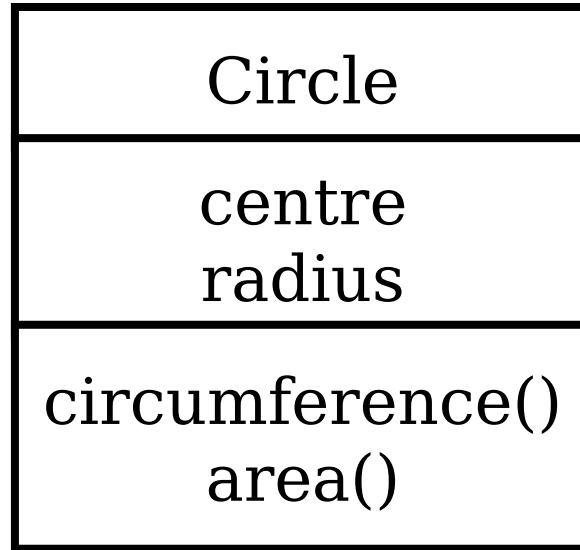


Chapter Three

Classes and Objects in Java

Introduction

- A **class** is a logical framework or blue print or template of an object.
- Anything we wish to represent in Java must be encapsulated in a class
 - defines the “state” and “behavior” of the basic program components known as objects.
- A class is a programmer defined **data type**
- Objects use methods to communicate between them.
- A **class** is a collection of ***fields*** (data) and ***methods***
- **Fields**: say what a class is.
- **Methods**: say what a class does



- The basic syntax for a class definition:

```
class  ClassName [extends SuperClassName]{  
    [fields declaration]  
    [methods declaration]  
}
```

- **Bare bone class** – no fields, no methods
 - The fields and methods are called **class members**
- ```
public class Circle {
 // my circle class
}
```

- **Adding Fields**
- ```
public class Circle {  
    public double x, y; // centre coordinate  
    public double r; // radius of the circle  
}
```

- The fields (data) are also called the **instance variables**.

■ Adding Methods

- A class with only data fields has no life.
 - Objects created by such a class cannot respond to any messages.

- Methods are declared inside the body of the class
- The structure of a method includes a **method signature** and a **code body**:

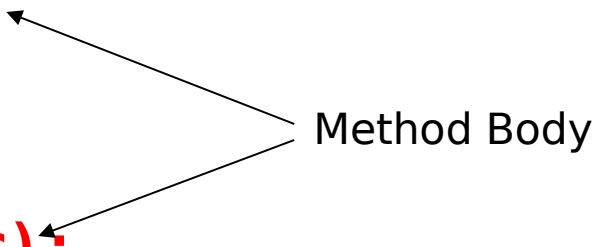
```
[access modifier] returnType methodName([parameters]){  
    //statements,  
    //including local variable declarations  
}
```

- The first line shows a method signature consisting of
 - **access modifier** - determines what other classes and subclasses can invoke this method.

- **Return Type** - what primitive or class type value will return from the invocation of the method.
 - If there is no value to return, use **void** for the return type.
- **Method Name** - The name of the method in which the method is identified with.
- **List of Parameters** - the values passed to the method
- The code body, delineated by the brackets, includes:
 - **Local Variables** - data variables can be declared and used within the method.
 - **Statements** - the code to carry out the task for the particular method

Adding Methods to Class Circle

```
public class Circle {  
    public double x, y; // centre of the circle  
    public double r;    // radius of circle  
  
    //Methods to return circumference and area  
    public double circum() {  
        return (2*3.14*r);  
    }  
    public double area() {  
        return (3.14 * r * r);  
    }  
}
```



Method Body

Object

- It is an instance of a class
- Represents something with which we can interact in a program
- **Creating an object:** It is a two step process

1.creating a reference variable

- **Syntax:**

`<class idn> <ref. idn>;`

e.g. `Circle c1;`

2.Setting or assigning the reference with the newly created object.

