
                Thread.currentThread().sleep(1000);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println("Thread interrupted");
        }
    }
}

class MultiThreadDemo{
    public static void main(String ar[])
    {
        ThreadDemo r = new ThreadDemo();
        Thread t1 = new Thread(r);
        t1.setName("First Thread");
        t1.setPriority(2);
        t1.start();

        Thread t2 = new Thread(r);
        t2.setName("Second Thread");
        t2.setPriority(7);
    }
}
```



```

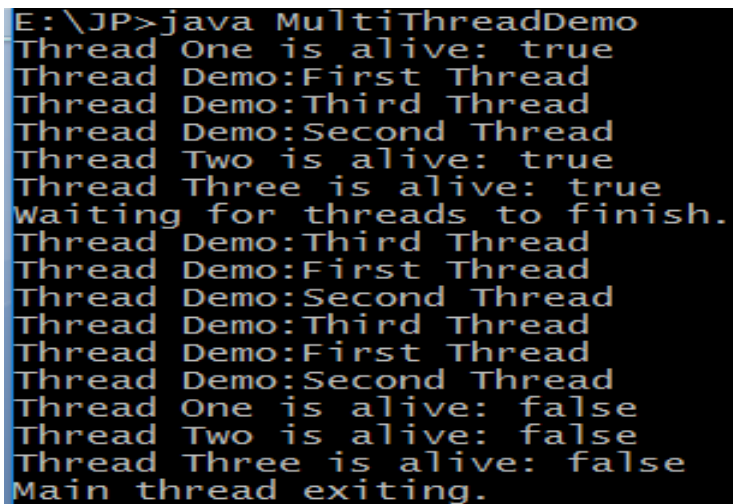
        t2.start();

        Thread t3 = new Thread(r);
        t3.setName("Third Thread");
        t3.setPriority(9);
        t3.start();

        System.out.println("Thread One is alive: "+ t1.isAlive());
        System.out.println("Thread Two is alive: "+ t2.isAlive());
        System.out.println("Thread Three is alive: "+ t3.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            t1.join();
            t2.join();
            t3.join();
        }
        catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "+ t1.isAlive());
        System.out.println("Thread Two is alive: "+ t2.isAlive());
        System.out.println("Thread Three is alive: "+ t3.isAlive());

        System.out.println("Main thread exiting.");
    }
}

```



```

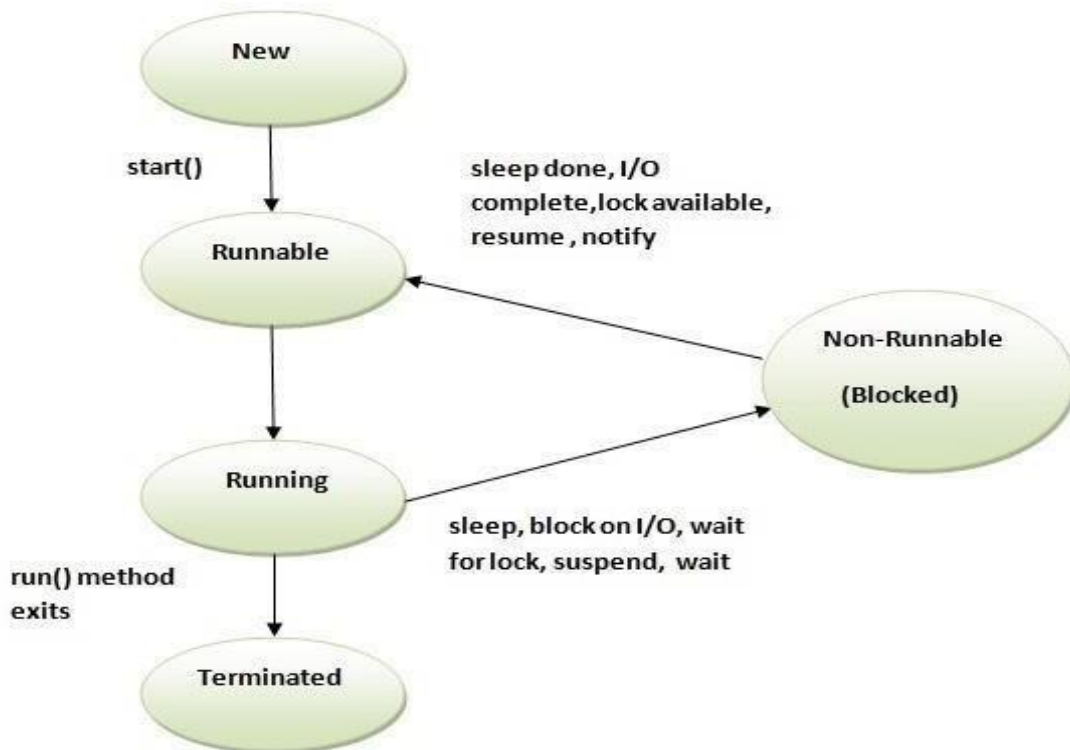
E:\JP>java MultiThreadDemo
Thread One is alive: true
Thread Demo:First Thread
Thread Demo:Third Thread
Thread Demo:Second Thread
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Thread Demo:Third Thread
Thread Demo:First Thread
Thread Demo:Second Thread
Thread Demo:Third Thread
Thread Demo:First Thread
Thread Demo:Second Thread
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

```

Thread States / Life cycle of a thread

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

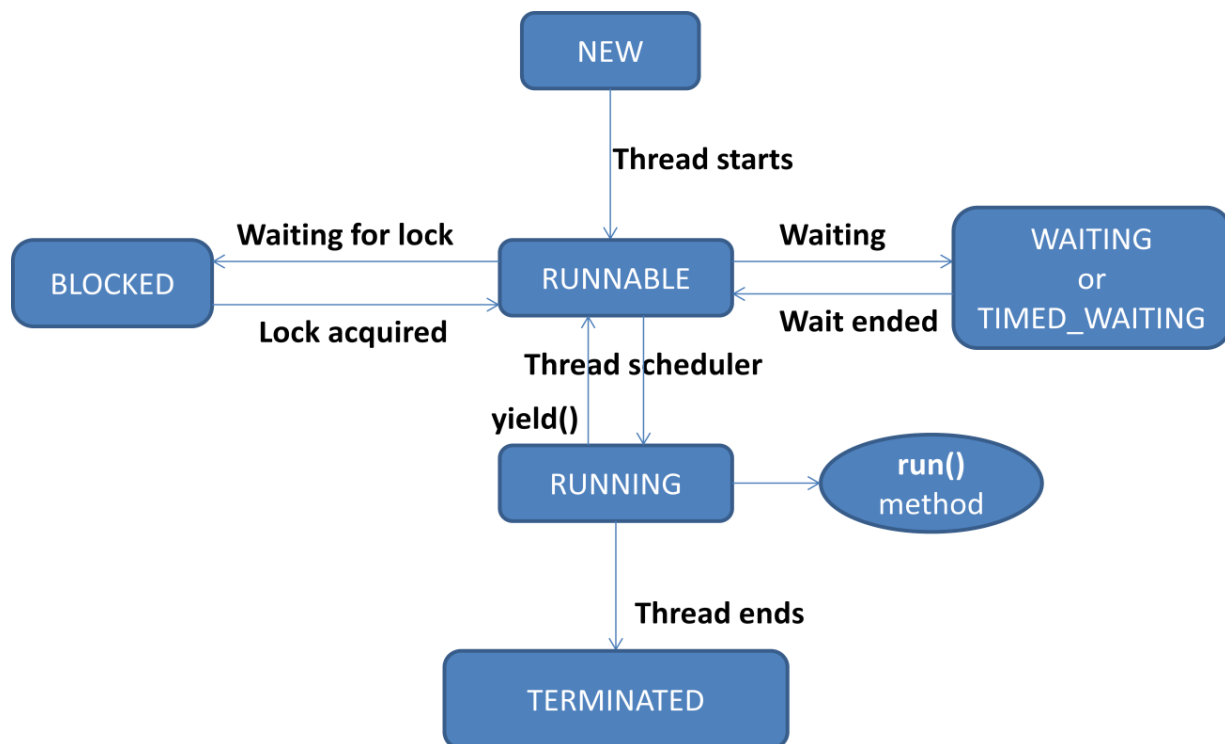


Obtaining A Thread's State

We can obtain the current state of a thread by calling the `getState()` method defined by `Thread`.

`Thread.State getState()`

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code> . This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code> .



Synchronization

Definition:

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called **synchronization**.

Process behind Synchronization

- Key to synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.

Synchronizing code

We can synchronize the code in two ways:

1. Using synchronized methods

```
synchronized void test( )
{
}
```

2. Using Synchronized statements (synchronized blocks)

synchronized statement

```
synchronized(objRef) {
    // statements to be synchronized
}
```

Here, objRef is a reference to the object being synchronized

Example for Synchronization

```
class Account {
    private int balance = 50;
    public int getBalance()
```

```

    {
        return balance;
    }
    public void withdraw(int amount)
    {
        balance = balance - amount;
    }
}
class AccountDanger implements Runnable
{
    private Account acct = new Account();

    public void run() {
        for (int x = 0; x < 5; x++) {
            makeWithdrawal(10);
            if (acct.getBalance() < 0) {
                System.out.println("account is overdrawn!");
            }
        }
    }

    private synchronized void makeWithdrawal(int amt) {
        if (acct.getBalance() >= amt) {
            System.out.println(Thread.currentThread().getName()+ " is going
            to withdraw");
            try {
                Thread.sleep(500);
            }
            catch (InterruptedException ex) { }
            acct.withdraw(amt);

            System.out.println(Thread.currentThread().getName()+ "
            completes the withdrawal");
        }
        else {

```

```

        System.out.println("Not enough in account for " +
        Thread.currentThread().getName()+ " to withdraw " + acct.getBalance());
    }
}
}
public class SyncExample
{
    public static void main (String [] args)
    {
        AccountDanger r = new AccountDanger();
        Thread t1 = new Thread(r);
        t1.setName("A");
        Thread t2 = new Thread(r);
        t2.setName("B");
        t1.start();
        t2.start();
    }
}

```

Output

```

E:\JP>java SyncExample
A is going to withdraw
A completes the withdrawal
A is going to withdraw
A completes the withdrawal
A is going to withdraw
A completes the withdrawal
B is going to withdraw
B completes the withdrawal
B is going to withdraw
B completes the withdrawal
Not enough in account for A to withdraw 0
Not enough in account for B to withdraw 0
Not enough in account for A to withdraw 0
Not enough in account for B to withdraw 0
Not enough in account for B to withdraw 0

```

Interthread Communication

- Java includes an elegant interprocess communication mechanism via the **wait()**, **notify ()**, and **notifyAll()** methods.
- These methods are implemented as final methods in Object, so all classes have them.
- All three methods can be called only from within a **synchronized context**.

wait() - wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()** or **notifyAll()**.

notify() - notify() wakes up a thread that called wait() on the same object.

notifyAll() - notifyAll() wakes up all the threads that called wait() on the same object. One of the threads will be granted access.

Example: Producer and Consumer Problem

Producer produces an item and consumer consumes an item produced by the producer immediately. Producer should not produce any item until the consumer consumes the item. Consumer should wait until producer produces a new item.

