

Java classes

Class and Object

- A class provides template for an object
- An object is an instance of a class.
- In run-time, a number of instances of class is created
- The state of an object represented by data fields (properties or attributes) with their current values
- The behavior of an object (its actions) is defined by methods

A simple form of class

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

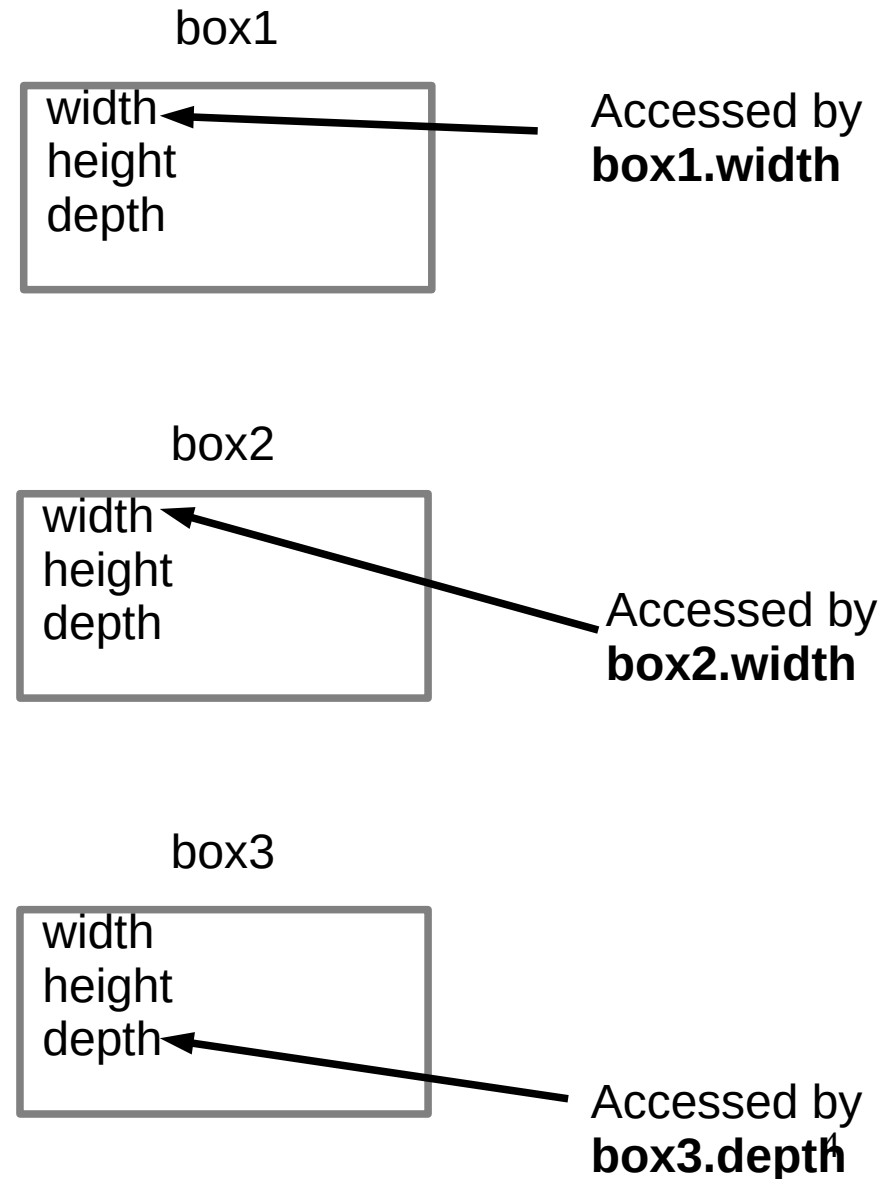
The diagram illustrates the structure of a class. A large curly brace on the right side groups the variables and methods together, with the label "Members of the class" pointing to it. A smaller curly brace on the right side groups the variables together, with the label "variables" pointing to it. Another smaller curly brace on the right side groups the methods together, with the label "methods" pointing to it. Arrows point from the "variables" and "methods" labels to the corresponding code blocks in the class definition.