- **Syntax:**

`<ref. idn> = new <class idn>(...);`

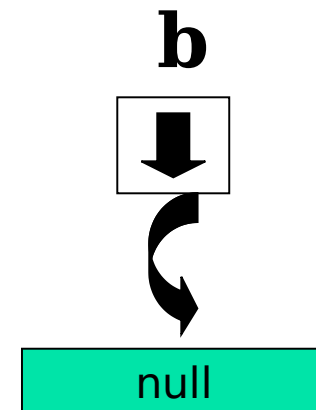
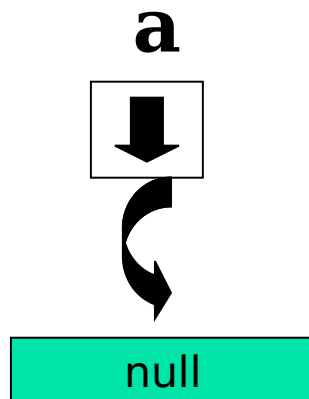
e.g. `c1 = new Circle();`

- The two steps can be done in a single statement
e.g. `Circle c2 = new Circle();`

- An object uniquely identified by
 - its name
 - defined state,
 - represented by the values of its attributes in a particular time

Circle a, b;

- **a and b** simply refer to a Circle object, they are not object themselves.

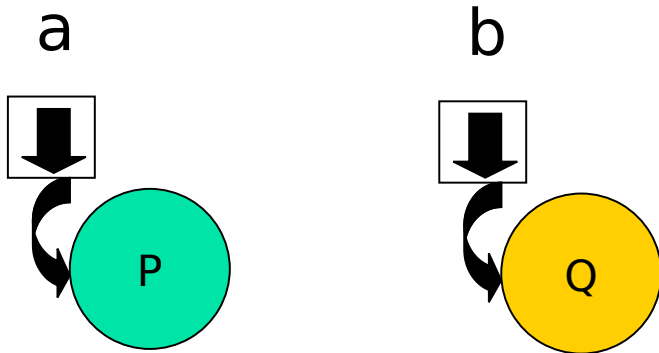


Points to nothing (Null Reference) Points to nothing (Null Reference)

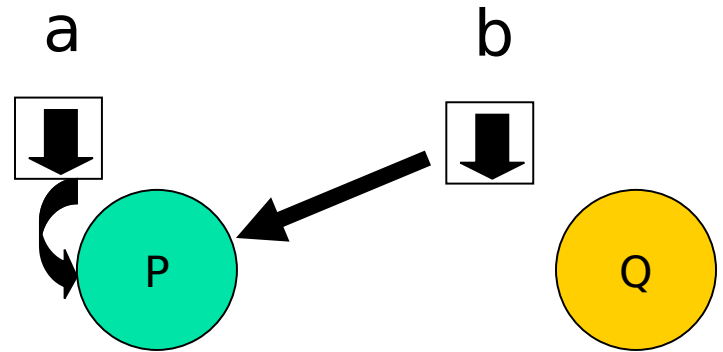
Creating objects of a class

```
a = new Circle();  
b = new Circle() ;  
b = a;
```

Before Assignment



After Assignment



■ Accessing Members of an object

- We use **'.'** (***dot***) operator together with the reference to an object
- *Syntax*:
 - `<ref. idn>.<member>;`

```
Class DriverClass{  
    Public static void main(String args[]){  
        Circle c1 = new Circle();  
        c1.r = 2.3;  
        c1.area();  
        c1.circum();  
    }  
}
```

- Consider the already defined class ***Circle*** and define a driver class

```
class MyMain
```

```
{
```

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

```
    {
```

```
        Circle aCircle;    // creating reference
```

```
        aCircle = new Circle(); // creating object
```

```
        aCircle.x = 10;    // assigning value to data field
```

```
        aCircle.y = 20;
```

```
        aCircle.r = 5;
```

```
        double area = aCircle.area(); // invoking method
```

```
        double circumf = aCircle.circumf();
```

```
        System.out.println ("Radius=" + aCircle.r " Area = " +  
area);
```

```
        System.out.println ("Radius = " + aCircle.r + "Circum  
= " +  
circumf);
```

```
    }
```

```
}
```

Exercise

- Write a program that has two classes. The first class should define a person's name, age, and sex and display these data. The second class should contain the main method.