```
class Q
{
    int n;
    boolean valueSet = false;
    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            }
            catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got: " + n);
        valueSet = false;
    }
}
```

```
        notify();
        return n;
    }
    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            }
            catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
class Producer implements Runnable
{
    Q q;
    Producer(Q q)
    {
        this.q = q;
        Thread t = new Thread(this, "Producer");
        t.start();
    }
    public void run()
    {
        int i = 0;
        while(i<10)
        {
            q.put(i++);
        }
    }
}
```



```

}
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {
        this.q = q;
        Thread t = new Thread(this, "Consumer");
        t.start();
    }
    public void run()
    {
        int i = 0;
        while(i < 10)
        {
            q.get();
        }
    }
}

class PC
{
    public static void main(String args[])
    {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);
    }
}

```

```
E:\JP>javac PC.java
```

```
E:\JP>java PC
```

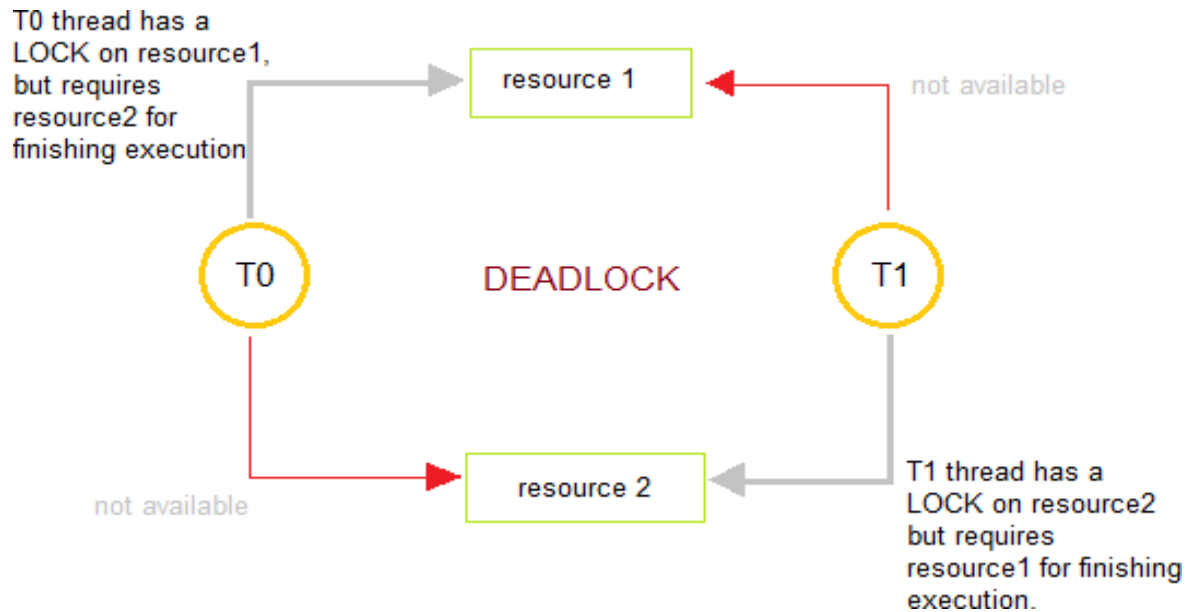
```

Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
Put: 8
Got: 8
Put: 9
Got: 9

```

Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.



Example:

```
public class DeadlockThread {
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();

    public static void main(String args[]) {
        ThreadDemo1 T1 = new ThreadDemo1();
        ThreadDemo2 T2 = new ThreadDemo2();
        T1.start();
        T2.start();
    }

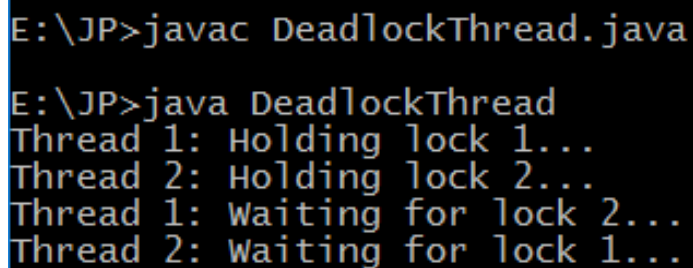
    private static class ThreadDemo1 extends Thread {
        public void run() {
            synchronized (Lock1) {
```

```
        System.out.println("Thread 1: Holding lock 1...");

        try { Thread.sleep(10); }
        catch (InterruptedException e) {}
        System.out.println("Thread 1: Waiting for lock 2...");

        synchronized (Lock2) {
            System.out.println("Thread 1: Holding lock 1 & 2...");
        }
    }
}

private static class ThreadDemo2 extends Thread {
    public void run() {
        synchronized (Lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (Lock1) {
                System.out.println("Thread 2: Holding lock 1 & 2...");
            }
        }
    }
}
```



```
E:\JP>javac DeadlockThread.java
E:\JP>java DeadlockThread
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: waiting for lock 1...
```

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful.

- ✓ Java 1.0 has methods for suspending, resuming and stopping threads.

Suspend() -> to pause a thread

Resume() -> to restart a thread

Stop() -> To stop the execution of a thread

- ✓ But these methods are inherently unsafe.
- ✓ Due to this, from Java 2.0, these methods are deprecated (available, but not recommended to use them).

Example

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 3; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
```

```
                while(suspendFlag) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

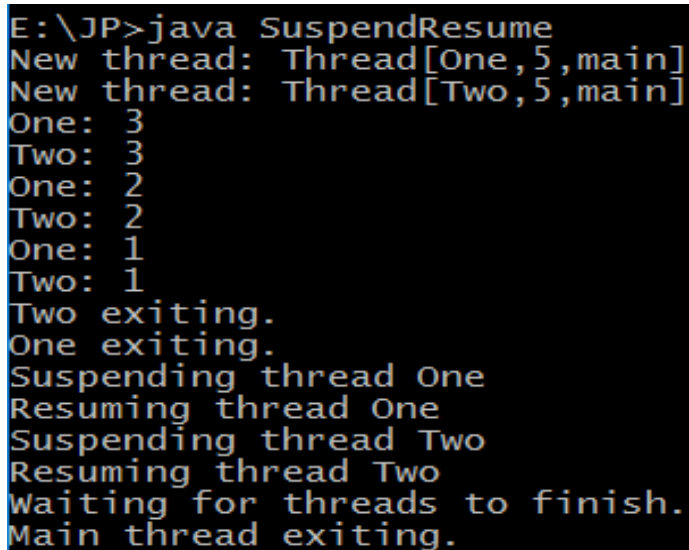
        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");

            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");

            ob2.mysuspend();
```

```
        System.out.println("Suspending thread Two");
        Thread.sleep(1000);
        ob2.myresume();
        System.out.println("Resuming thread Two");
    }
    catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
    }

    // wait for threads to finish
    try {
        System.out.println("Waiting for threads to finish.");
        ob1.t.join();
        ob2.t.join();
    }
    catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
}
}
```



```
E:\JP>java SuspendResume
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
Two exiting.
One exiting.
Suspending thread One
Resuming thread One
Suspending thread Two
Resuming thread Two
Waiting for threads to finish.
Main thread exiting.
```

Input and Output (I/O)

stream

- Java programs perform I/O through **streams**.
- A **stream** is an abstraction that either produces or consumes information.
- A **stream** is linked to a physical device by the Java I/O system.
- Java implements **streams** within class hierarchies defined in the **java.io** package.
- Streams are two types:
 - Byte streams
 - Character Streams

Byte Streams

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

The Byte Stream classes

InputStream -> **abstract class**
OutputStream -> **abstract class**
BufferedInputStream
BufferedOutputStream
ByteArrayInputStream
ByteArrayOutputStream
DataInputStream
DataOutputStream
PrintStream
RandomAccessFile

Character Streams

Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

Character Stream classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams.

Reader -> **abstract class**

Writer -> **abstract class**

BufferedReader

BufferedWriter

CharArrayReader

CharArrayWriter

FileReader

FileWriter

InputStreamReader -> translates bytes to characters

OutputStreamWriter -> translates characters to bytes

PrintWriter -> output stream contains print() and println() methods

The Predefined Streams

System class defines three predefined streams – **in, out, err**

- 1) **System.in** is an object of type **InputStream**
- 2) **System.out** and **System.err** are objects of type **PrintStream**.
- 3) **System.in** refers to standard input, which is the keyboard by default.
- 4) **System.out** refers to the standard output stream.
- 5) **System.err** refers to the standard error stream, which also is the console by default.

Reading Console Input – reading characters & Strings

- ➔ In Java, console input is accomplished by reading from **System.in**.
- ➔ To obtain a characterbased stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.


```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

→ To read a character from a `BufferedReader`, use **`read()`** method.

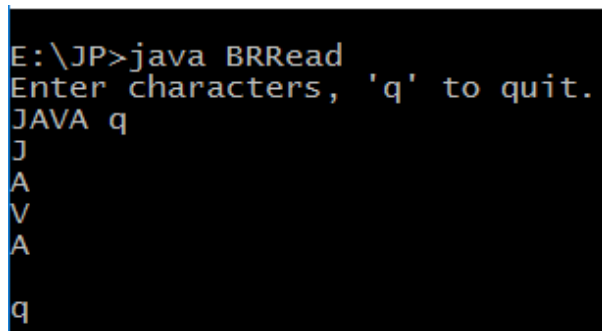
`int read() throws IOException`

→ Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when the end of the stream is encountered.

Example

// Use a `BufferedReader` to read characters from the console.

```
import java.io.*;
class BRRead {
    public static void main(String args[]) throws IOException {
        char c;
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```



```
E:\JP>java BRRead
Enter characters, 'q' to quit.
JAVA q
J
A
V
A
q
```

Applet Fundamentals

An applet is a GUI based program. Applets are event –driven programs. **applets do not contain main() method**

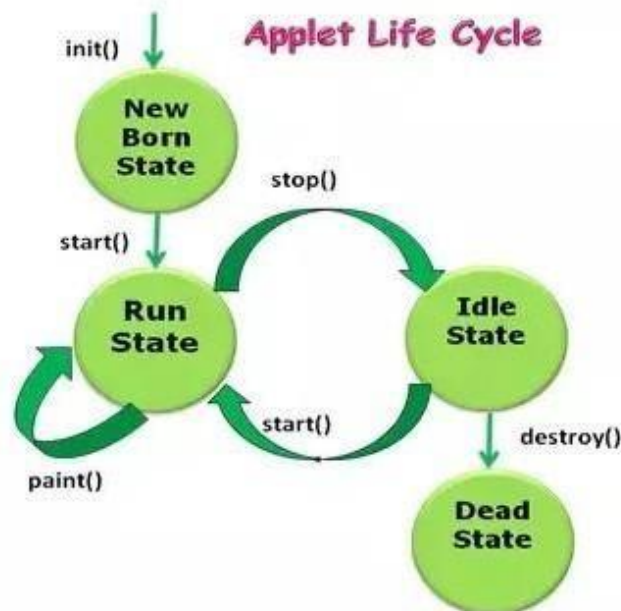
Two types:

- AWT based
- SWING based

- applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and executed by Java compatible **web browser** or **appletviewer**.

Applet Lifecycle / Applet Skeleton

- When an applet begins, the following methods are called, in this sequence:
 1. init()
 2. start()
 3. paint()
- When an applet is terminated, the following sequence of method calls takes place:
 1. stop()
 2. destroy()



init() : The init() method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start() : The start() method is called after init(). It is also called to restart an applet after it has been stopped. start() is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start().

stop(): The stop() method is called when a web browser leaves the HTML document containing the applet—when it goes to another page.

destroy(): The destroy() method is called when the environment determines that your applet needs to be removed completely from memory. The stop() method is always called before destroy().

AppletSkel.java

// An Applet skeleton.

```
import java.awt.*;
```

```
import java.applet.*;
```

```
public class AppletSkel extends Applet {
```

```
    String s;
```

```
    // Called first.
```

```
    public void init() {
```

```
        // initialization
```

```
        s = "WELCOME TO JAVA APPLET";
```

```
    }
```

```
    /* Called second, after init(). Also called whenever the applet is restarted. */
```

```
    public void start() {
```

```
        // start or resume execution
```

```
        System.out.println("START");
```

```
    }
```

```
    // Called when the applet is stopped.
```

```
    public void stop() {
```

```
        // suspends execution
```

```
        System.out.println("STOPPED");
```

```
    }
```

```
    /* Called when applet is terminated. This is the last method executed. */
```

```
    public void destroy() {
```

```
        // perform shutdown activities
```

```
        System.out.println("DESTROY");
```

```
    }
```

```
    // Called when an applet's window must be restored.
```

```
    public void paint(Graphics g) {
```

```
        // redisplay contents of window
```

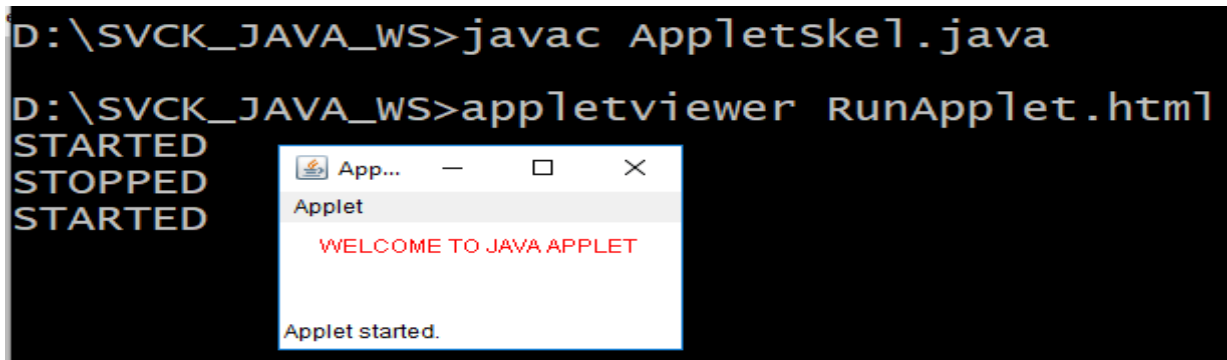
```

        g.setColor(Color.red);
        g.drawString(s,20,20);
    }
}

```

RunApplet.html

```
<applet code="AppletSkel.class" width=200 height=60> </applet>
```



Applet Program with Parameters

```
import java.awt.*;
```

```
import java.applet.*;
```

```

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        String myFont = getParameter("font");
        String myString = getParameter("string");
        int mySize    = Integer.parseInt(getParameter("size"));
        Font f = new Font(myFont, Font.BOLD, mySize);
        g.setFont(f);
        g.setColor(Color.red);
        g.drawString(myString, 20, 20);
    }
}

```

RunApplet.html

```

<applet code="SimpleApplet.class" width=200 height=60>
  <PARAM NAME="font"  VALUE="Dialog">

```

```
<PARAM NAME="size"  VALUE="24">
<PARAM NAME="string" VALUE="Hello, world...It's Java Applet">
</applet>
```



Enumerations

- An enumeration is a list of named constants.
- Java enumerations are class types.
- Each enumeration constant is an object of its enumeration type.
- Each enumeration constant has its own copy of any instance variables defined by the enumeration.
- All enumerations automatically inherit one: **java.lang.Enum**.