Example

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

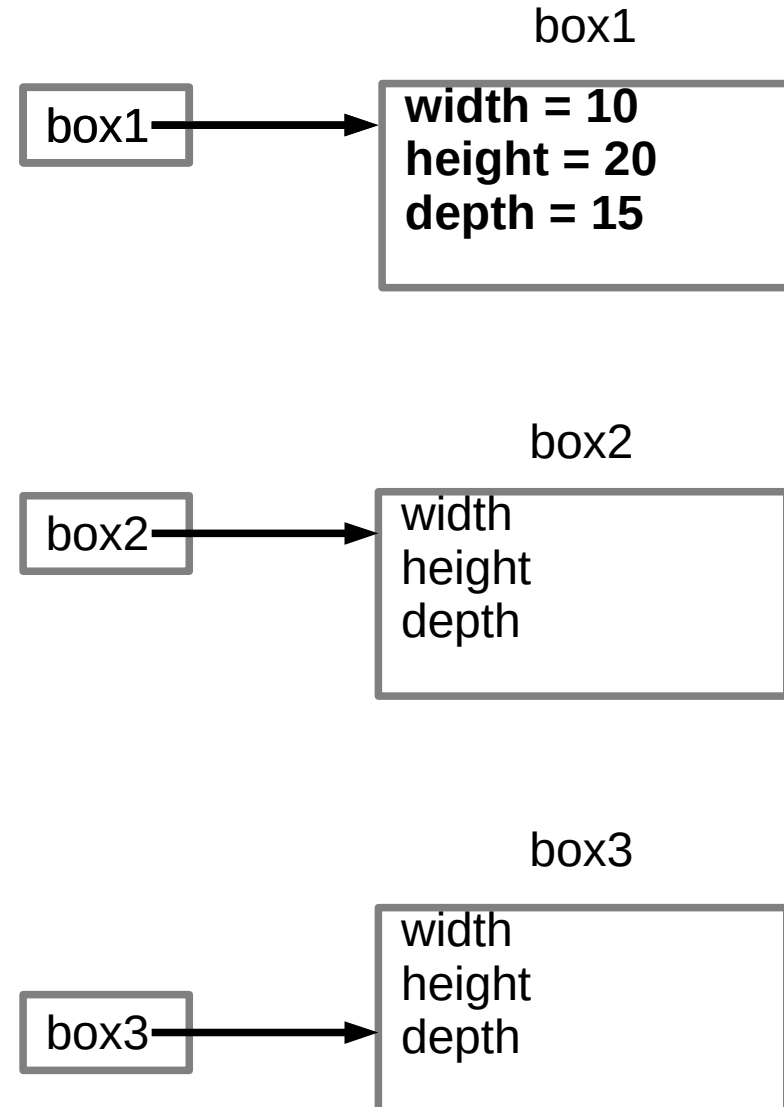
Creating objects of Box class

```
Box box1 = new Box();  
Box box2 = new Box();  
Box box3 = new Box();
```



Example

```
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box box1 = new Box();  
        Box box2 = new Box();  
        Box box3 = new Box();  
  
        box1.width = 10;  
        box1.height = 20;  
        box1.depth = 15;  
  
        box2.width = 12;  
        box2.height = 30;  
        box2.depth = 20;  
  
        double vol;  
        vol = box1.width * box1.height * box1.depth;  
        System.out.println("Volume of box1 is " + vol);  
  
        vol = box2.width * box2.height * box2.depth;  
        System.out.println("Volume of box2 is " + vol);  
    }  
}
```



Example(2)

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

```
class BoxDemo {  
    public static void main(String args[]) {  
        Box box1 = new Box();  
        Box box2 = new Box();  
        Box box3 = new Box();
```

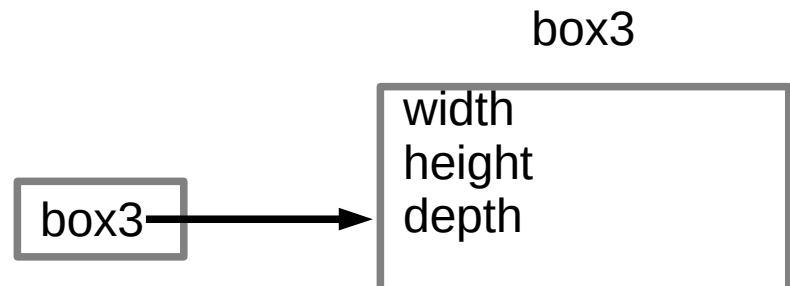
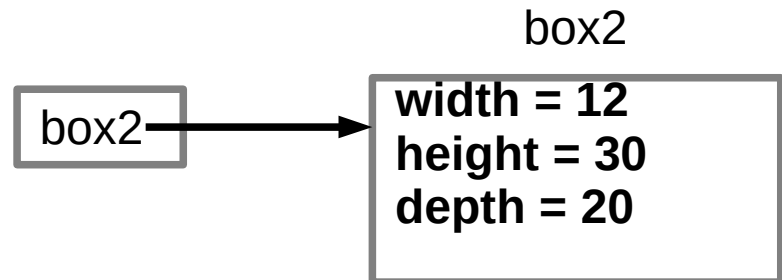
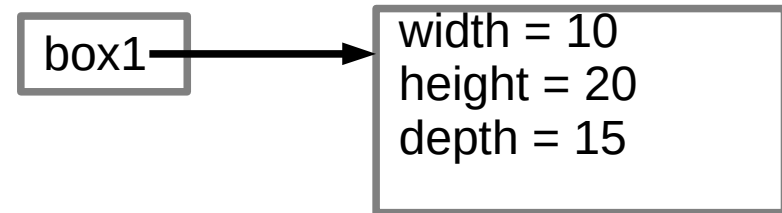
```
        box1.width = 10;  
        box1.height = 20;  
        box1.depth = 15;
```

```
        box2.width = 12;  
        box2.height = 30;  
        box2.depth = 20;
```

```
        double vol;  
        vol = box1.width * box1.height * box1.depth;  
        System.out.println("Volume of box1 is " + vol);
```

```
        vol = box2.width * box2.height * box2.depth;  
        System.out.println("Volume of box2 is " + vol);
```

```
    }  
}
```



