

Java Programming – 18BCS43C

By Dr. S. Chitra,

Associate Professor,

Post Graduate & Research Department of Computer Science,

Government Arts College(Autonomous), Coimbatore - 641018

UNIT-II

Classes, Objects and Methods: Classes and Objects - Constructors- Method Overloading- Static Members-Inheritance- Overriding Methods- Final Variables, Final Methods and Final Classes - Finalize Method- Abstract Methods and Abstract Classes –Visibility Control - Arrays - Strings.

Objects in Java

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Some of the real-world objects around us are cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.
- A class is a blueprint from which individual objects are created.

Definition: A class is a collection of objects of similar type. Once a class is defined, any number of objects can be produced which belong to that class.

Class Declaration

```
class classname
```

```
{
```

```
... ClassBody
```

```
...}
```

Objects are instances of the Class. Classes and Objects are very much related to each other. Without objects you can't use a class. Without class objects cannot be created.

A general class declaration:

```
class name1
{
//public variable declaration
void methodname()
{
//body of method...
//Anything
}
}
```

Now following example shows the use of method.

```
class Demo
{
private int x,y,z;
public void input()
{
x=10;
y=15;
}
public void sum()
{
z=x+y;
}
public void print_data()
{
System.out.println("Answer is =" +z);
}
public static void main(String args[])
{
Demo object=new Demo();
object.input();
object.sum();
object.print_data();
}
}
```

In program,

```
Demo object=new Demo();
object.input();
object.sum();
```

```
object.print_data();
```

In the first line we created an object. The three methods are called by using the dot operator. When we call a method the code inside its block is executed. The dot operator is used to call methods or access them.

Creating “main” in a separate class

We can create the main method in a separate class, but during compilation you need to make sure that you compile the class with the “main” method.

```
class Demo
{
private int x,y,z;
public void input() {
x=10;
y=15;
}
public void sum()
{
z=x+y;
}
public void print_data()
{
System.out.println(“Answer is =” +z);
}
}
class SumDemo
{
public static void main(String args[])
{
Demo object=new Demo();
object.input();
object.sum();
object.print_data();
}
}
```

Use of dot operator

We can access the variables by using dot operator. Following program shows the use of dot operator.

```
class DotDemo
{
int x,y,z;
public void sum(){
z=x+y;
}
```

```

public void show(){
System.out.println("The Answer is "+z);
}
}
class Demo1
{
public static void main(String args[]){
DotDemo object=new DotDemo();
DotDemo object2=new DotDemo();
object.x=10;
object.y=15;
object2.x=5;
object2.y=10;
object.sum();
object.show();
object2.sum();
object2.show();
}}

```

output :

C:\cc>javac Demo1.java

C:\cc>java Demo1

The Answer is 25

The Answer is 15

Instance Variable

All non-static variables declared inside a class are also known as instance variable. This is because of the fact that each instance or object has its own copy of values for the variables. Hence other use of the “*dot*” operator is to initialize the value of variable for that instance.

Methods with parameters

Following program shows the method with passing parameter.

```

class prg
{
int n,n2,sum;
public void take(int x,int y)
{
n=x;
n2=y;
}
public void sum()
{
sum=n+n2;
}
public void print()
{
System.out.println("The Sum is "+sum);
}
}

```

```

}
}
class prg1
{
public static void main(String args[])
{
prg obj=new prg();
obj.take(10,15);
obj.sum();
obj.print();
}
}

```

Methods with a Return Type

When method return some value that is the type of that method.

For Example: some methods are with parameter but that method did not return any value that means type of method is void. And if method return integer value then the type of method is an integer.

Following program shows the method with their return type.

```

class Demo1
{
int n,n2;
public void take( int x,int y)
{
n=x;
n=y;
}
public int process()
{
return (n+n2);
}
}
class prg
{
public static void main(String args[])
{
int sum;
Demo1 obj=new Demo1();
obj.take(15,25);
sum=obj.process();
System.out.println("The sum is "+sum);
}
}

```

Output:

The sum is 25

Following is a sample of a class.

Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

A class can contain any of the following variable types.

- **Local variables** – Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** – Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** – Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Creating an Object

A class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class –

- **Declaration** – A variable declaration with a variable name with an object type.
- **Instantiation** – The 'new' keyword is used to create the object.
- **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object –

Example

```

public class Puppy {
    public Puppy(String name) {
        // This constructor has one parameter, name.
        System.out.println("Passed Name is :" + name );
    }

    public static void main(String []args) {
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}

```

If we compile and run the above program, then it will produce the following result –

Output

Passed Name is :tommy

Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path –

```

/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();

```

Example

This example explains how to access instance variables and methods of a class.

```

public class Puppy {
    int puppyAge;

    public Puppy(String name) {
        // This constructor has one parameter, name.
        System.out.println("Name chosen is :" + name );
    }

    public void setAge( int age ) {
        puppyAge = age;
    }

    public int getAge( ) {
        System.out.println("Puppy's age is :" + puppyAge );
        return puppyAge;
    }

    public static void main(String []args) {
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );
    }
}