// An enumeration of apple varieties.

```
enum Apple {
    A, B, C, D, E
}
```

- The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- Each is implicitly declared as a public, static, final member of Apple.
- In the language of Java, these constants are called **self-typed**.

Built-in Methods of ENUM

2 methods: values() and valueOf()

The values() and valueOf() Methods

All enumerations automatically contain two predefined methods: **values()** and **valueOf()**.

Their general forms are shown here:

```
public static enum-type [ ] values( )
public static enum-type valueOf(String str )
```

- ➔ The **values()** method returns an array that contains a list of the enumeration constants.
- ➔ The **valueOf()** method returns the enumeration constant whose value corresponds to the string passed in str.

Enum Example:

// An enumeration of apple varieties.

```
enum Apple {
    A, B, C, D, E
}

class EnumDemo {
    public static void main(String args[]) {
        Apple ap;
        System.out.println("Here are all Apple constants:");
        // use values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);
        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

```
Here are all Apple constants:
A
B
C
D
E

ap contains D
```

Java enumerations with the constructors, instance variables and method

// Use an enum constructor, instance variable, and method.

```
enum Apple {
    A(10), B(9), C(12), D(15), E(8);

    private int price; // price of each apple
```

```

// Constructor
Apple(int p) {
    price = p;
}
int getPrice() {
    return price;
}

}

class EnumConsDemo {
    public static void main(String args[]) {
        Apple ap;

        // Display price of B.
        System.out.println("B costs " + Apple.B.getPrice() + " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() + " cents.");
    }
}

```

```

B costs 9 Rupees.

All apple prices:
A costs 10 Rupees.
B costs 9 Rupees.
C costs 12 Rupees.
D costs 15 Rupees.
E costs 8 Rupees.

```

Type Wrappers (Autoboxing and autounboxing)

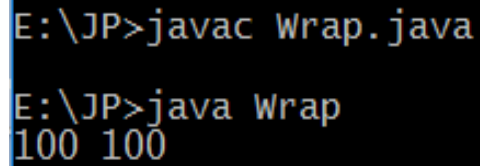
- ➔ Type wrappers are classes that encapsulate a primitive type within an object.
- ➔ The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character, and Boolean**.
- ➔ Type wrappers are related directly to **auto-boxing / auto-unboxing** feature.

Type Wrapper	Constructor	Method name
Character	Character(char ch)	char charValue()
Boolean	Boolean(boolean boolValue) Boolean(String boolString)	boolean booleanValue()
Numeric Type Wrappers		
Byte	Byte(int num)	byte byteValue()
Short	Short(int num)	short shortValue()
Long	Long(int num), Long(String str)	long longValue()
Float	Float(float num)	float floatValue()

Double	Double(double num)	double doubleValue()
--------	--------------------	-----------------------

// Demonstrate a type wrapper.

```
class Wrap {
    public static void main(String args[]) {
        // boxing
        Integer iOb = new Integer(100);
        //unboxing
        int i = iOb.intValue();
        System.out.println(i + " " + iOb);
    }
}
```



```
E:\JP>javac Wrap.java
E:\JP>java Wrap
100 100
```

boxing: The process of encapsulating a value within an object is called **boxing**.

Unboxing: The process of extracting a value from a type wrapper is called **unboxing**.

Autoboxing and Autounboxing

Beginning with JDK 5, Java does this **boxing** and **unboxing** automatically through auto-boxing and auto-unboxing features.

Autoboxing

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper.

Autounboxing

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper.

Autoboxing/Autounboxing- benefits

1. No manual boxing and unboxing values.
2. It prevents errors
3. It is very important to generics, which operate only on objects.
4. autoboxing makes working with the Collections Framework.

Where it Works

- 1) In assignments
- 2) In Method arguments and return types
- 3) In Expressions
- 4) In switch statement and any loop statements

1) Autoboxing in assignments

```
Integer iOb = 100; // autobox an int
```

```
int i = iOb; // auto-unbox
```

2) Autoboxing in Methods

```
int m(Integer v) {  
    return v ; // auto-unbox to int  
}
```

3) Autoboxing in expressions

```
Integer iOb, iOb2;  
int i;  
iOb = 100;  
++iOb;
```

```
System.out.println("After ++iOb: " + iOb);    // output: 101
```

```
iOb2 = iOb + (iOb / 3);  
System.out.println("iOb2 after expression: " + iOb2); // output: 134
```

4) Autoboxing in switch and loops

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("one");  
    break;  
    case 2: System.out.println("two");  
    break;  
    default: System.out.println("error");  
}
```

Annotations

Definition:

Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

In Java, Annotations can be **Built-in annotations** or **Custom annotations / user-defined annotations**.

Java's Built-in Annotations / Predefined Annotation Types

→ Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

1) @Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

2) Deprecated

@Deprecated annotation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

3) SuppressWarnings

@SuppressWarnings annotation is used to suppress warnings issued by the compiler.

→ Built-In Java Annotations used in other annotations are called as meta-annotations. There are several **meta-annotation types** defined in **java.lang.annotation**.

- @Target
- @Retention
- @Inherited
- @Documented

1) @Target

- a) `@Target` annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.
- b) A target annotation specifies one of the following element types as its value:
 - ElementType.FIELD** can be applied to a field or property.
 - ElementType.LOCAL_VARIABLE** can be applied to a local variable.
 - ElementType.METHOD** can be applied to a method-level annotation.
 - ElementType.PACKAGE** can be applied to a package declaration.
 - ElementType.PARAMETER** can be applied to the parameters of a method.
 - ElementType.TYPE** can be applied to any element of a class.

2) @Retention

`@Retention` annotation specifies how the marked annotation is stored:

- **RetentionPolicy.SOURCE** – The marked annotation is retained only in the source level and is ignored by the compiler.
- **RetentionPolicy.CLASS** – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- **RetentionPolicy.RUNTIME** – The marked annotation is retained by the JVM so it can be used by the runtime environment.

3) @Inherited

`@Inherited` annotation indicates that the annotation type can be inherited from the super class.

4) @Documented

`@Documented` annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.

Built-in Annotations Example

```
class Animal{
    void eat(){
        System.out.println("eating something");
    }
}

class Dog extends Animal{
    @Override
    void Eat(){
```

```

        System.out.println("eating foods");
    } //should be eatSomething
}
class TestAnnotation1{
    public static void main(String args[]){
        Animal a=new Dog();
        a.eat();
    }
}

```

```

E:\JP>javac TestAnnotation1.java
TestAnnotation1.java:9: error: method does not override or implement a method from a supertype
@Override
^
1 error

```

Custom Annotations / User defined annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The `@interface` element is used to declare an annotation. For example:

```

@interface MyAnnotation
{
}

```

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```

@interface MyAnnotation{ }

```

The `@Override` and `@Deprecated` are marker annotations.

2) Single-value annotation

An annotation that has one method, is called single-value annotation. We can provide the default value also.

For example:

```
@interface MyAnnotation{
    int value() default 0;
}
```

3) Multi-value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation{
    int value1();
    String value2();
    String value3();
}
```

Example of Custom Annotation (creating, applying and accessing annotation)

```
//Creating custom annotation
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
    int value();
}

//Applying annotation
class Hello{
    @MyAnnotation(value=10)
    public void sayHello(){
        System.out.println("hello annotation");
    }
}

//Accessing annotation
class TestCustomAnnotation{
    public static void main(String args[])throws Exception{

        Hello h=new Hello();
```

```

        Method m=h.getClass().getMethod("sayHello");

        MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
        System.out.println("value is: "+manno.value());
    }
}

```

```

E:\JP>java TestCustomAnnotation
value is: 10

```

Generics

- ➔ Generics are called as parameterized types.
- ➔ The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.
- ➔ Generics can be applied to a class, interface or a method.
- ➔ Generics Work Only with Reference Types.

Advantages of Java Generics

1) Type-safety:

We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Compile-Time Checking:

It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

3) Enabling programmers to implement generic algorithms

Generic class

The General Form of a Generic Class

```
class class-name<type-param-list > { // ...
```

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

```

class MyGen<T>{
    T obj;
    void add(T obj){
        this.obj=obj;
    }
}

```

```

    }
    T get(){
        return obj;
    }
}
class TestGenerics{
    public static void main(String args[]){
        MyGen<Integer> m=new MyGen<Integer>();
        m.add(2);
        //m.add("vivek");//Compile time error
        System.out.println(m.get());
    }
}

```

```

E:\JP>java TestGenerics
2

```

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

```

public class GenericMethod{
    public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A'};

        System.out.println( "Printing Integer Array" );
        printArray( intArray );

        System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}

```

```

E:\JP>java GenericMethod
Printing Integer Array
10
20
30
40
50

Printing Character Array
J
A
V
A

```

Generic Constructor

It is possible for constructors to be generic, even if their class is not.

// Use a generic constructor.

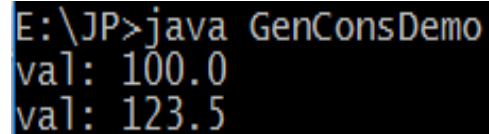
```
class GenCons {
    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }
    void showval() {
        System.out.println("val: " + val);
    }
}

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

        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);

        test.showval();
        test2.showval();
    }
}
```



```
E:\JP>java GenConsDemo
val: 100.0
val: 123.5
```

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces.

interface interface-name<type-param-list> { // ...

class class-name<type-param-list> implements interface-name<type-arg-list> {

```
    interface GenInterface<T> {
        void test(T a);
        T get();
    }

    class MyGen<T> implements GenInterface<T> {
    }
```

Important Questions

Part-A

1. What is main thread?
2. Explain thread priorities.

3. What is context switch
4. Define thread group and thread priority.
5. What are the benefits of Java's multithreaded programming.
6. Define Generics? Explain its advantages.
7. Define autoboxing and unboxing.
8. Define Annotation.
9. Explain PrintWriter class
10. Define Stream? Explain its types.

Part-B

1. Explain Java Thread Model and Thread states (transitions)
(OR)
What is Multithreaded programming? Explain thread states and how to obtain the thread state.
2. What is Thread class and explain its methods: getName(), getPriority(), isAlive(), join(), run(), sleep(), start(), currentThread().
3. What is multitasking? What are the types of multitasking? Write the differences between **process based multitasking** and **thread based multitasking**.
4. Explain how to create a thread or multiple threads with an example.
5. What is synchronization? What is the need for synchronization? Explain with a suitable example how to synchronize the code. (Account example)
6. In Java, how interthread communication happens. Explain with an example.
[OR]
Explain producer and consumer problem using Interthread communication with an example.
[OR]
Explain the interthread communication methods - **wait()**, **notify()** and **notifyAll()**.
7. What is Deadlock? Explain with an example.
8. Explain how to perform Suspending, Resuming and stopping threads in Java.
9. Define Stream? Explain the different types of streams.
[OR]
What is Stream? Explain **byte stream** and **character stream** classes.
10. Explain with an example how to perform the reading the console input and writing the console output.
11. Explain with an example how to read, write and closing of files (automatically).
12. What is an Applet? Explain the lifecycle of an applet.
[OR]
Explain a simple Applet program with an example. Also explain how to compile and execute an applet program
13. Define Generics? Explain the general form of a generic class and write its advantages.
14. Explain Generic Method and Generic Interfaces.
15. Define Enumerations? Explain with an example.
16. What is a Type wrapper? Explain Autoboxing and unboxing with an example.
17. Write short notes on Annotations.