Person Example

```
class Person {  
    String name = "You";  
    int age = 26;  
    void setName(String n) {  
        name = n;  
    }  
    String getName() {  
        return name;  
    }  
    void setAge(int a) {  
        age = a;  
    }  
    int getAge() {  
        return age;  
    }  
}
```

Constructing Person Objects

- To create an instance of the Person class with a name of **"Tafera"** and an age of **22**

```
Person person = new Person();  
person.setName("Tafera");  
person.setAge(22);
```

- Can we create a Person that has the name **Tafera** and the age **22** from the moment it is created?
- **Answer: Yes!**

Constructors

- are used to initialize the instance variables (fields) of an object
- **Constructors** are special methods used to construct an instance of a class
- **Constructors** are similar to methods, but with some important differences
 - They have the same name as the class
 - They have no return type not even **void**
 - No return statement
 - The *first line* of a constructor must
 - either be a call on another constructor in the same class (using **this**),
 - or a call on the superclass constructor (using **super**)
 - If the first line is neither of these, the compiler automatically inserts a call to the **parameter-less super class constructor**
 - Call the constructor by preceding it with the **new** keyword

Person Constructor

```
class Person {  
    String name;  
    int age;  
  
    Person(String n, int a) {  
        name = n;  
        age = a;  
    }  
  
    // . . .  
}
```

- Now we can construct the previous example as follows:

```
Person person= new Person("Tafera", 22);
```

■ **Types of constructors**

- **Default constructor:** this initializes objects based on default values, takes no argument.
- **Parameterized constructor:** these initialize objects based on some parameter values.
- **Copy constructor:** new objects are initialized based on existing object of the same type.

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.*/
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
}
```

```

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        //declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    } }

```

- When this program is run, it generates the following results:
 - **Constructing Box**
 - **Constructing Box**
 - **Volume is 1000.0**
 - **Volume is 1000.0**
- **Parameterized Constructors**

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

- // declare, allocate, and initialize **Box** objects
 Box mybox1 = new Box(10, 20, 15);
 Box mybox2 = new Box(3, 6, 9);
- each object is initialized as specified in the parameters to its constructor.
- **Default Constructor**
 - When you do not write a constructor in a class, it implicitly has a constructor with no arguments and an empty body
ClassName () { }

■ The **this** Keyword

- can be used inside any method to refer to the *current* object.
- is always a reference to the object on which the method was invoked.
- can use anywhere a reference to an object of the current class' type is permitted.
- consider the following version of **Box()**:

// A redundant use of this.

```
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

■ Instance Variable Hiding

- when a local variable has the same name as an instance variable, the local variable *hides the instance variable*.
- **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

// Use **this** to resolve name-space collisions.

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

Garbage Collection

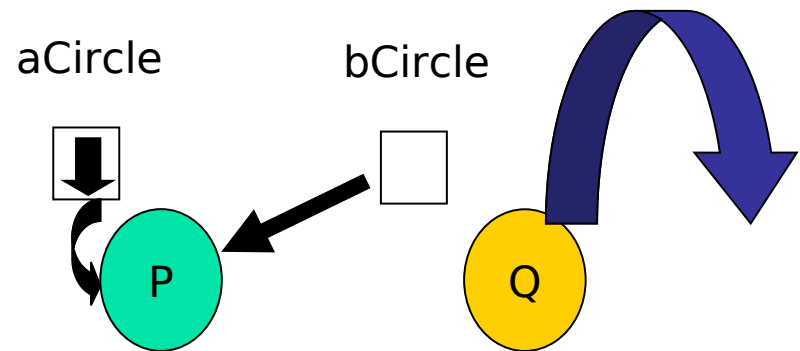
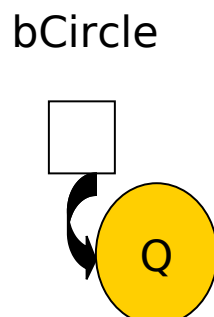
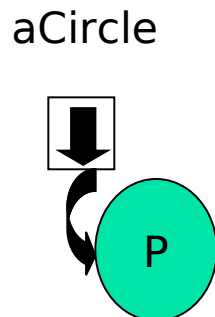
- is the process of automatically finding memory blocks that are no longer being used ("garbage")
- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs sporadically (if at all) during the execution of your program
- E.g.