```

```

/* Call class method to set puppy's age */
myPuppy.setAge( 2 );

/* Call another class method to get puppy's age */
myPuppy.getAge( );

/* You can access instance variable as follows as well */
System.out.println("Variable Value : " + myPuppy.puppyAge );
}
}

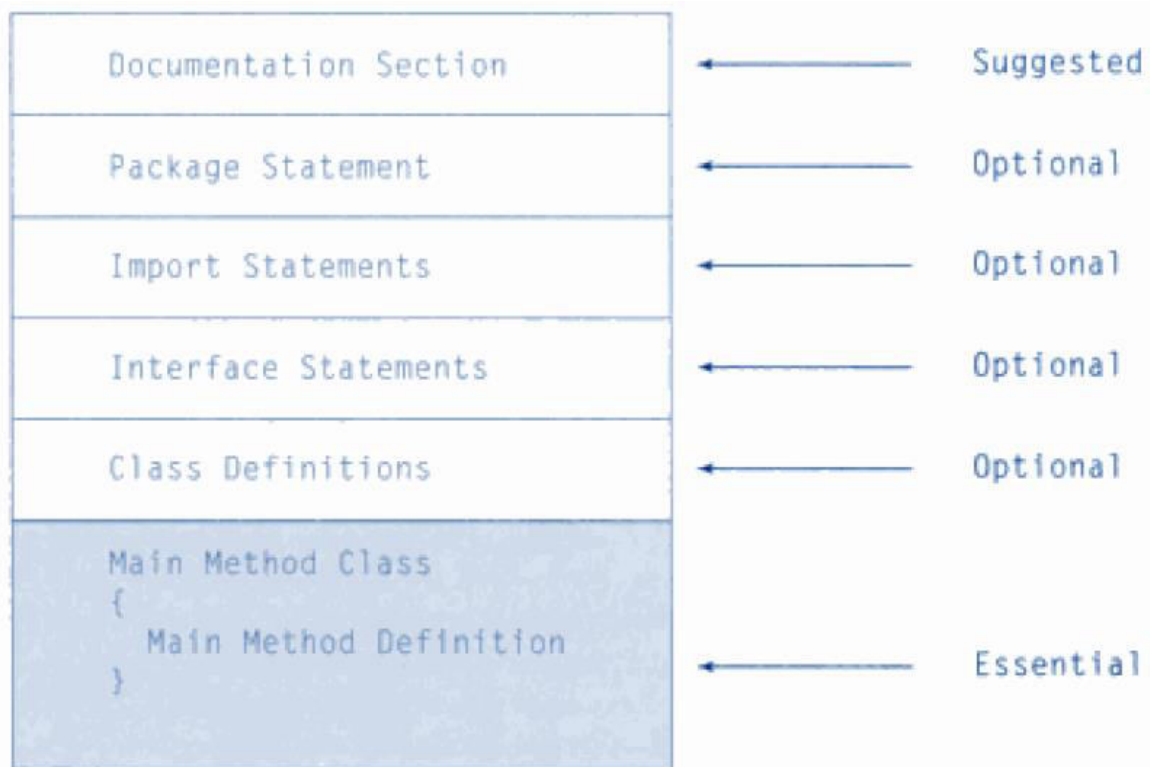
```

If we compile and run the above program, then it will produce the following result –

Output

Name chosen is :tommy
Puppy's age is :2
Variable Value :2

Java Program structure



General structure of a Java program

Source File Declaration Rules

Source file declaration rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.

- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java_installation/java/io –

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

A Simple Case Study

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

Example

```
import java.io.*;
public class Employee {

    String name;
    int age;
    String designation;
    double salary;
```

```

// This is the constructor of the class Employee
public Employee(String name) {
    this.name = name;
}

// Assign the age of the Employee to the variable age.
public void empAge(int empAge) {
    age = empAge;
}

/* Assign the designation to the variable designation.*/
public void empDesignation(String empDesig) {
    designation = empDesig;
}

/* Assign the salary to the variable salary.*/
public void empSalary(double empSalary) {
    salary = empSalary;
}

/* Print the Employee details */
public void printEmployee() {
    System.out.println("Name:" + name );
    System.out.println("Age:" + age );
    System.out.println("Designation:" + designation );
    System.out.println("Salary:" + salary);
}
}

```

In Java, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```

import java.io.*;
public class EmployeeTest {

    public static void main(String args[]) {
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();

        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}

```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows –

Output

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Java Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println()` method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Considering the following example to explain the syntax of a method –

Syntax

```
public static int methodName(int a, int b) {
    // body
}
```

Here,

- **public static** – modifier
- **int** – return type
- **methodName** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax –

Syntax

```
modifier returnType nameOfMethod (Parameter List) {
    // method body
}
```

The syntax shown above includes –

- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.

- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **min()**. This method takes two parameters num1 and num2 and returns the maximum between the two –

```
/** the snippet returns the minimum between two numbers */

public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
```

Method Calling

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

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

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example –

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

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

```
int result = sum(6, 9);
```

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

Example

```
public class ExampleMinNumber {

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

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
```

```
int min;
if (n1 > n2)
    min = n2;
else
    min = n1;

return min;
}
```

This will produce the following result –

Output

Minimum value = 6

The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

Example