UNIT – IV – END

Object Oriented Programming using JAVA

UNIT V

AWT

awt was the first GUI framework in JAVA since 1.0. It defines numerous classes and methods to create windows and simple control such as Labels, Buttons, and TextFields etc.

Delegation Event Model

The modern approach to handle events is based on delegation event model.

Delegation event model consists of only two parts – **Source** and **Listeners**.

Concept

A source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event.

Once an event is received, the listener processes the event and then returns. In this model, Listeners must register with a source in order to receive an event notification.

Events

An event is an object that describes a state change in a source.

Ex: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources

A source is an object that generates an event. Sources may generate more than one type of event.

Ex: mouse, keyboard

Event Listeners

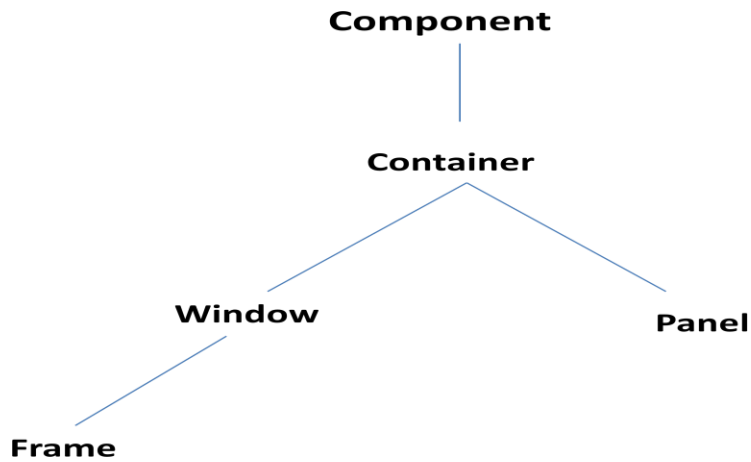
A listener is an object that is notified when an event occurs. It has two major requirements.

- 1) It must have been registered with one or more sources to receive notifications about specific types of events.
- 2) It must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those

found in **java.awt.event**.

AWT classes & Hierarchy



Component

Component is an abstract class that encapsulates all of the attributes of a visual component.

Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

Container

The Container class is a subclass of Component. Other Container objects can be stored inside of a Container

A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

Panel

The Panel class is a concrete subclass of Container. Panel is the superclass for Applet. a Panel is a window that does not contain a title bar, menu bar, or border.

Window

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop.

Frame

It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Creating Windows (Frame based and Applet based)

Windows are created by using applet and by using Frame.

Windows can be created :

1. By extending Applet class
2. By extending Frame class
 - a. Frame class is used to create child windows within applets
 - b. Top-level or child-windows of stand-alone applications

Frame constructor

Frame() throws HeadlessException

Frame(String title) throws HeadlessException

A **HeadlessException** is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.

Key methods in Frame windows

1. Setting the Window's Dimensions

void setSize(int newWidth, int newHeight)

void setSize(Dimension newSize)

2. Get the window's size

Dimension getSize()

3. Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call setVisible().

void setVisible(boolean visibleFlag)

The component is visible if the argument to this method is true. Otherwise, it is hidden.

4. Setting a Window's Title

void setTitle(String newTitle)

5. Closing a Frame Window

To close the window, We must implement the **windowClosing()** method of the WindowListener interface. Inside windowClosing(), we must remove the window from the screen by calling setVisible(false).

Creating a Frame window in an AWT-based Applet

Creating a new frame window from within an AWT-based applet is actually quite easy.

- a) First, create a subclass of Frame.
- b) Next, override any of the standard applet methods, such as `init()`, `start()`, and `stop()`, to show or hide the frame as needed.
- c) Finally, implement the `windowClosing()` method of the `WindowListener` interface, calling `setVisible(false)` when the window is closed.

AppletFrame.java

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class AppletFrame extends Applet
{
    Frame f;

    public void init(){
        f = new SampleFrame("A frame window");
        f.setSize(250,250);
    }
    public void start()
    {
        f.setVisible(true);
    }
    public void stop()
    {
        f.setVisible(false);
    }
    public void paint(Graphics g)
    {
        g.drawString("Applet window",10,20);
    }
}
class SampleFrame extends Frame
{
    SampleFrame(String title)
    {
        setTitle(title);

        // remove the window when closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {

                setVisible(false);
            }
        });
    }
}
```

```

    }
    public void paint(Graphics g)
    {
        g.drawString("This is in Frame windiw",10,40);
    }
}

```

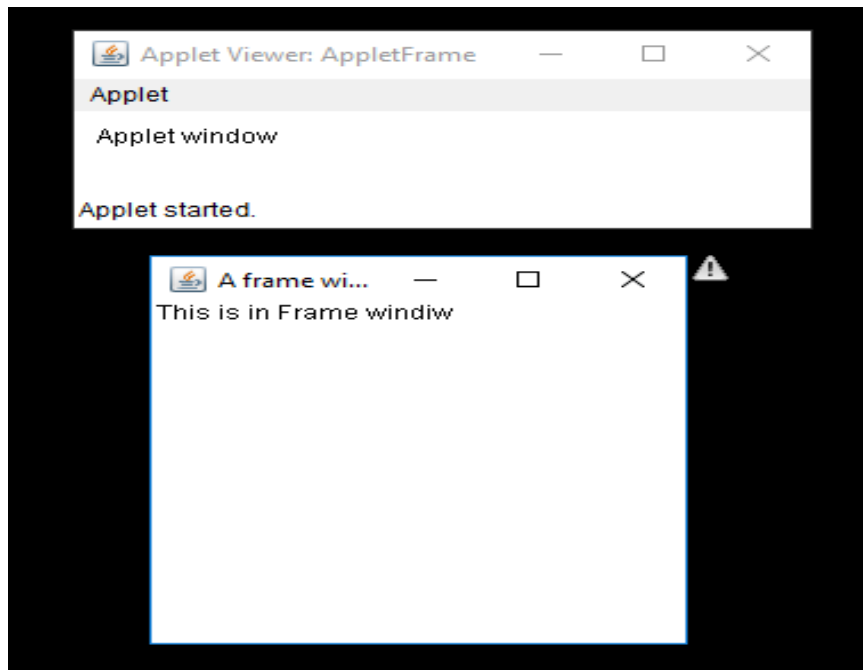
RunAppletFrame.html

```

<applet code ="AppletFrame" width=300 height=50>
</applet>

```

Output:



Creating a window using Frame class – AWT-window

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class FrameWindow extends Frame
{
    FrameWindow(String title)
    {
        setTitle(title);

        // remove the window when closed
        addWindowListener(new WindowAdapter() {

```

```

        public void windowClosing(WindowEvent we) {

            //setVisible(false);
            System.exit(0);

        }

    });

}

public void paint(Graphics g)
{
    g.drawString("This is in Frame windiw",50,100);
}

public static void main(String ar[])
{
    FrameWindow fw = new FrameWindow("An AWT based application");

    fw.setSize(100,200);
    fw.setVisible(true);
}
}

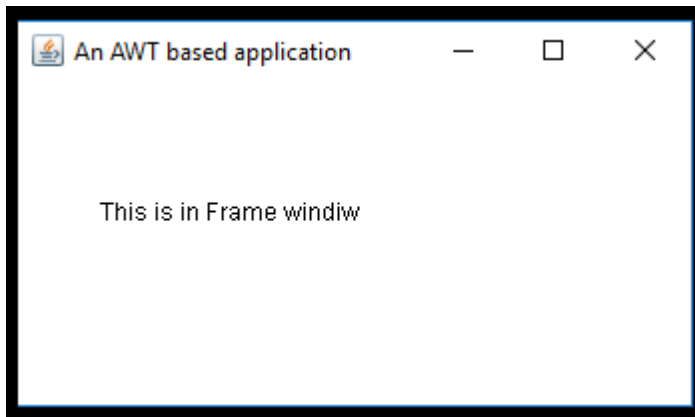
```

Output:

```

javac FrameWindow.java
java FrameWindow

```

**Handling Events in a Frame Window using AWT based Applet**

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class AppletFrameEvents extends Applet
{
    Frame f;

```



```

    public void init(){
        f = new SampleFrame("A frame window");
        f.setSize(250,250);
    }
    public void start()
    {
        f.setVisible(true);
    }
    public void stop()
    {
        f.setVisible(false);
    }
    public void paint(Graphics g)
    {
        g.drawString("Applet window",10,20);
    }
}
class SampleFrame extends Frame implements MouseMotionListener
{
    String msg = " ";
    int mouseX=10,mouseY=40;
    int movX=0,movY=0;
    SampleFrame(String title)
    {
        setTitle(title);

        // remove the window when closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {

                setVisible(false);
            }
        });

        addMouseMotionListener(this);
    }

    public void mouseDragged(MouseEvent me) {
        mouseX = me.getX();
        mouseY = me.getY();

        movX = me.getX();
        movY = me.getY();
        msg = "Mouse Dragged";
        repaint();
    }

    public void mouseMoved(MouseEvent me) {
        mouseX = me.getX();
        mouseY = me.getY();
    }
}

```

```

        movX = me.getX();
        movY = me.getY();
        msg = "Mouse Moved";
        repaint();
    }

    public void paint(Graphics g)
    {
        g.drawString(msg,mouseX,mouseY);
        g.drawString("Mouse at "+ movX +", "+movY,10,40);
    }
}

```

Handling Events in a Frame Window using AWT

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class FrameWindowEvents extends Frame implements MouseMotionListener
{
    String msg = " ";
    int mouseX=10,mouseY=40;
    int movX=0,movY=0;
    FrameWindowEvents (String title)
    {
        setTitle(title);

        // remove the window when closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        addMouseMotionListener(this);
    }

    public void mouseDragged(MouseEvent me) {
        mouseX = me.getX();
        mouseY = me.getY();

        movX = me.getX();
        movY = me.getY();
        msg = "Mouse Dragged";
    }
}

```

```
        repaint();
    }

    public void mouseMoved(MouseEvent me) {
        mouseX = me.getX();
        mouseY = me.getY();

        movX = me.getX();
        movY = me.getY();
        msg = "Mouse Moved";
        repaint();
    }

    public void paint(Graphics g)
    {
        g.drawString(msg,mouseX,mouseY);
        g.drawString("Mouse at "+ movX +", "+movY,10,40);
    }

    public static void main(String ar[])
    {
        FrameWindowEvents fw = new FrameWindowEvents("An AWT based
application");

        fw.setSize(100,200);
        fw.setVisible(true);
    }
}
```

Graphics Class

Graphics class has methods which are used to display the information within a window.

Methods in Graphics class

1. Drawing Strings

drawstring(String msg,int x, int y)

2. Drawing Lines

void drawLine(int startX, int startY, int endX, int endY)

3. Drawing Rectangles

The drawRect() and fillRect() methods display an outlined and filled rectangle, respectively.

```
void drawRect(int left, int top, int width, int height)
void fillRect(int left, int top, int width, int height)
```

```
void drawRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)
void fillRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)
```

4. Drawing Ellipses and Circles

To draw an ellipse, use `drawOval()`. To fill an ellipse, use `fillOval()`.

```
void drawOval(int left, int top, int width, int height)
void fillOval(int left, int top, int width, int height)
```

5. Drawing Arcs

```
void drawArc(int left, int top, int width, int height, int startAngle,int sweepAngle)
void fillArc(int left, int top, int width, int height, int startAngle,int sweepAngle)
```

The arc is drawn counterclockwise if `sweepAngle` is positive, and clockwise if `sweepAngle` is negative.

6. Drawing Polygons

- It is possible to draw arbitrarily shaped figures using `drawPolygon()` and `fillPolygon()`:

```
void drawPolygon(int x[ ], int y[ ], int numPoints) void fillPolygon(int x[ ], int y[ ], int numPoints)
```

- The polygon's endpoints are specified by the coordinate pairs contained within the `x` and `y` arrays. The number of points defined by these arrays is specified by `numPoints`.