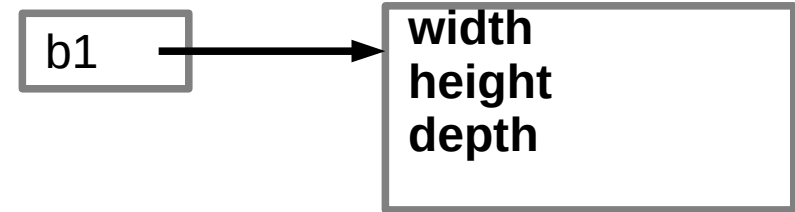
Two steps involved in object declaration

- `Box box1;`
 - Declare a reference to object
 - Its simply a variable that can refer to an object
- `box1 = new Box();`
 - `new` operator dynamically allocates memory for object of type `Box` and returns a reference to it.
 - `Box()` specifies the constructor for the class `Box`.
 - A constructor defines what occurs when an object of a class is created.

What will be result of the following statement?

```
Box b1 = new Box();
```

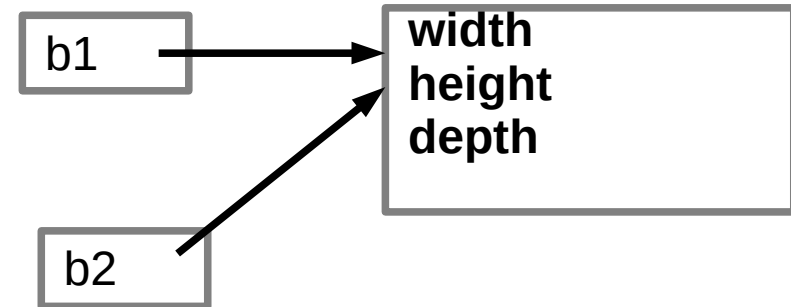
```
Box b2 = b1;
```



Assignment of object reference variable

```
Box b1 = new Box();
```

```
Box b2 = b1;
```



- The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
- It simply makes b2 refer to the same object as does b1.
- Any changes made to the object through b2 will affect the object to which b1 is referring

Method example

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
  
        // display volume of first box  
        mybox1.volume();  
        // display volume of second box  
        mybox2.volume();  
    }  
}
```

Method with parameters

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

```
class BoxDemo5 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        // initialize each box  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 9);  
  
        // 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);  
    }  
}
```

Another example

```
public class TV {  
    int channel = 1; // Default channel is 1  
    int volumeLevel = 1; // Default volume level is  
1  
    boolean on = false;  
  
    public void turnOn() {  
        on = true;  
    }  
    public void turnOff() {  
        on = false;  
    }  
    public void setChannel(int newChannel) {  
        if (on && newChannel >= 1  
            && newChannel <= 120)  
            channel = newChannel;  
    }  
    public void setVolume(int newVolumeLevel) {  
        if (on && newVolumeLevel >= 1  
            && newVolumeLevel <= 7)  
            volumeLevel = newVolumeLevel;  
    }  
}
```

```
    public void channelUp() {  
        if (on && channel < 120)  
            channel++;  
    }  
    public void channelDown() {  
        if (on && channel > 1)  
            channel--;  
    }  
    public void volumeUp() {  
        if (on && volumeLevel < 7)  
            volumeLevel++;  
    }  
    public void volumeDown() {  
        if (on && volumeLevel > 1)  
            volumeLevel--;  
    }  
}
```


Constructor

- Java allows objects to initialize themselves when they are created using a constructor.
- Constructors have the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately when the object is created, before the new operator completes.
- Constructors have no return type because, the implicit return type of a class constructor is the class type itself.

Example of constructor