- **aCircle = new Circle();**

- **bCircle = new Circle() ;**

- **bCircle = aCircle;**
Before Assignment

After Assignment



■ The **finalize()** Method

- Sometimes an object will need to perform some action when it is destroyed.
- Java provides a mechanism called ***finalization***.
- **Finalization** is used to 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.
- 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  
}
```
- the keyword **protected** is access specifier that prevents access to **finalize()** by code defined outside its class
- **finalize()** is only called just prior to garbage collection.

Overloading Methods and

Constructors

■ Overloading Methods

- Two or more methods share the same name
- overloaded methods must differ in the **type and/or number** of their parameters
- 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 implements polymorphism.

- Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
```

```
class OverloadDemo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    // Overload test for one integer parameter.  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
    // Overload test for two integer parameters.  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
}
```

```
double test(double a) {  
    System.out.println("double a: " + a);  
    return a*a;  
}  
  
}  
  
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        double result;  
        ob.test();  
        ob.test(10);  
        ob.test(10, 20);  
        result = ob.test(123.2);  
        System.out.println("Result of ob.test (123.2): " +  
result);  
    } }  
}
```

- This program generates the following output:
 - No parameters
 - a: 10
 - a and b: 10 20
 - double a: 123.2
 - Result of ob.test(123.2): 15178.24
- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.

■ **Overloading Constructors**

- `/* Here, Box defines three constructors to initialize the dimensions of a box various ways. */`

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // constructor used when all dimensions specified  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

```
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class OverloadCons {  
    public static void main(String args[]) {  
        // create boxes using the various constructors  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box();  
        Box mycube = new Box(7);  
        double vol;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
        // get volume of cube  
        vol = mycube.volume();  
        System.out.println("Volume of mycube is " + vol);  
    } }  
}
```

- The output produced by this program is shown here:
 - Volume of mybox1 is 3000.0
 - Volume of mybox2 is -1.0
 - Volume of mycube is 343.0

- **Objects as Parameters**

- consider the following simple program:

// Objects may be passed to methods.

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
}
```

```

// return true if o is equal to the invoking object
boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
}
}
class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " +
ob1.equals(ob2));
        System.out.println("ob1 == ob3: " +
ob1.equals(ob3));
    }
}

```

- This program generates the following output:
ob1 == ob2: true
ob1 == ob3: false
- One of the most common uses of object parameters involves constructors.

■ **Argument Passing**

- there are two ways that a computer language can pass an argument to a subroutine.
- The first way is *call-by-value*.
- The second way an argument can be passed is *call-by-reference*.
 - a reference to an argument (not the value of the argument) is passed to the parameter.
 - changes made to the parameter will affect the argument used to call the subroutine.
- when you pass a simple type to a method, it is passed by value.

- consider the following program:

```
// Simple types are passed by value.
```

```
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}  
  
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.meth(a, b);  
        System.out.println("a and b after call: " + a + " " + b);  
    } }  
}
```

- The output from this program is shown here:
 - a and b before call: 15 20
 - a and b after call: 15 20
- Changes to the object inside the method do affect the object used as an argument.
- For example, consider the following program:

// Objects are passed by reference.

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j; }  
    // pass an object  
    void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2; } }
```

```
class CallByRef {  
    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.meth(ob);  
System.out.println("ob.a and ob.b after call: " + ob.a + " "  
    + ob.b);  
}  
}
```

- This program generates the following output:
 - ob.a and ob.b before call: 15 20
 - ob.a and ob.b after call: 30 10