Program to demonstrate Graphics class methods – Drawing methods

// Draw graphics elements.

```
import java.awt.*;
import java.applet.*;
```

```
public class GraphicsDemo extends Applet {
    public void paint(Graphics g)
    {
        // Draw lines.
        g.drawLine(0, 0, 100, 90);
        g.drawLine(0, 90, 100, 10);
        g.drawLine(40, 25, 250, 80);

        // Draw rectangles.
```

```

g.drawRect(10, 150, 60, 50);
g.fillRect(100, 150, 60, 50);
g.drawRoundRect(190, 150, 60, 50, 15, 15);
g.fillRoundRect(280, 150, 60, 50, 30, 40);

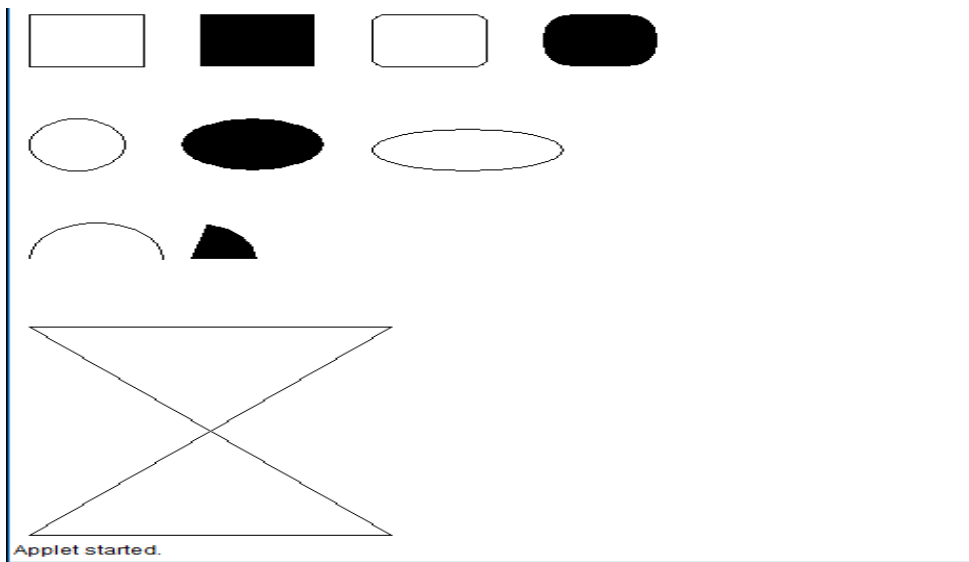
// Draw Ellipses and Circles
g.drawOval(10, 250, 50, 50);
g.fillOval(90, 250, 75, 50);
g.drawOval(190, 260, 100, 40);

// Draw Arcs
g.drawArc(10, 350, 70, 70, 0, 180);
g.fillArc(60, 350, 70, 70, 0, 75);

// Draw a polygon
int xpoints[] = {10, 200, 10, 200, 10};
int ypoints[] = {450, 450, 650, 650, 450};
int num = 6;

g.drawPolygon(xpoints, ypoints, num);
    }
}

```

Output:**Color class**

Java supports color in a portable, device-independent fashion. The AWT color system allows us to specify any color we want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet.

We can create our own color by using the below constructor:

`Color(int red, int green, int blue)`

It takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

Once we have created a color, we can use it to set the foreground and/or background color by using the **setForeground()** and **setBackground()** methods.

Program on Color class

```
import java.awt.*;  
import java.applet.*;
```

```
public class ColorAppletEx extends Applet  
{
```

```
    Color c = new Color(10,200,255);
```

```
    public void init()  
    {
```

```
        //setColor(c);
```

```
        setBackground(c);
```

```
        //setBackground(Color.red); // another way to set the background color
```

```
    }
```

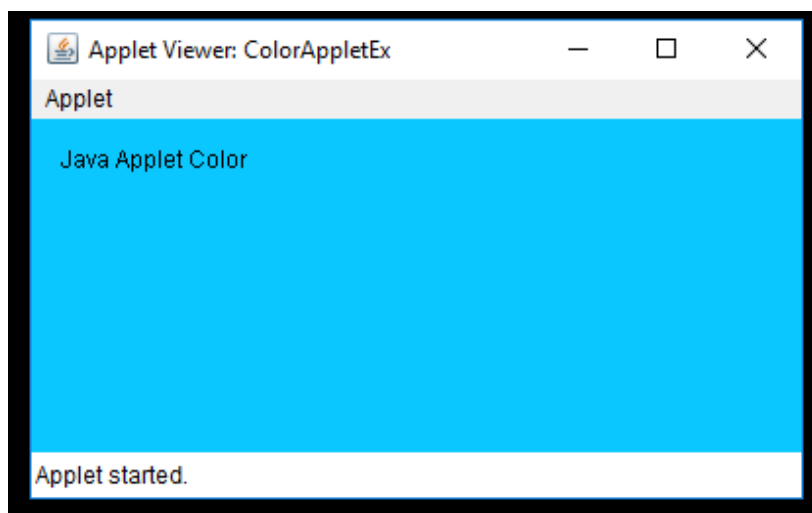
```
    public void paint(Graphics g)
```

```
    {
```

```
        g.drawString("Java Applet Color", 15,25);
```

```
    }
```

```
}
```



Font class

The AWT supports multiple type fonts. Fonts have a family name, font style and font size.

Methods in Font class

<u>Method</u>	<u>Description</u>
String getFamily()	Returns the name of the font family
String getName()	Returns the logical name of the invoking font.
int getSize()	Returns the size
int getStyle()	Returns the style of the font

Determining the Available Fonts

To obtain the available fonts on our machine, use the **getAvailableFontFamilyNames()** method defined by the **GraphicsEnvironment** class.

```
String[ ] getAvailableFontFamilyNames( )
```

In addition, the **getAllFonts()** method is defined by the GraphicsEnvironment class.

```
Font[ ] getAllFonts( )
```

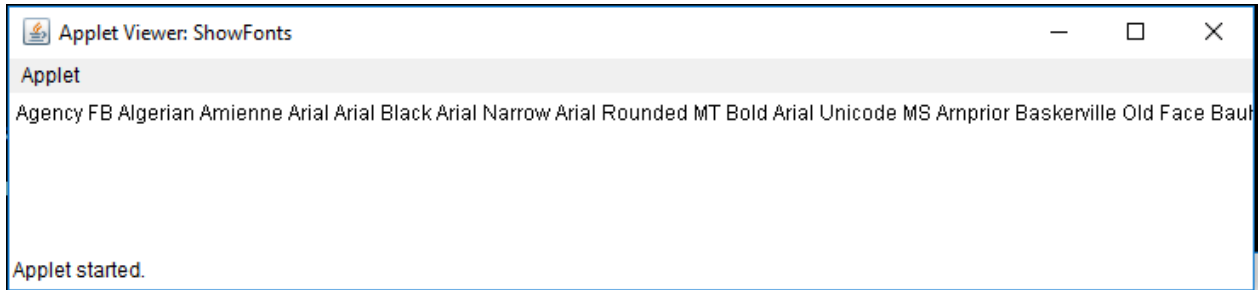
Ex: Program to obtain the names of the available font families.

```
// Display Fonts
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
    public void paint(Graphics g) {
        String msg = "";
        String FontList[];

        GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
        FontList = ge.getAvailableFontFamilyNames();
        for(int i = 0; i < FontList.length; i++)
            msg += FontList[i] + " ";

        g.drawString(msg, 4, 16);
    }
}
```

Output:**Creating and Selecting Font**

To create a new font, construct a Font object that describes that font.

Font(String fontName, int fontStyle, int pointSize)

- ➔ **fontName** specifies the name of the desired font.
- ➔ The **style** of the font is specified by fontStyle. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**.
- ➔ To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.
- ➔ To use a font that you have created, you must select it using **setFont()**, which is defined by Component. It has this general form:
void setFont(Font fontObj)

// Creating and Selecting Fonts

```
import java.applet.*;
import java.awt.*;

public class CreateFont extends Applet {
    Font f;
    String msg=" ";
    public void init()
    {
        f = new Font("Times New Roman",Font.BOLD, 12);
        msg = "JAVA FONT APPLET";
        setFont(f);
    }
    public void paint(Graphics g) {
        g.drawString(msg, 4, 16);
    }
}
```

Displaying a Font information

To obtain the information about the currently selected font, we must get the current font by calling **getFont()** method.


```

public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();
}

```

FontMetrics Class

FontMetrics class is used to get the information about a font.

Methods in FontMetrics class

Method	Description
Font getFont()	Returns the font.
int getHeight()	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int stringWidth(String str)	Returns the width of the string specified by str.
int getLeading()	Returns the space between lines of text.

// Demonstrate multiline output using FontMetrics class

```

import java.awt.*;
import java.applet.*;

public class FontMetricsEx extends Applet {
    int X=0, Y=0; // current position
    String s1 = "This is on line one.";
    String s2 = "This is on line two.";

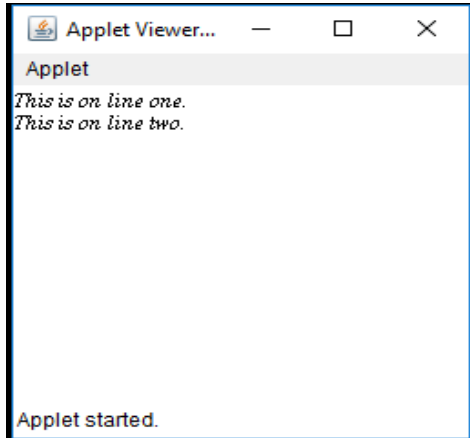
    public void init() {
        Font f = new Font("Times New Roman", Font.PLAIN, 12);
        setFont(f);
    }
    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics();

        Y += fm.getHeight(); // advance to next line
    }
}

```

```
X = 0;
g.drawString(s1, X, Y);
X = fm.stringWidth(s1); // advance to end of line

Y += fm.getHeight(); // advance to next line
X = 0;
g.drawString(s2, X, Y);
X = fm.stringWidth(s2); // advance to end of line
}
}
```



AWT Controls

AWT supports the following controls:

1. Labels
2. Push buttons
3. Check boxes
4. Check box group
5. Choice lists
6. Lists
7. Scroll bars
8. Text Editing (TextField, TextArea)

1. Labels

A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

Constructors

- ➔ Label() throws HeadlessException
-> This creates a blank label.

→ `Label(String str)` throws `HeadlessException`
 -> This creates a label that contains the string specified by `str`.

→ `Label(String str, int how)` throws `HeadlessException`

The value of ***how*** must be one of these three constants: **`Label.LEFT`**, **`Label.RIGHT`**, or **`Label.CENTER`**.

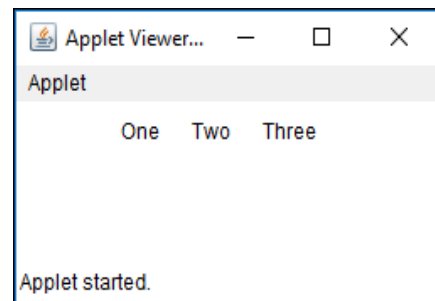
→ **`setText(String str)`** -> we can set or change the text in a label by using the `setText()` method.

→ **`getText()`** -> We can obtain the current label by calling `getText()`.

Example

```
public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");

        // add labels to applet window
        add(one);
        add(two);
        add(three);
    }
}
```



2. Buttons

→ A push button is a component that contains a label and that generates an event when it is pressed.

→ Push buttons are objects of type `Button`.

constructors

`Button()` throws `HeadlessException`

`Button(String str)` throws `HeadlessException`

→ **`setLabel(String str)`** -> this is used to set the label.

→ **`getLabel()`** -> this is used to retrieve the label

Event Handling with Buttons

- a) Each time a button is pressed, an action event is generated.
- b) The listener implements the **ActionListener** interface.
- c) **ActionListener** defines the **actionPerformed()** method, which is called when an event occurs.