```
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;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}
```

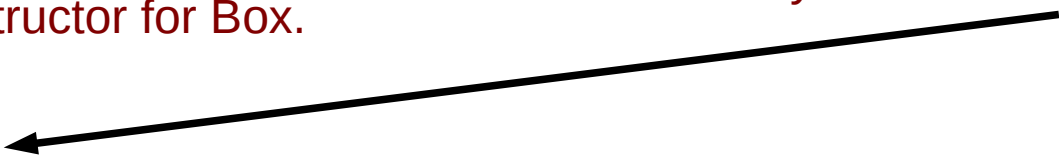
Box mybox1 = new Box();



Example of parameterized constructor

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // This is the constructor for Box.  
  
    Box(double w, double h, double d) {  
        System.out.println("Constructing Box");  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Box mybox1 = new Box(10, 20, 15);



The **this** Keyword

- Sometimes a method will need to refer to the object that invoked it.
- Java defines the **this** keyword can be used inside any method to refer to the current object.
- **this** is always a reference to the object on which the method was invoked.
- Can be used anywhere as reference to an object of the current class' type.

Consider the following

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // This is the constructor for Box.  
  
    Box(double width, double height, double depth)  
    {  
        width = 10; which width ?  
        height = 10; which height?  
        depth = 10; which depth?  
    }  
}
```

Use of this to resolve namespace-collision

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

Garbage Collection

- Automatic memory deallocation technique is called **garbage collection**.
- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- Different Java run-time implementations will take varying approaches to garbage collection

The finalize() Method

finalize method is a predefined method in java and is defined in Object Class and it is protected.

- It is the alternate term for destructors in C++
- It defines an action to be performed when an object is destroyed
- The mechanism is called **finalization**
- Executed when an object is just about to be reclaimed by the garbage collector
- to add a finalizer to a class, we define the **finalize()** method

```
protected void finalize( )
```

```
{
```

```
    // finalization code here
```

```
}
```

No return type and no parameters are passed to finalize method. we can override finally method. It is automatically called by JVM once but if we are overriding we need to explicitly call it.

Example

```
class Box {  
    double width;  
    double height;  
    double depth;  
    Box(double w, double h, double d) {  
        System.out.println("Constructing Box");  
        width = 10;  
        height = 10;  
        depth = 10;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
    public void finalize() {  
        System.out.println("box object destroyed");  
        System.out.println("Inside finalize of Box");  
    }  
}
```

Polymorphism via method overloading

- Two or more methods within the same class that share the same name is referred as the process of method overloading.
- Java's one way to support polymorphism.
- Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- **NOTE:** Different return types of overloaded method are insufficient to distinguish two versions of a method.
- Java simply executes the version of the method whose parameters match the arguments used in the call.

Method overloading example

```
class Overload{  
    void display(){  
        System.out.println("no parameter");  
    }  
    void display(int a){  
        System.out.println("a =" +a);  
    }  
    void display(double a){  
        System.out.println("a =" +a);  
    }  
    void display(int a, int b){  
        System.out.println("a =" +a+ " b =" +b);  
    }  
    void display(double a, double b){  
        System.out.println("a =" +a+ " b =" +b);  
    }  
}
```

```
...  
Overload obj1 = new Overload();  
...  
...  
..  
obj1.display();  
obj1.display(20,30);  
obj1.display(30);  
obj1.display(33.5,19.33);  
...  
...  
...
```

Which method ??

```
class Overload{  
    void display(){  
        System.out.println("no parameter");  
    }  
    void display(double a){  
        System.out.println("a =" + a);  
    }  
    void display(int a, int b){  
        System.out.println("a =" + a + " b =" + b);  
    }  
    void display(double a, double b){  
        System.out.println("a =" + a + " b =" + b);  
    }  
}
```

```
...  
Overload obj1 = new Overload();  
...  
...  
..  
obj1.display();  
obj1.display(20,30);  
obj1.display(30); ??  
obj1.display(33.5);  
...  
...  
...
```


Automatic type conversion

```
class Overload{  
    void display(){  
        System.out.println("no parameter");  
    }  
    void display(double a){  
        System.out.println("a =" +a);  
    }  
    void display(int a, int b){  
        System.out.println("a =" +a+ " b =" +b),  
    }  
    void display(double a, double b){  
        System.out.println("a =" +a+ " b =" +b);  
    }  
}
```

```
...  
Overload obj1 = new Overload();  
...  
...  
..  
obj1.display();  
obj1.display(20,30);  
obj1.display(30);  
obj1.display(33.5);  
...  
...  
...
```

Overloading Constructors

```
// Recall Box class
```

```
...
```

```
...
```

```
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;  
  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
  
        vol = mycube.volume();  
        System.out.println("Volume of mycube is " + vol);  
    }  
}
```

Using Objects as Parameters

```
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));
    }
}
```

Object passing to constructors

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // construct clone of an object  
    Box(Box ob) { // pass object to constructor  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    ...  
    ...
```

...

...