```
public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
            System.out.println("Rank:A1");
        } else if (points >= 122.4) {
            System.out.println("Rank:A2");
        } else {
            System.out.println("Rank:A3");
        }
    }
}
```

This will produce the following result –

Output

Rank:A1

Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample {

    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b);

        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping values will be same here**");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }

    public static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

        // Swap n1 with n2
        int c = a;
        a = b;
        b = c;
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);
    }
}
```

This will produce the following result –

Output

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

Now, Before and After swapping values will be same here:
After swapping, a = 30 and b is 45

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same –

Example

```
public class ExampleOverloading {

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
```

```

double c = 7.3;
double d = 9.4;
int result1 = minFunction(a, b);

// same function name with different parameters
double result2 = minFunction(c, d);
System.out.println("Minimum Value = " + result1);
System.out.println("Minimum Value = " + result2);
}

// for integer
public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}

// for double
public static double minFunction(double n1, double n2) {
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
}

```

This will produce the following result –

Output

```

Minimum Value = 6
Minimum Value = 7.3

```

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the `String` array passed to `main()`.

Example

The following program displays all of the command-line arguments that it is called with –

```

public class CommandLine {

    public static void main(String args[]) {

```

```

for(int i = 0; i<args.length; i++) {
    System.out.println("args[" + i + "]: " + args[i]);
}
}
}

```

Try executing this program as shown here –

\$java CommandLine this is a command line 200 -100

This will produce the following result –

Output

```

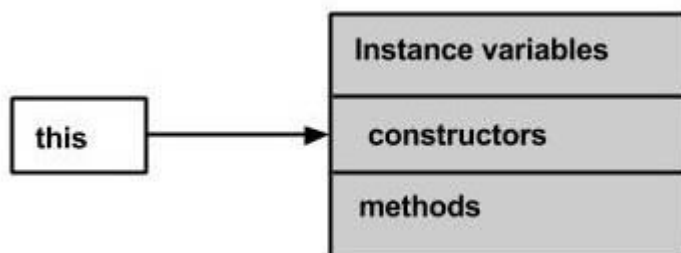
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100

```

The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

Note – The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to –

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```

class Student {
    int age;
    Student(int age) {
        this.age = age;
    }
}

```

- Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```

class Student {
    int age
    Student() {
        this(20);
    }

    Student(int age) {

```



```
    this.age = age;
}
}
```

Example

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name, **This_Example.java**.

```
public class This_Example {
    // Instance variable num
    int num = 10;

    This_Example() {
        System.out.println("This is an example program on keyword this");
    }

    This_Example(int num) {
        // Invoking the default constructor
        this();

        // Assigning the local variable num to the instance variable num
        this.num = num;
    }

    public void greet() {
        System.out.println("Hi Welcome to Tutorialspoint");
    }

    public void print() {
        // Local variable num
        int num = 20;

        // Printing the local variable
        System.out.println("value of local variable num is : "+num);

        // Printing the instance variable
        System.out.println("value of instance variable num is : "+this.num);

        // Invoking the greet method of a class
        this.greet();
    }

    public static void main(String[] args) {
        // Instantiating the class
        This_Example obj1 = new This_Example();

        // Invoking the print method
        obj1.print();

        // Passing a new value to the num variable through parametrized constructor
        This_Example obj2 = new This_Example(30);

        // Invoking the print method again
        obj2.print();
    }
}
```

This will produce the following result –

Output

This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Hi Welcome to Tutorialspoint
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Hi Welcome to Tutorialspoint

Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows –

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```
public class VarargsDemo {  
  
    public static void main(String args[]) {  
        // Call method with variable args  
        printMax(34, 3, 3, 2, 56.5);  
        printMax(new double[]{1, 2, 3});  
    }  
  
    public static void printMax( double... numbers) {  
        if (numbers.length == 0) {  
            System.out.println("No argument passed");  
            return;  
        }  
  
        double result = numbers[0];  
  
        for (int i = 1; i < numbers.length; i++)  
            if (numbers[i] > result)  
                result = numbers[i];  
        System.out.println("The max value is " + result);  
    }  
}
```

This will produce the following result –

Output

The max value is 56.5
The max value is 3.0

The finalize() Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize()**, and it can be used to ensure that an object terminates cleanly.

For example, you might use `finalize()` to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the `finalize()` method. The Java runtime 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.

This means that you cannot know when or even if `finalize()` will be executed. For example, if your program ends before garbage collection occurs, `finalize()` will not execute.

Constructor

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor –

Example

```
public class Puppy {  
    public Puppy() {  
    }  
  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
    }  
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note – We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Syntax

Following is the syntax of a constructor –

```
class ClassName {  
    ClassName() {  
    }  
}
```

Java allows two types of constructors namely –

- No argument Constructors
- Parameterized Constructors

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

Example

```
Public class MyClass {  
    Int num;  
    MyClass() {  
        num = 100;  
    }  
}
```

You would call constructor to initialize objects as follows

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

This would produce the following result

100 100

Parameterized Constructors

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor –

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i) {
        x = i;
    }
}
```

You would call constructor to initialize objects as follows –

```
public class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Inheritance - extends Keyword

A mechanism of creating new classes by acquiring the properties of the existing classes is called Inheritance.