// Program to demonstrate Button Event handling

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

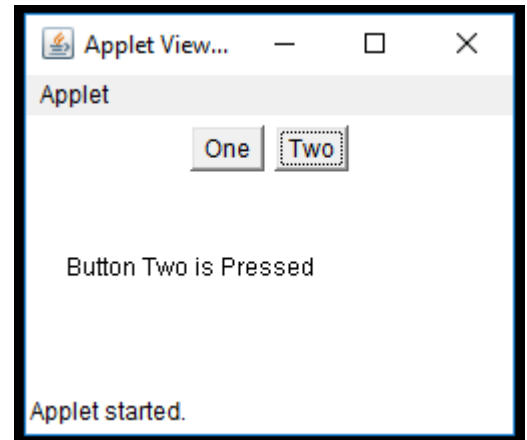
public class ButtonDemo extends Applet implements ActionListener{
    String actionName,msg="";
    public void init() {
        Button b1 = new Button("One");
        Button b2 = new Button("Two");

        // add buttons to applet window
        add(b1);
        add(b2);

        b1.addActionListener(this);
        b2.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae)
    {
        actionName=ae.getActionCommand();
        if(actionName.equals("One"))
            msg = "Button One is Pressed";
        else
            msg = "Button Two is Pressed";

        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString(msg,20,80);
    }
}
```



3. Check Boxes

- A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

constructors:

Checkbox() throws HeadlessException

Checkbox(String str) throws HeadlessException

Checkbox(String str, boolean on) throws HeadlessException

Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException

Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException

➔ If on is true, the check box is initially checked; otherwise, it is cleared.

➔ If the check box is not part of a group, then cbGroup must be null.

- **getState()** -> Used to retrieve the current state of a check box
- **setState(boolean on)** -> We can set the state.
- **getLabel()** -> obtain the current label associated with a check box
- **setLabel(String str)** -> Used to set the label.

Event Handling with Checkboxes

- 1) Each time a check box is selected or deselected, an **item event** is generated.
- 2) Each listener implements the **ItemListener** interface.
- 3) That interface defines the **itemStateChanged()** method.

// Demonstrate check boxes.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, win7;
    public void init() {
        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");

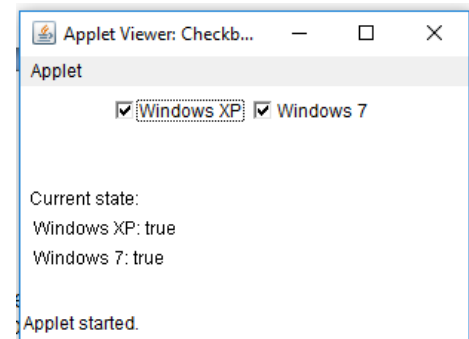
        add(winXP);
        add(win7);

        winXP.addItemListener(this);
        win7.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
```

```

        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows XP: " + winXP.getState();
        g.drawString(msg, 6, 100);
        msg = " Windows 7: " + win7.getState();
        g.drawString(msg, 6, 120);
    }
}

```



4. Checkbox Group

Check box group is used to create **radio buttons**. It is used to create a set of mutually exclusive check boxes.

Check box groups are objects of type **CheckboxGroup**.

- **getSelectedCheckbox()** -> used to determine which check box in a group is currently selected.
- **setSelectedCheckbox(Checkbox ch)** -> We can set a check box to be selected.

Example:

```

Checkbox windows, android, solaris, mac;
CheckboxGroup cbg;

```

```

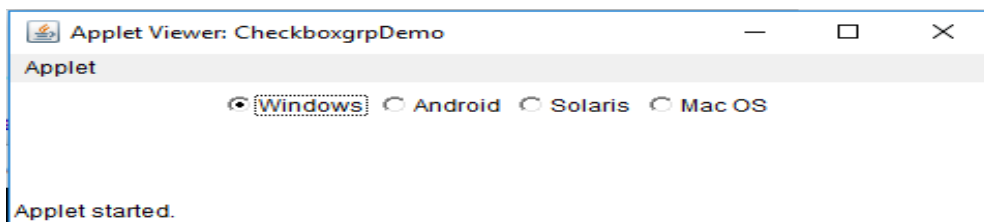
cbg = new CheckboxGroup();
windows = new Checkbox("Windows", cbg, true);
android = new Checkbox("Android", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("Mac OS", cbg, false);

```

```

add(windows);
add(android);
add(solaris);
add(mac);

```



Event Handling with Checkbox Groups

- 1) Each time a check box is selected or deselected, an **item event** is generated.
- 2) Each listener implements the **ItemListener** interface.
- 3) That interface defines the **itemStateChanged()** method.

5. Choice Controls

- The Choice class is used to create a *pop-up list of items from which the user may choose*.
- Thus, a Choice control is a form of menu.
- **getSelectedItem()** -> returns a string containing the name of the item
- **int getSelectedIndex()** -> returns the index of the item, The first item is at index 0. By default, the first item added to the list is selected.
- **getItemCount()** -> obtain the number of items in the list.

Event Handling with Choice control

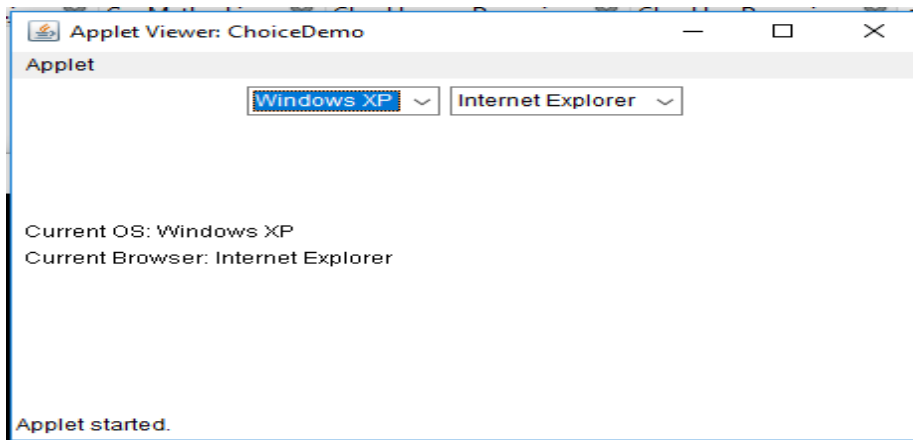
- 1) Each time a choice is selected, an item event is generated.
- 2) Each listener implements the **ItemListener** interface.
- 3) That interface defines the **itemStateChanged()** method.

```
// Demonstrate choice control
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";
    public void init() {
        os = new Choice();
        browser = new Choice();
        // add items to os list
        os.add("Windows XP");
        os.add("Windows 7");
        os.add("Solaris");
        os.add("Mac OS");
        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
    }
}
```

```

        browser.add("Opera");
        // add choice lists to window
        add(os);
        add(browser);
        // register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
}
// Display current
// Display current selections.
public void paint(Graphics g) {
    msg = "Current OS: ";
    msg += os.getSelectedItemId();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItemId();
    g.drawString(msg, 6, 140);
}
}

```



6. Lists

The **List** class provides a compact, multiple-choice, scrolling selection list.

Choice	List
Shows only the selected item	Shows any number of choices in the visible window
Allows single selection	Allows multiple selections
Single click generates ItemEvent	Single click generates ItemEvent and doubleclick generates ActionEvent

Constructors

- **List() throws HeadlessException** -> The first version creates a List control that allows only one item to be selected at any one time.
- **List(int numRows) throws HeadlessException** -> the value of *numRows* specifies the number of entries in the list that will always be visible
- **List(int numRows, boolean multipleSelect) throws HeadlessException** -> if *multipleSelect* is true, then the user may select two or more items at a time.

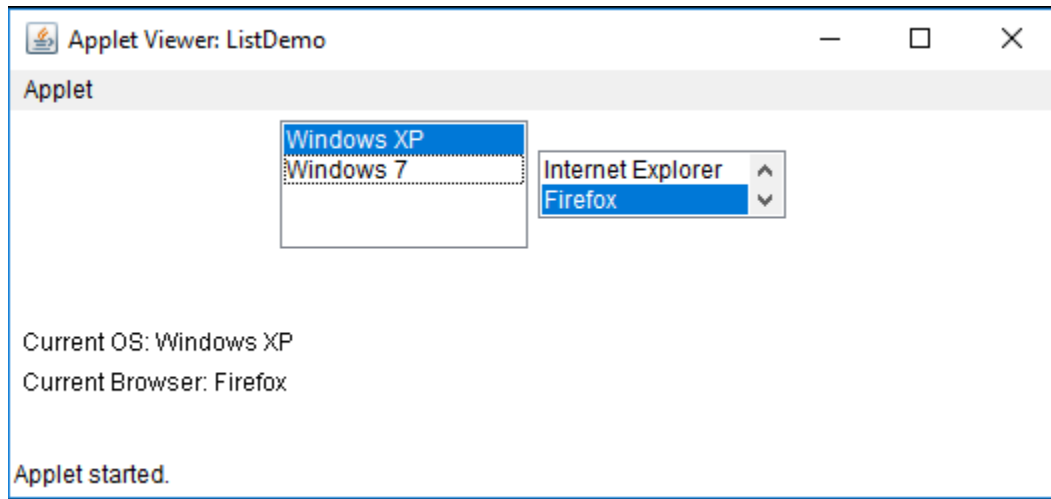
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";
    public void init() {
        os = new List(4, true);
        browser = new List(2, false);
        // add items to os list
        os.add("Windows XP");
        os.add("Windows 7");
        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");
        browser.select(1);
        // add lists to window
        add(os);
        add(browser);
        // register to receive action events
        os.addActionListener(this);
        browser.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g) {
        int idx[];
        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
            msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
```

```

        msg += browser.getSelectedItemId();
        g.drawString(msg, 6, 140);
    }
}

```



7. Scroll bars

- *Scroll bars are used to select continuous values between a specified minimum and maximum*
- *Scroll bars may be oriented horizontally or vertically.*
- *Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.*

Constructors

- a) `Scrollbar()` throws `HeadlessException` -> creates a vertical scroll bar.
- b) `Scrollbar(int style)` throws `HeadlessException`
 - If style is **`Scrollbar.VERTICAL`**, a vertical scroll bar is created.
 - If style is **`Scrollbar.HORIZONTAL`**, the scroll bar is horizontal.
- c) `Scrollbar(int style, int initialValue, int thumbSize, int min, int max)` throws `HeadlessException`

`getValue()` -> To obtain the current value of the scroll bar.

Event Handling in Scrollbars

- Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated.
- To process scroll bar events, you need to implement the **AdjustmentListener** interface.
- **getAdjustmentType()** method can be used to determine the type of the adjustment.

// Demonstrate scroll bars.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /* </applet> */

public class SBDemo extends Applet implements AdjustmentListener {
    String msg = "";
    Scrollbar vertSB, horzSB;

    public void init() {

        vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 50);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 50);
        horzSB.setPreferredSize(new Dimension(100, 20));

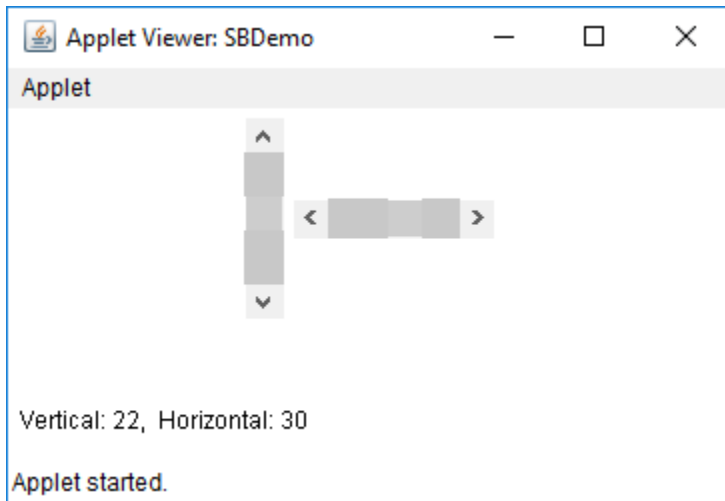
        add(vertSB);
        add(horzSB);

        // register to receive adjustment events
        vertSB.addAdjustmentListener(this);
        horzSB.addAdjustmentListener(this);
    }

    public void adjustmentValueChanged(AdjustmentEvent ae) {
        repaint();
    }

    // Display current value of scroll bars.
    public void paint(Graphics g) {
        msg = "Vertical: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();

        g.drawString(msg, 6, 160);
    }
}
```



8. Text Editing (TextField and TextArea classes)

The TextField class implements a single-line text-entry area, usually called an edit control.

TextField is a subclass of TextComponent.

Constructors

TextField() throws HeadlessException
 TextField(int numChars) throws HeadlessException
 TextField(String str) throws HeadlessException
 TextField(String str, int numChars) throws HeadlessException

Methods

getText() -> Obtain the string currently contained in the text field

getSelectedText() -> returns the selected text.

setEchoChar(char ch) -> We can disable the echoing of the characters as they are typed.