```
Box mybox1 = new Box(10, 20, 15);
```

```
Box myclone = new Box(mybox1);
```

Java way of passing parameter

- When you pass a primitive type to a method, it is passed by value
- When you pass an object to a method
 - Objects are passed effectively by call-by-reference.
 - The parameter that receives a reference to an object will refer to the same object as that referred to by the argument.

Object pass 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);  
    }  
}
```

Returning Objects

```
class Test {  
    int a;  
  
    Test(int i) {  
        a = i;  
    }  
  
    Test incrByTen() {  
        Test temp = new Test(a+10);  
        return temp;  
    }  
}
```

```
class RetOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(2);  
        Test ob2;  
  
        ob2 = ob1.incrByTen();  
        System.out.println("ob1.a: " + ob1.a);  
        System.out.println("ob2.a: " + ob2.a);  
  
        ob2 = ob2.incrByTen();  
        System.out.println("ob2.a after second  
            increase: "+ ob2.a);  
    }  
}
```

Access Control

- Encapsulation a feature of OOP that provides Access Control
- Access control can prevent misuse of data
- 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 and private

- When a member of a class is specified as **public**, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class
- When no access 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
- Example :
 - `public int i;`
 - `private double j;`
 - `private int myfunction(int a, char b) { // ... }`

Example

```
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();  
  
        ob.a = 10;        //ok?  
        ob.b = 20;        //ok?  
        ob.c = 100;       // ok??  
  
        ob.setc(100); // OK??  
  
        System.out.println("a, b, and c: " + ob.a + " " +  
                           ob.b + " " + ob.getc());  
    }  
}
```

static

- It is possible to create a member that can be used by itself, without reference to a specific instance.
- Such a member, precedes its declaration with the keyword static.
- A static member can be accessed before any objects of its class are created, and without reference to any object.
- Both methods and variables can be declared to static.
- The most common example of a static member is main().
- Instance variables declared as static are essentially, global variables for a class.
- When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

static method

- Methods declared as static have several restrictions:
 - They can only call other static methods.
 - They must only access static data.
 - They cannot refer to **this** or **super** in any way.
 - If you need to do computation in order to initialize your static variables, you can declare a **static block** that gets executed exactly once, when the class is first loaded.

Example of static

```
class UseStatic {  
    static int a = 3;  
    static int b;  
  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;  
    }  
  
    public static void main(String args[]) {  
        meth(42);  
    }  
}
```

Accessing static outside its own class

- Outside of the class in which they are defined, static methods and variables can be used by specifying the name of their class followed by the dot operator.
 - `classname.method()`
- This is how Java implements a controlled version of global methods and global variables

Example

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}
```

```
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

The 'final' keyword

- A variable can be declared as **final** to prevent its content from being modified.
- We must initialize a final variable when it is declared.
- It is a common coding convention to choose all uppercase identifiers for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.
- Thus, a final variable is essentially a constant.
- The keyword final can also be applied to methods, but its meaning is substantially different
- Example
 - `final int PI = 3.14;`
 - `final int FILE_NEW = 1;`

Nested class

- The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested.
- However, the enclosing class does not have access to the members of the nested class.
- It is also possible to declare a nested class that is local to a block.

Nested class(2)

- There are two types of nested classes:
 - Static
 - Non-static (Inner class)
- A static nested class is one that has the static modifier applied.
- Because it is static, it must access the members of its enclosing class through an object.
- Because of this restriction, static nested classes are seldom used.
- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example

```
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an innner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output??

```
class Outer {  
    int outer_x = 100;  
  
    void test() {  
        Inner inner = new Inner();  
        inner.display();  
    }  
}
```

// this is an inner class

```
class Inner {  
    int y = 10; // y is local to Inner  
    void display() {  
        System.out.println("display: outer_x = " + outer_x);  
    }  
}
```

```
void showy() {  
    System.out.println(y);  
}  
}
```

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```