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
    ....
    ....
}
class Sub extends Super {
    ....
    ....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

Example

```
class Calculation {
    int z;

    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }

    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}
```

Compile and execute the above code as shown below.

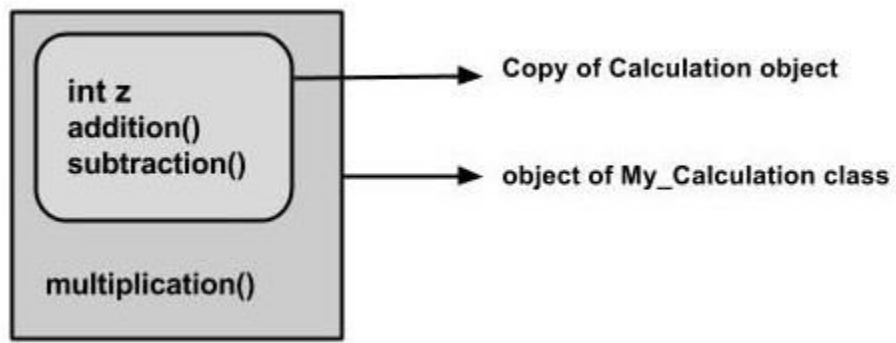
```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result –

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass **My_Calculation**.

```
Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);
```

Note – A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
super.method();
```

Sample Code

Demonstration of the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named `display()` with different implementations, and a variable named `num` with different values. We are invoking `display()` method of both classes and printing the value of the variable `num` of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name `Sub_class.java`.

Example

```

class Super_class {
    int num = 20;

    // display method of superclass
    public void display() {
        System.out.println("This is the display method of superclass");
    }
}

public class Sub_class extends Super_class {
    int num = 10;

    // display method of sub class
    public void display() {
        System.out.println("This is the display method of subclass");
    }

    public void my_method() {
        // Instantiating subclass
        Sub_class sub = new Sub_class();

        // Invoking the display() method of sub class
        sub.display();

        // Invoking the display() method of superclass
        super.display();

        // printing the value of variable num of subclass
        System.out.println("value of the variable named num in sub class:"+ sub.num);

        // printing the value of variable num of superclass
        System.out.println("value of the variable named num in super class:"+ super.num);
    }

    public static void main(String args[]) {
        Sub_class obj = new Sub_class();
        obj.my_method();
    }
}

```

Compile and execute the above code using the following syntax.

```

javac Super_Demo
java Super

```

On executing the program, you will get the following result –

Output

```

This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

```
class Superclass {
    int age;

    Superclass(int age) {
        this.age = age;
    }

    public void getAge() {
        System.out.println("The value of the variable named age in super class is: " +age);
    }
}

public class Subclass extends Superclass {
    Subclass(int age) {
        super(age);
    }

    public static void main(String args[]) {
        Subclass s = new Subclass(24);
        s.getAge();
    }
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
java Subclass
```

On executing the program, you will get the following result –

Output

The value of the variable named age in super class is: 24

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal {
}
```

```
public class Mammal extends Animal {  
}  
  
public class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say –

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal {  
}  
  
class Mammal extends Animal {  
}  
  
class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

This will produce the following result –

Output

true

true
true

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {  
}  
  
public class Mammal implements Animal {  
}  
  
public class Dog extends Mammal {  
}
```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```
interface Animal{  
class Mammal implements Animal{  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

This will produce the following result –

Output

true
true
true

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example –

Example

```

public class Vehicle{}
public class Speed{}

public class Van extends Vehicle {
    private Speed sp;
}

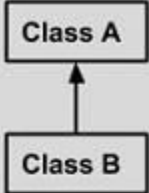
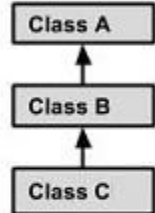
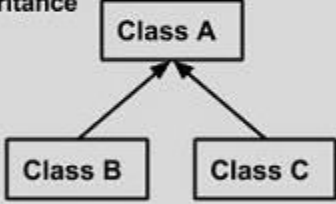
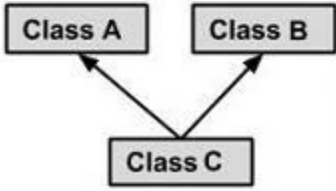
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support multiple Inheritance </pre>

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal –

Example

```

public class extends Animal, Mammal{}

```

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal(); // Animal reference and object
        Animal b = new Dog();    // Animal reference but Dog object

        a.move(); // runs the method in Animal class
        b.move(); // runs the method in Dog class
    }
}
```

This will produce the following result –

Output

Animals can move

Dogs can walk and run

In the above example, you can see that even though **b** is a type of **Animal** it runs the **move** method in the **Dog** class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since **Animal** class has the method **move**. Then, at the runtime, it runs the method specific for that object.