Event Handling TextField

When user presses ENTER, an action event is generated.

Example

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;

public class TextFieldDemo extends Applet implements ActionListener {

    TextField name, pass;

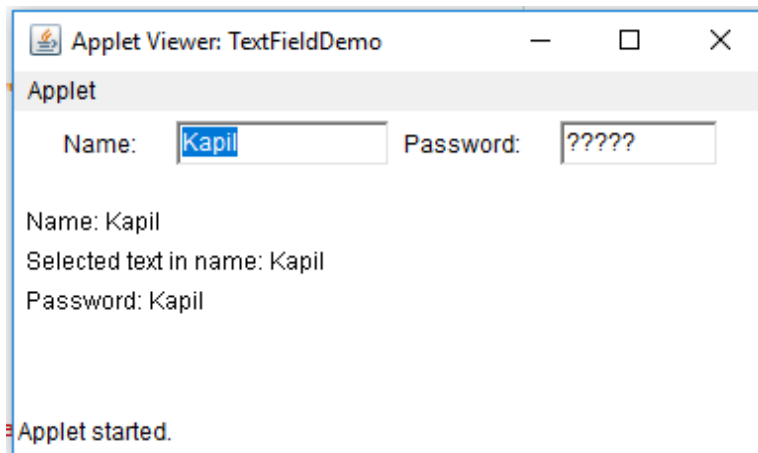
    public void init() {
        Label namep = new Label("Name: ");
        Label passp = new Label("Password: ");
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');

        add(namep);
        add(name);
        add(passp);
        add(pass);

        // register to receive action events
        name.addActionListener(this);
        pass.addActionListener(this); }

    // User pressed Enter.
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(), 6, 60);
        g.drawString("Selected text in name: " + name.getSelectedText(), 6, 80);
        g.drawString("Password: " + pass.getText(), 6, 100);
    }
}
```



TextArea

It is a multiline editor. TextArea is a subclass of TextComponent.

Constructors:

TextArea() throws HeadlessException

TextArea(int numLines, int numChars) throws HeadlessException

TextArea(String str) throws HeadlessException

TextArea(String str, int numLines, int numChars) throws HeadlessException

TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException

Methods

- **append(String str)** -> appends the string specified by str to the end of the current text.
- **insert(String str, int index)** -> Inserts the string passed in str at the specified index.
- **void replaceRange(String str, int startIndex, int endIndex)** -> It replaces the characters from startIndex to endIndex-1, with the replacement text passed in str.

// Demonstrate TextArea.

```
import java.awt.*;
```

```
import java.applet.*;
```

```
public class TextAreaDemo extends Applet {
```

```
    public void init() {
```

```
        String val = "Java 8 is the latest version of the most \n" +
```

```
                    "widely-used computer language for Internet programming.\n" +
```

```
                    "Building on a rich heritage, Java has advanced both \n" +
```

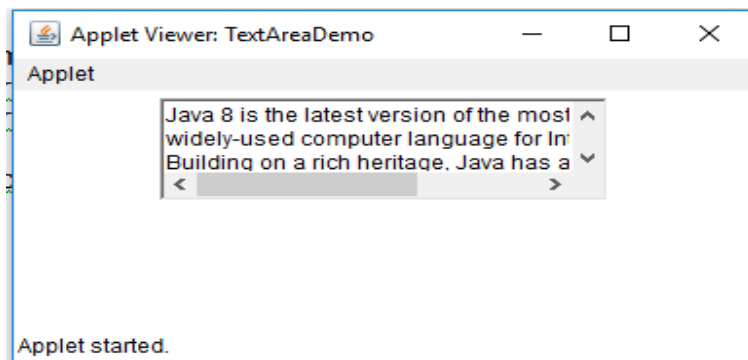
```
                    "the art and science of computer language design. \n\n";
```

```
        TextArea text = new TextArea(val, 3, 30);
```

```
        add(text);
```

```
    }
```

```
}
```



Layout Managers

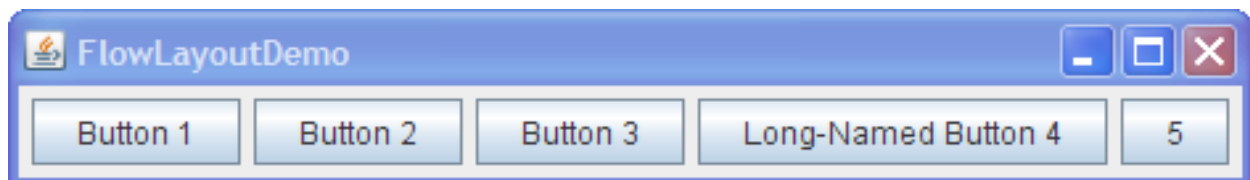
- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- Each **Container object** has a **layout manager associated with it**.
- A layout manager is an instance of any class that implements the **LayoutManager interface**.
- The layout manager is set by the **setLayout(LayoutManager lmg)** method.
- Layout managers basically do two things:
 - Calculate the minimum/preferred/maximum sizes for a container.
 - Lay out the container's children.

There are 5 layouts supported by AWT:

1. FlowLayout
2. BorderLayout
3. GridLayout
4. GridbagLayout
5. CardLayout

Flow Layout

- The FlowLayout is used to arrange the components in a line, one after another.
- It simply lays out components in a single row, starting a new row if its container is not sufficiently wide.



Constructors

FlowLayout()

FlowLayout(int how)

FlowLayout(int how, int horz, int vert)

**Values for how : *FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT*
*FlowLayout.LEADING ,FlowLayout.TRAILING***

FlowLayout Example:

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

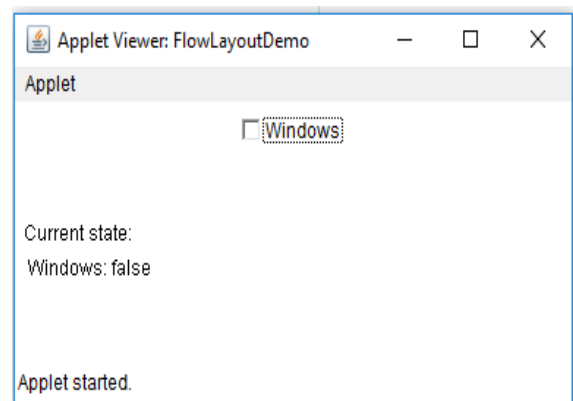
public class FlowLayoutDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.CENTER));
        windows = new Checkbox("Windows");
        add(windows);

        // register to receive item events
        windows.addItemListener(this);
    }

    // Repaint when status of a check box changes.
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
    }
}
```



BorderLayout

- The **BorderLayout** is used to arrange the components in five regions: north, south, east, west and center.



BorderLayout – constructors

1. **BorderLayout()** -> creates a default border layout
2. **BorderLayout(int horz, int vert)** -> allows to specify the horizontal and vertical space left between components in horz and vert, respectively.
3. BorderLayout defines the following constants that specify the regions:
 - BorderLayout.CENTER
 - BorderLayout.SOUTH
 - BorderLayout.EAST
 - BorderLayout.WEST
 - BorderLayout.NORTH

Adding the components to a BorderLayout

void add(Component compRef, Object region) -> region specifies where the component will be added.

// Demonstrate BorderLayout.

```
import java.awt.*;
import java.applet.*;
import java.util.*;

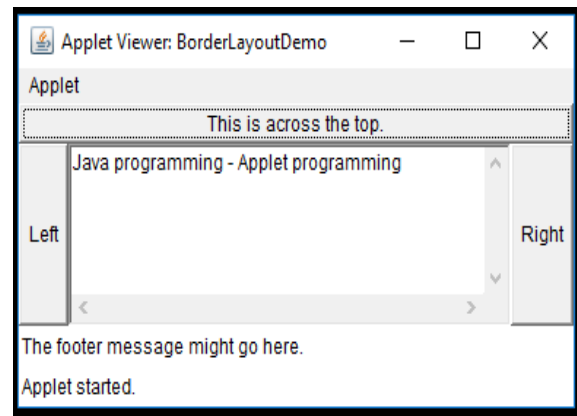
public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),BorderLayout.NORTH);
        add(new Label("The footer message might go here."),BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
    }
}
```

```

add(new Button("Left"), BorderLayout.WEST);

String msg = "Java programming - " +
            "Applet programming" +
            "\n";
add(new TextArea(msg), BorderLayout.CENTER);
}
}

```



GridLayout

- GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.
- GridLayout lays out components in a two-dimensional grid.
- When we instantiate a GridLayout, we define the number of rows and columns.

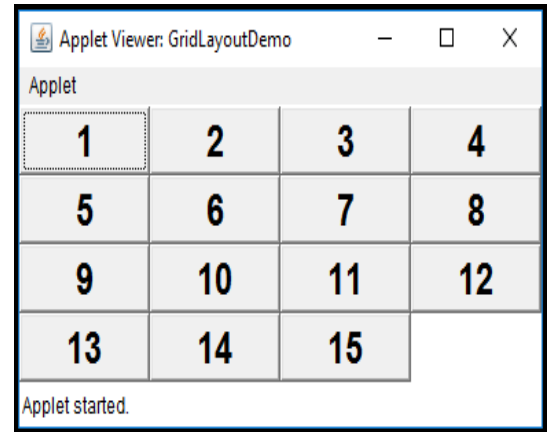


GridLayout Constructors

1. **GridLayout()** -> creates a single-column grid layout.
2. **GridLayout(int numRows, int numColumns)** -> creates a grid layout with the specified number of rows and columns.
3. **GridLayout(int numRows, int numColumns, int horz, int vert)** -> specify the horizontal and vertical space left between components in horz and vert, respectively.

Example: A sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
import java.awt.*;
import java.applet.*;
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    add(new Button("" + k));
            }
        }
    }
}
```



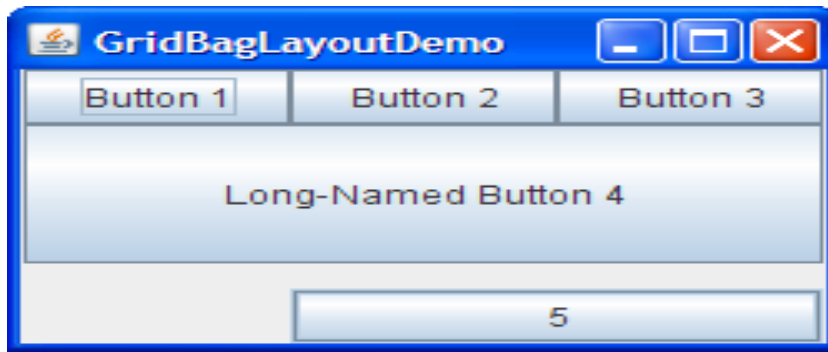
GridBagLayout

- GridBagLayout is a sophisticated, flexible layout manager.
- The rows in the grid can have different heights, and grid columns can have different widths.
- The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.
- The location and size of each component in a grid bag are determined by a set of constraints linked to it.
- The constraints are contained in an object of type **GridBagConstraints**

GridBagLayout – constructor

GridBagLayout defines only one constructor,

GridBagLayout()



GridBagLayout – constraints

GridBagConstraints.CENTER

GridBagConstraints.SOUTH

GridBagConstraints.EAST

GridBagConstraints.SOUTHEAST

GridBagConstraints.NORTH

GridBagConstraints.SOUTHWEST

GridBagConstraints.NORTHEAST

GridBagConstraints.WEST

GridBagConstraints.NORTHWEST

int anchor -> Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.

int gridheight -> Specifies the height of component in terms of cells. The default is 1.

int gridwidth -> Specifies the width of component in terms of cells. The default is 1

int gridx -> Specifies the X coordinate of the cell to which the component will be added

int gridy -> Specifies the Y coordinate of the cell to which the component will be added.

Example:

// Use GridBagLayout.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
public class GridBagDemo extends Applet implements ItemListener {
```

```
    String msg = "";
    Checkbox windows, android, solaris, mac;
```

```

public void init() {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Define check boxes.
    windows = new Checkbox("Windows ", null, true);
    android = new Checkbox("Android");

    gbc.anchor = GridBagConstraints.NORTHEAST;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(windows, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(android, gbc);

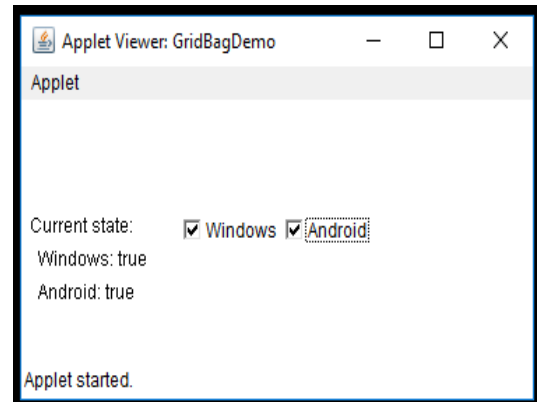
    // Add the components.
    add(windows);
    add(android);

    // Register to receive item events.
    windows.addItemListener(this);
    android.addItemListener(this);
}

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = " Android: " + android.getState();
    g.drawString(msg, 6, 120);
}
}

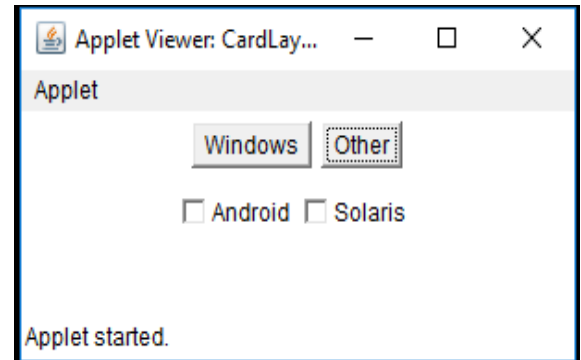
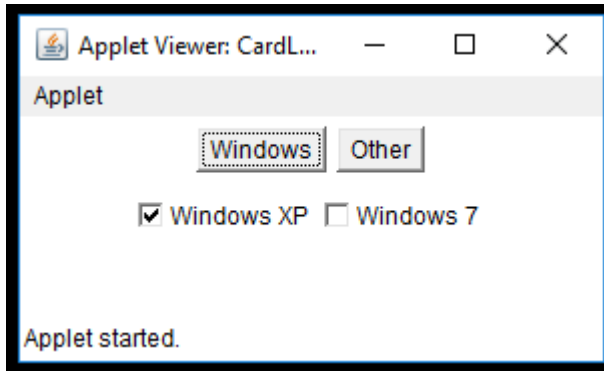
```