Cascading Constructors

- A constructor can call another constructor with **this (arguments)**

```
public class Person {  
    int age;  
    String s = new String("null");  
    Person(int a) {  
        age = a;  
        System.out.println("Constructor with int arg only,  
Age= "+ age);  
    }  
    Person(String ss) {  
        System.out.println("Constructor with String arg  
only, s=" + ss);  
        s = ss;  
    }  
}
```

```

Person(String s, int a) {
    this(a);
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
}
Person() {
    this("hi", 47);
    System.out.println("default constructor (no args)");
}
void print() {
    System.out.println("Age = " + age + " s = " + s);
}
public static void main(String[] args) {
    Person x = new Person();
    x.print();
}
}

```

- you can't call a constructor from inside any method other than a constructor.

Exercise

- Write a program that deals with Students.
 - Your program should have at least two classes and it should have more than one methods.
 - Try to apply Constructor Overloading and if necessary, Constructor Cascading.

```
public class MyMath {
    public double PI = 3.14159;
    public double square (double x) {
        return x * x;
    }

    public static void main(String[ ] args) {
        System.out.println("The first Object");
        MyMath m = new MyMath();
        System.out.println("value of PI is " + m.PI);
        System.out.println("square of 5 is " +
            m.square(5));
        System.out.println("The second Object");
        MyMath n = new MyMath();
        System.out.println("value of PI is " + n.PI);
        System.out.println("square of 5 is " +
            n.square(5));
    }
}
```

- The results of invoking square() method on instances m and n are the same:

The First Object

value of PI is 3.14159

square of 5 is 25

The Second Object

value of PI is 3.14159

square of 5 is 25

- square() behaves the same no matter which instance it is called on.
- So . . . why not have one square() method for the entire class?

Understanding 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.
- When objects of its class are declared, no copy of a static variable is made.
- all instances of the class share the same static variable.
- Two kind of methods
 - Instance Methods
 - Static Methods

■ **Instance methods**

- associated with an object
- use the instance variables of that object
- the default
- called by prefixing it with an *object*
 - E.g Circle a1 = new Circle()
A1.area();

■ **Static Methods:**

- They can only call other static methods.
- They must only access static data.
- They cannot refer to **this** or **super** in any way.
- Can't access instance variables of any object
- Calling static methods
 - Called from within the same class: Just write the static method name
 - E.g. **double avgAtt = mean(attendance);**
 - Called from outside the class: by prefixing it with a *class name*
 - E.g. **Math.max(i, j)**

MyMath with static

```
public class MyMath {  
    public static double PI = 3.14159;  
    public static double square (double x) {  
        return x * x;  
    }  
  
    public static void main( String[ ] args) {  
        // invoke square() method on the  
MyMath class  
        System.out.println("Value of PI is " +  
PI);  
        System.out.println("Square of 5 is" +  
square(5));  
    }  
}
```

- a **static block**
 - is used to initialize static variables
 - which gets executed exactly once, when the class is first loaded.
- The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
```

```
class StaticBlock{  
    private int x;  
    private static int y, z;  
    StaticBlock(){  
        x = y + z;  
        y = z = 40 ; }  
    static{  
        y = 10;  
        z = 20 ;  
        System.out.println("Inside static block");    }  
}
```

```

public static void main(String args[]){
    System.out.println("Before
object creation " + y + " " + z);
    StaticBlock s = new StaticBlock();
    System.out.println("After object
creation " + y
                                + " " + z + "
" + s.x);
}
}

```

- Here is the output of the program:
 Inside Static Block
 Before object creation 10 20
 After object creation 40 40 30

- What if the driver class is in a separate class?