Consider the following example –

Example

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}
```

```

    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
    public void bark() {
        System.out.println("Dogs can bark");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal(); // Animal reference and object
        Animal b = new Dog(); // Animal reference but Dog object

        a.move(); // runs the method in Animal class
        b.move(); // runs the method in Dog class
        b.bark();
    }
}

```

This will produce the following result –

Output

```

TestDog.java:26: error: cannot find symbol
    b.bark();
    ^
symbol:   method bark()
location: variable b of type Animal
1 error

```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

Example

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        super.move(); // invokes the super class method
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal b = new Dog(); // Animal reference but Dog object
        b.move(); // runs the method in Dog class
    }
}
```

This will produce the following result –

Output

```
Animals can move
Dogs can walk and run
```

Method Overloading

Method overloading means method name will be same but each method should be different parameter list.

```
class prg1
{
    int x=5,y=5,z=0;
    public void sum()
    {
        z=x+y;
```

```

System.out.println("Sum is "+z);
}

public void sum(int a,int b)
{
x=a;
y=b;
z=x+y;
System.out.println("Sum is "+z);
}
public int sum(int a)
{
x=a;
z=x+y;
return z;
}
}
class Demo
{
public static void main(String args[])
{
prg1 obj=new prg1();
obj.sum();
obj.sum(10,12);
System.out.println(+obj.sum(15));
}
}

```

Output:

```

sum is 10
sum is 22
27

```

Passing Objects as Parameters

Objects can even be passed as parameters.

```

class para123
{
int n,n2,sum,mul;
public void take(int x,int y)
{
n=x;
n2=y;
}
public void sum()
{
sum=n+n2;
System.out.println("The Sum is "+sum);
}
}

```



```

    }
    public void take2(para123 obj)
    {
        n=obj.n;
        n2=obj.n2;
    }
    public void multi()
    {
        mul=n*n2;
        System.out.println("Product is"+mul);
    }
}
class DemoPara
{
    public static void main(String args[])
    {
        para123 ob=new para123();
        ob.take(3,7);
        ob.sum();
        ob.take2(ob);
        ob.multi();
    }
}

```

Output:

```

C:\cc>javac DemoPara.java
C:\cc>java DemoPara
The Sum is10
Product is21

```

We have defined a method “*take2*” that declares an object named obj as parameter. We have passed ob to our method. The method “*take2*” automatically gets 3,7 as values for n and n2.

Passing Values to methods and Constructor:

These are two different ways of supplying values to methods. Classified under these two titles –

- 1.Pass by Value
- 2.Pass by Address or Reference

1. **Pass by Value**-When we pass a data type like int, float or any other datatype to a method or some constant values like(15,10). They are all passed by value. A copy of variable’s value is passed to the receiving method and hence any changes made to the values do not affect the actual variables.

```

class Demopbv
{

```

```

int n,n2;
public void get(int x,int y)
{
x=x*x; //Changing the values of passed arguments
y=y*y; //Changing the values of passed arguments
}
}
class Demo345
{
public static void main(String args[])
{
int a,b;
a=1;
b=2;
System.out.println("Initial Values of a & b "+a+" "+b);
Demopbv obj=new Demopbv();
obj.get(a,b);
System.out.println("Final Values "+a+" "+b);
}
}

```

Output:

C:\cc>javac Demo345.java

C:\cc>java Demo345

Initial Values of a & b 1 2

Final Values 1 2

2. Pass by Reference

Objects are always passed by reference. When we pass a value by reference, the reference or the memory address of the variables is passed. Thus any changes made to the argument causes a change in the values which we pass.

Demonstrating Pass by Reference---

```

class pass_by_ref
{
int n,n2;
public void get(int a,int b)
{
n=a;
n2=b;
}
public void doubleit(pass_by_ref temp)
{
temp.n=temp.n*2;
temp.n2=temp.n2*2;
}
}
class apply7
{

```

```

public static void main(String args[])
{
    int x=5,y=10;
    pass_by_ref obj=new pass_by_ref();
    obj.get(x,y); //Pass by Value
    System.out.println("Initial Values are-- ");
    System.out.println(+obj.n);
    System.out.println(+obj.n2);
    obj.doubleit(obj); //Pass by Reference
    System.out.println("Final Values are");
    System.out.println(+obj.n);
    System.out.println(+obj.n2);
}
}

```

Abstract Classes

Definition: An abstract class is a class that is declared as abstract. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclass.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void studtest(int rollno, double testfees);
```

If a class includes abstract methods, the class itself must be declared abstract, as in:

```

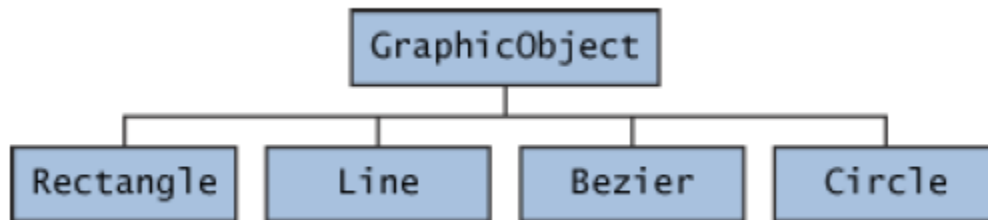
public abstract class GraphicObject
{
    // declare fields
    // declare non-abstract methods
    abstract void draw();
}

```

When an abstract class is subclass, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

For example: In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for

example, GraphicObject, as shown in the following figure.



How to implement above diagram concept with source code:

```
abstract class GraphicObject
{
int x, y;
...
void moveTo(int newX, int newY)
{
...
}
abstract void draw();
abstract void resize();
}
```

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
void draw() {
...
}
void resize() {
...
}
}

class Rectangle extends GraphicObject {
void draw() {
...
}
void resize() {
...
}
}
```

Abstract classes are those which can be used for creation of objects. However their methods and constructors can be used by the child or extended class. The need for abstract classes is that you can generalize the super class from which child classes can share its methods. The subclass of an abstract class which can create an object is called as "concrete class".