CardLayout

The CardLayout class is used to implement an area that contains different components at different times.

It is similar to deck of cards.

**Example:**

// Demonstrate CardLayout with the output shown above

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CardLayoutDemo extends Applet implements ActionListener{
    Checkbox windowsXP, windows7, windows8, android, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;
    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win);
        add(Other);
        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // set panel layout to card layout
        windowsXP = new Checkbox("Windows XP", null, true);
        windows7 = new Checkbox("Windows 7", null, false);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        // add Windows check boxes to a panel
        Panel winPan = new Panel();
        winPan.add(windowsXP);
```

```
winPan.add(windows7);
// Add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(android);
otherPan.add(solaris);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
}

public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {
        cardLO.show(osCards, "Windows");
    }
    else {
        cardLO.show(osCards, "Other");
    }
}
}
```

CardLayout – Methods

- 1) **void first(Container deck)** -> the first card in the deck will be shown
- 2) **void last(Container deck)** -> the last card in the deck will be shown
- 3) **void next(Container deck)** -> the next card in the deck will be shown
- 4) **void previous(Container deck)** -> the previous card in the deck will be shown
- 5) **void show(Container deck, String cardName)** -> displays the card whose name is passed in cardName

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**.

- To create a menu bar, first create an instance of **MenuBar**.

// create menu bar and add it to frame

```
MenuBar mbar = new MenuBar();
setMenuBar(mbar)
```

- Next, create instances of **Menu** that will define the selections displayed on the bar.

// create the menu items

```
Menu file = new Menu("File");
```

- Next, create individual menu items are of type **MenuItem**

```
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New"));
file.add(item2 = new MenuItem("Open"));
```

Example

Create a Sample Menu Program

```
import java.awt.*;
import java.awt.event.*;

public class SimpleMenuExample extends Frame implements ActionListener
{
    Menu file, edit;
    public SimpleMenuExample()
    {
        MenuBar mb = new MenuBar();           // begin with creating menu bar
        setMenuBar(mb);                       // add menu bar to frame

        file = new Menu("File");              // create menus
        edit = new Menu("Edit");

        mb.add(file);                         // add menus to menu bar
        mb.add(edit);

        file.addActionListener(this);         // link with ActionListener for event handling
        edit.addActionListener(this);
        file.add(new MenuItem("Open"));
```



```

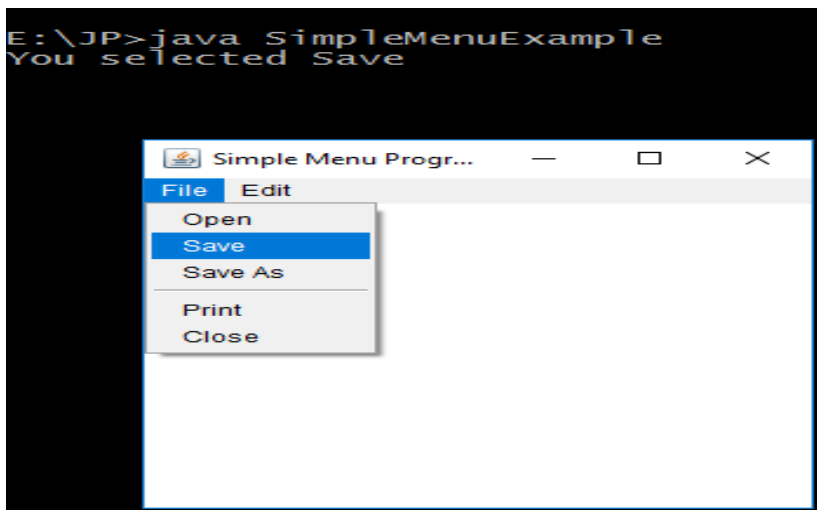
file.add(new MenuItem("Save"));
file.add(new MenuItem("Save As"));
file.addSeparator();
file.add(new MenuItem("Print"));
file.add(new MenuItem("Close"));

edit.add(new MenuItem("Cut"));
edit.add(new MenuItem("Copy"));
edit.add(new MenuItem("Paste"));
edit.addSeparator();
edit.add(new MenuItem("Special"));
edit.add(new MenuItem("Debug"));

setTitle("Simple Menu Program");           // frame creation methods
setSize(300, 300);
setVisible(true);

//closing the window
addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e) {
        dispose();
    }
});
}
public void actionPerformed(ActionEvent e)
{
    String str = e.getActionCommand();           // know the menu item selected by the user
    System.out.println("You selected " + str);
}
public static void main(String args[])
{
    new SimpleMenuExample();
}
}

```



Dialog Boxes

- We use a dialog box to hold a set of related controls.
- Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window.
- Dialog boxes don't have menu bars.
- Dialog boxes are two types : **modal** or **modeless**.
- **Modal dialog box** : In this, other parts of the program can not be accessible while the dialog box is active
- **Modeless dialog box**: In this, other parts of the program can be accessible while the dialog box is active.

Constructors:

Dialog(Frame parentWindow, boolean mode)

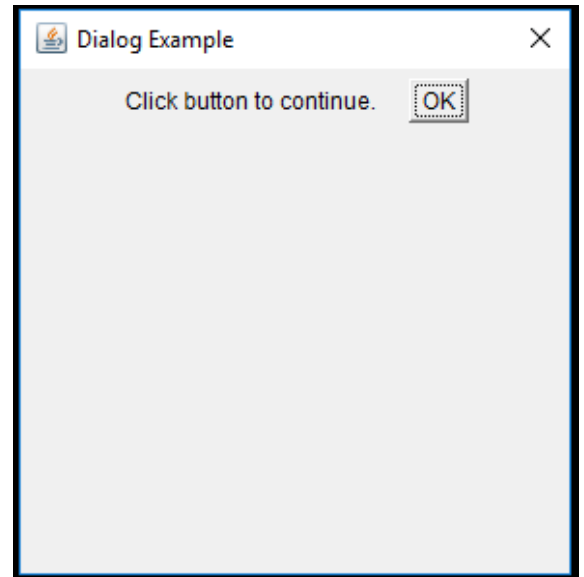
Dialog(Frame parentWindow, String title, boolean mode)

- If mode is true, the dialog box is modal
- If mode is false, the dialog box is modeless

Example:

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static Dialog d;
    DialogExample() {
        Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        Button b = new Button ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed((ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
        })
    }
}
```

```
});  
d.add( new Label ("Click button to continue."));  
d.add(b);  
d.setSize(300,300);  
d.setVisible(true);  
}  
public static void main(String args[])  
{  
    new DialogExample();  
}  
}
```



FileDialog

- Java provides a built-in dialog box that lets the user specify a file.
- To create a file dialog box, instantiate an object of type `FileDialog`. This causes a file dialog box to be displayed.

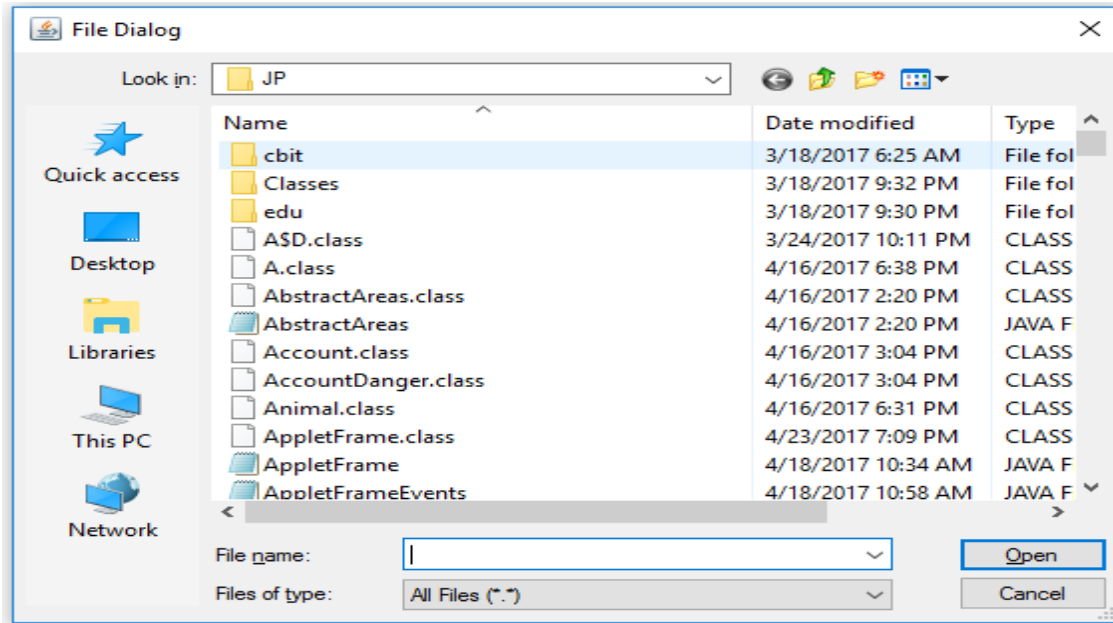
Constructors:

`FileDialog(Frame parent)`

`FileDialog(Frame parent, String boxName)`

`FileDialog(Frame parent, String boxName, int how)`

- If **how** is `FileDialog.LOAD`, then the box is selecting a file for reading.
- If **how** is `FileDialog.SAVE`, the box is selecting a file for writing.
- If **how** is **omitted**, the box is selecting a file for reading.



EVENTS – EVENTS SOURCES – EVENTS LISTENERS

SOURCE	Event Class	Listener	Listener Methods
MOUSE	MouseEvent	MouseListener, MouseMotionListener	mouseClicked(),mouseEntered(), mouseExited(),mousePressed() mouseReleased(), mouseDragged(),mouseMoved()
KEYBOARD	KeyEvent	KeyListener	keyPressed(),keyTyped()
BUTTON	ActionEvent	ActionListener	actionPerformed()
CHECKBOX	ItemEvent	ItemListener	itemStateChanged()
LIST	ItemEvent	ItemListener	itemStateChanged()
CHOICE	ItemEvent	ItemListener	itemStateChanged()
MENUITEM	ActionEvent	ActionListener	actionPerformed()
TEXTFIELD or TEXTAREA	ActionEvent	ActionListener	actionPerformed()
SCROLLBAR	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()

UNIT – V - END