```
class StaticBlock{  
    int x;  
    static int y, z;  
    StaticBlock(){  
        System.out.println("Inside Constructor");  
        x = y + z;  
        y = z = 40 ;  
    }  
    static{  
        y = 10;  
        z = 20 ;  
        System.out.println("Inside static block");  
    }  
}
```

```

class MainTest{
    public static void main(String args[]){
        System.out.println("Before object creation");
        System.out.println(StaticBlock.y+" " +StaticBlock.z);
        StaticBlock s = new StaticBlock();
            System.out.println("After object creation" +
                               StaticBlock.y + " "+ StaticBlock.z + " " +
                               s.x);
        }
    }
}

```

■ Output:

Before object creation

Inside static block

10 20

Inside Constructor

After object creation40 40 30

- static methods and variables can be used independently of any object.
- you need only specify the name of their class followed by the dot operator.
- The general form:

classname.method()

- A static variable can be accessed in the same way —by use of the dot operator on the name of the class

Access Control

- Through encapsulation, you can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse
- How a member can be accessed is determined by the ***access specifier*** that modifies its declaration
- Some aspects of access control are related mostly to inheritance or packages.
- Java's access specifiers are: **public**, **private**, and **protected**
- Java also defines a default access level.
- **protected** applies only when inheritance is involved.
- **Public** - member can be accessed by any other code in your program
- **Private** - member can only be accessed by other members of its class.

- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- To understand the effects of public and private access, consider the following program:

```
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());
    }
}

```

The **final** keyword

- We declared PI as

```
public static double PI = 3.14159;
```

but this does not prevent changing its value:

```
MyMath.PI = 9999999999;
```

- Use keyword **final** to denote a constant :

```
public static final double PI = 3.14159;
```

- Once we declare a variable to be **final**, it's value can no longer be changed!
- The keyword **final** can also be applied to methods,
- but its meaning is substantially different than when it is applied to variables.

What is Package ?

- is a structure for containing a group of related classes
- is both a namespace management as well as visibility control.
- To resolve the name conflicts between class names
- A package name implies the directory structure where files reside.
- There is zero or one package declaration per class
- Must be first non-comment statement
- **Advantages of using packages**
 - reduce the complexity of application components
 - Software reuse
 - Solves the problem of unique class name conflict

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.
- the general form of the **package** statement:
Package *packageName*;
- For example:
package MyPackage;
- Java uses file system directories to store packages.

■ Working with classes inside packages

- To access classes in a package we should use:

1. Fully qualified class name

- *Syntax:*

<pack_name>.<class Idn.>

```
class TestPackage {  
    public static void main(String args[]) {  
        Rectangle r = new Rectangle(); //error  
        Figure.Rectangle r = new Figure.Rectangle();  
        . . .  
    }  
}
```

2. Using import

- Zero, one or many import statements per program.
- Must precede any class definition.
- *Syntax:*

import <pack_name>.<* | class Idn.>

- The import statement is used to bring
 - an entire package (i.e all classes in a package) or
 - a single class into your program

```
import figure.*;
```

```
class TestFigures{
```

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

```
    Rectangle r = new Rectangle();    Circle c1 = new  
Circle();
```

```
    . . .
```

```
    }
```

```
}
```

- 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:

`package pkg1[.pkg2[.pkg3]];`

- For example, a package declared as:

`package java.awt.image;`

- You cannot rename a package without renaming the directory in which the classes are stored.

```
// A simple package
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        if(bal<0)
            System.out.print("—> ");
        System.out.println(name + ": $" + bal);
    }
}
```

```
class AccountBalance {  
    public static void main(String args[]) {  
        Balance current[] = new Balance[3];  
        current[0] = new Balance("K. J. Fielding", 123.23);  
        current[1] = new Balance("Will Tell", 157.02);  
        current[2] = new Balance("Tom Jackson", -12.33);  
        for(int i=0; i<3; i++)  
            current[i].show();  
    }  
}
```

- you cannot use this command line:

java AccountBalance

- **AccountBalance** must be qualified with its package name.

Overview of java.lang

- Provides classes that are considered fundamental to the design of the Java programming language.
- The **java.lang** package does NOT have to be imported
- **Classes of java.lang**
 - Object
 - String
 - StringBuffer
 - System
 - The “Wrapper” Classes
 - Math