For example:

Abstract class A

```

{
abstract void method1();
void method2()
{
System.out.println("this is real method");
}}
class B extends A
{
void method1()
{
System.out.println("B is execution of method1");
}}
class demo
{
public static void main(String arg[])
{
B b=new B();
b.method1();
b.method2();
}}

```

Extending the class:

Inheritance allows to subclass or child class to access all methods and variables of parent class.

Syntax:

```

class subclassname extends superclassname
{
Variables;
Methods;
.....
}

```

For example: calculate area and volume by using Inheritance.

```

class data
{
int l;
int b;
data(int c, int d)
{
l=c;
b=d;
}
int area( )
{
return(l*b);
}
}

```

```

class data2 extends data
{
int h;
data2(int c,int d, int a)
{
super(c,d);
h=a;
}
int volume()
{
return(l*b*h);
}
}

```

```

class dataDemo
{
public static void main(String args[])
{
data2 d1=new data2(10,20,30);
int area1=d1.area(); //superclass method
int volume1=d1.volume( );// subclass method
System.out.println("Area="+area1);
System.out.println("Volume="+volume1);
}
}

```

Output:

C:\cc>javac dataDemo.java

C:\cc>java dataDemo

Area=200

Volume=6000

"Is A" - is a subclass of a superclass (ex: extends)

"Has A" - has a reference to (ex: variable, ref to object).

*****Access Control –**

Away to limit the access others have to your code.

**** Same package** - can access each others' variables and methods, except for private members.

**** Outside package** - can access public classes. Next, can access members that are public. Also, can access protected members if the class is a subclass of that class.

Same package - use package keyword in first line of source file, or no package keyword and in same directory.

*****Keywords -**

1. public - outside of package access.
2. [no keyword] - same package access only.
3. protected - same package access. Access if class is a subclass of, even if in another package.
4. private - same class access only.

Static Members

A static member is a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

```
static int count;  
static int max(int x, int y):
```

The members that are declared **static** as shown above are called *static members*. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as *class variables* and *class methods* in order to distinguish them from their counterparts, instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. One of the most common examples is to have a variable that could keep a count of how many objects of a class have been created.

Like static variables, static methods can be called without using the objects. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java class libraries contain a large number of class methods. For example, the **Math** class of Java library defines many static methods to perform math operations that can be used in any program. We have used earlier statements of the types.

```
float x = Math.sqrt(25.0);
```

The method **sqrt** is a class method (or static method) defined in **Math** class.

Defining and using static members

```
class Mathoperation  
{  
    static float mul(float x, float y)  
    {  
        return x*y;  
    }  
    static float divide (float x, float y)  
    {  
        return x/y;  
    }  
}  
class MathApplication  
{  
    public void static main(String args[ ])   
    {  
        float a = MathOperation.mul(4.0,5.0);  
        float b = MathOperation.divide(a,2.0);  
        System.out.println("b = "+ b);  
    }  
}
```

Output of Program

```
b = 10.0
```

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have several restrictions:

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

Final Classes

Sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclassed is called a *final class*. This is achieved in Java using the keyword **final** as follows:

```
final class Aclass {.....}  
final class Bclass extends Someclass {.....}
```

Any attempt to inherit these classes will cause an error and the compiler will not allow it.

Declaring a class **final** prevents any unwanted extensions to the class. It also allows the compiler to perform some optimisations when a method of a final class is invoked.

Abstract Methods and Classes

We have seen that by making a method *final* we ensure that the method is not redefined in a subclass. That is, the method can never be subclassed. Java allows us to do something that is exactly opposite to this. That is, we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition. Example:

```
abstract class Shape  
{  
    .....  
    .....  
    abstract void draw( );  
    .....  
    .....  
}
```

When a class contains one or more abstract methods, it should also be declared **abstract** as shown in the example above.

While using abstract classes, we must satisfy the following conditions:

- We cannot use abstract classes to instantiate objects directly. For example,
 Shape s = new Shape()
 is illegal because **shape** is an abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

Visibility Control

The visibility modifiers are also known as *access modifiers*. Java provides three types of visibility modifiers: **public**, **private** and **protected**. They provide different levels of protection as described below.

public Access

```
public int number;  
public void sum( ) {.....}
```

A variable or method declared as **public** has the widest possible visibility and accessible everywhere. In fact, this is what we would like to prevent in many programs. This takes us to the next levels of protection.

friendly Access

The difference between the “public” access and the “friendly” access is that the **public** modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages. (A package is a group of related classes stored separately.). A package in Java is similar to a source file in C.

protected Access

The visibility level of a “protected” field lies in between the public access and friendly access. That is, the **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that non-subclasses in other packages cannot access the “protected” members.

private Access

private fields enjoy the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as **private** behaves like a method declared as **final**. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

private protected Access

A field can be declared with two keywords **private** and **protected** together like:

```
private protected int codeNumber;
```

This gives a visibility level in between the “protected” access and “private” access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package. Table : summarises the visibility provided by various access modifiers.

Visibility of Field in a Class

Access modifier → Access location ↓	public	protected	friendly (default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Rules of Thumb

The details discussed so far about field visibility may be quite confusing and seem complicated. Given below are some simple rules for applying appropriate access modifiers.

1. Use **public** if the field is to be visible everywhere.
2. Use **protected** if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use "default" if the field is to be visible everywhere in the current package only.
4. Use **private protected** if the field is to be visible only in subclasses, regardless of packages.
5. Use **private** if the field is **not** to be visible anywhere except in its own class.

Creating an Array

Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

1. Declaring the array
2. Creating memory locations
3. Putting values into the memory locations.

Declaration of Arrays

Arrays in Java may be declared in two forms :

Form 1

```
type arrayname[ ];
```

Form 2

```
type [ ] arrayname;
```

Examples:

```
int      number[ ];
float    average[ ];
int[ ]   counter;
float[ ] marks;
```

Remember, we do not enter the size of the arrays in the declaration.

Creation of Arrays

After declaring an array, we need to create it in the memory. Java allows us to create arrays using **new** operator only, as shown below:

```
arrayname = new type[size];
```

Examples:

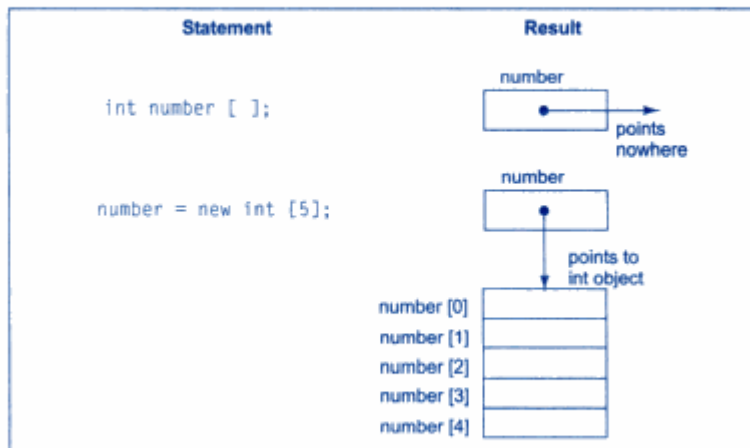
```
number = new int[5];
average = new float[10];
```

These lines create necessary memory locations for the arrays **number** and **average** and designate them as **int** and **float** respectively. Now, the variable **number** refers to an array of 5 integers and **average** refers to an array of 10 floating point values.

It is also possible to combine the two steps—declaration and creation—into one as shown below:

```
int number[ ] = new int[5];
```

Figure 9.1 illustrates creation of an array in memory.



Creation of an array in memory

Initialization of Arrays

The final step is to put values into the array created. This process is known as *initialization*. This is done using the array subscripts as shown below.

```
arrayname[subscript] = value ;
```

Example:

```
number[0] = 35;  
number[1] = 40;  
.....  
.....  
number[4] = 19;
```

Note that Java creates arrays starting with the subscript of 0 and ends with a value one less than the *size* specified.

Unlike C, Java protects arrays from overruns and underruns. Trying to access an array bound its boundaries will generate an error message.

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

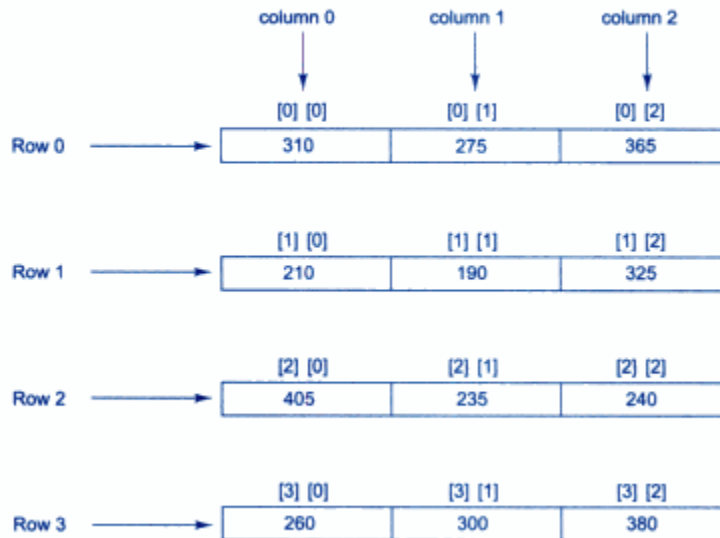
```
type arrayname[ ] = {list of values};
```

The array initializer is a list of values separated by commas and surrounded by curly braces. Note that no size is given. The compiler allocates enough space for all the elements specified in the list.

Two-dimensional Arrays

v[4][3]

Two dimensional arrays are stored in memory as shown in Fig. As with the single dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.



Representation of a two-dimensional array in memory

```
int table[ ][ ] = {{0, 0, 0}, {1, 1, 1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

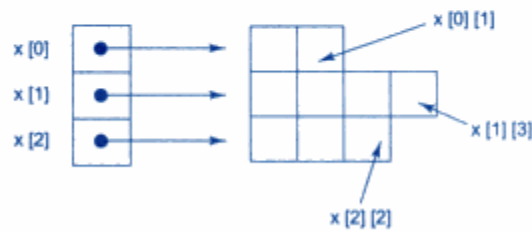
```
int table[ ][ ] = {  
    {0, 0, 0},  
    {1, 1, 1}  
};
```

Variable Size Arrays

Java treats multidimensional array as “arrays of arrays”. It is possible to declare a two-dimensional array as follows:

```
int x[ ][ ] = new int[3][ ];  
x[0] = new int[2];  
x[1] = new int[4];  
x[2] = new int[3];
```

These statements create a two-dimensional array as having different lengths for each row as shown in Fig.



Variable size arrays

Strings

String manipulation is the most common part of many Java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is by using a character array.

Example:

```
char charArray[ ] = new char[4];
charArray[0]      = 'J';
charArray[1]      = 'a';
charArray[2]      = 'v';
charArray[3]      = 'a';
```

Although character arrays have the advantage of being able to query their length, they themselves are not good enough to support the range of operations we may like to perform on strings. For example, copying one character array into another might require a lot of book keeping effort. Fortunately, Java is equipped to handle these situations more efficiently.

In Java, strings are class objects and implemented using two classes, namely, **String** and **StringBuffer**. A Java string is an instantiated object of the **String** class. Java strings, as compared to C strings, are more reliable and predictable. This is basically due to C's lack of bounds-checking. A Java string is not a character array and is not NULL terminated. Strings may be declared and created as follows:

```
String stringName;
StringName = new String ("string");
```

Example:

```
String firstName;
firstName = new String("Anil");
```

These two statements may be combined as follows:

```
String firstName = new String ("Anil");
```

Like arrays, it is possible to get the length of string using the **length** method of the **String** class.

```
int m = firstName.length( );
```

Note the use of parentheses here. Java strings can be concatenated using the + operator. Examples:

```
String fullName = name1 + name2;
String city1    = "New" + "Delhi";
```

where **name1** and **name2** are Java strings containing string constants. Another example is:

```
System.out.println(firstName + "Kumar");
```

String Methods

The **String** class defines a number of methods that allow us to accomplish a variety of string manipulation tasks. Table lists some of the most commonly used string methods, and their tasks.

Some Most Commonly Used String Methods	
Method Call	Task performed
s2 = s1.toLowerCase();	Converts the string s1 to all lowercase
s2 = s1.toUpperCase();	Converts the string s1 to all Uppercase
s2 = s1.replace('x', 'y');	Replace all appearances of x with y
s2 = s1.trim();	Remove white spaces at the beginning and end of the string s1
s1.equals(s2)	Returns 'true' if s1 is equal to s2
s1.equalsIgnoreCase(s2)	Returns 'true' if s1 = s2, ignoring the case of characters
s1.length()	Gives the length of s1
s1.charAt(n)	Gives nth character of s1
s1.compareTo(s2)	Returns negative if s1 < s2, positive if s1 > s2, and zero if s1 is equal s2
s1.concat(s2)	Concatenates s1 and s2
s1.substring(n)	Gives substring starting from n th character
s1.substring(n, m)	Gives substring starting from n th character up to m th (not including m th)
String.valueOf(p)	Creates a string object of the parameter p (simple type or object)
p.toString()	Creates a string representation of the object p
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
s1.indexOf('x', n)	Gives the position of 'x' that occurs after nth position in the string s1
String.valueOf(Variable)	Converts the parameter value to string representation

StringBuffer Class

StringBuffer is a peer class of **String**. While **String** creates strings of fixed_length, **StringBuffer** creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end. Table 11.1 lists some of the methods that are frequently used in string manipulations.

Commonly Used StringBuffer Methods

<i>Method</i>	<i>Task</i>
<code>s1.setCharAt(n, 'x')</code>	Modifies the nth character to x
<code>s1.append(s2)</code>	Appends the string s2 to s1 at the end
<code>s1.insert(n, s2)</code>	Inserts the string s2 at the position n of the string s1
<code>s1.setLength(n)</code>	Sets the length of the string s1 to n. If $n \leq s1.length()$ s1 is truncated. If $n > s1.length()$ zeros are added to s1

LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.
2. An Introduction to Object Oriented Programming with JAVA, C THOMAS WU
3. “Java Programming” Fourth Edition, E. Balagurusamy, Tata McGraw Hill.

BIBLIOGRAPHY

http://www.michael-homas.com/tech/java/javacert/JCP_Access.htm

http://en.wikipedia.org/wiki/Class_%28computer_science%29#Sealed_classes

<http://www.javabeginner.com/learn-java/java-abstract-class-andinterface>