

Java Classes	Access Modifiers	Defining Methods	Constructors	Passing Primitive Data Type Arguments
Passing Reference Data Type Arguments	Objects	Using Objects	Copy Constructor in Java	Understanding Class Members (Static Keyword)
Java destructor – Why is it missing?				

Similarities with C++	Differences from C++
<ul style="list-style-type: none"> User-defined classes can be used like built-in types. Basic syntax are same 	<ul style="list-style-type: none"> Methods (member functions) are the only function type Object is the topmost ancestor for all classes In C++, all non-static member functions by default are non virtual. If you need to make a function virtual, you must exclusively specify it. <p>But in Java, all non-static member functions by default are virtual (no keyword needed) and to make a function non-virtual, you must exclusively make it final. So all non-static methods use dynamic binding.</p> <ul style="list-style-type: none"> All objects are allocated on the heap. Single inheritance only Java 8 has multiple inheritance, but via interfaces not by normal classes, so is a bit of a nonstandard variation of multiple inheritance

Java Classes

A sample code for a possible implementation of a Bicycle class is given below :

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;   public int gear;   public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;    cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) { cadence = newValue; }
    public void setGear(int newValue) { gear = newValue; }
    public void applyBrake(int decrement){ speed -= decrement; }
```

```
public void speedUp(int increment) { speed += increment; }
}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass has one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass has one method
    public void setHeight(int newValue) { seatHeight = newValue; }

}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight` and a method to set it.

A **class declaration** is shown above. The *class body* contains all the code that provides for the life cycle of the objects created from the class. It has at least three components (required).

- **Constructors** for initializing new objects,
- Declarations for the **fields** that provide the state of the class and its objects
- **Methods** to implement the behavior of the class and its objects.

You can provide more information about the class, such as the **name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration.** For example,

```
class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

Means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*

A class name can be used in Java as the **type** of a field or local variable or as the return type of a function (method). There are also fancy uses with **generic** types such as `List<String>`.

Access Modifiers

The modifier controls what other classes have access to a member. Following modifiers are available in Java:

- **public** —the field is accessible from all classes.
- **private** —the field is accessible only within its own class.
- **protected** - the field is accessible only within its own class, subclass (even in different package).
- **package-private** (no explicit modifier) - accessible to all classes within that package.

The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

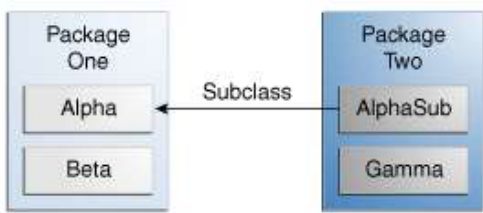
As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.

Access modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the **top level**—**public**, or **package-private** (no explicit modifier). A **class** may be declared with the modifier **public**, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as **package-private**), it is visible only within its **own package**.
- At the member level—**public**, **private**, **protected**, or **package-private** (no explicit modifier).

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.

Classes and Packages of the Example Used to Illustrate Access Levels



The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility				
Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

It is common to make fields **private** (to encapsulate). This means that they can only be **directly accessed from the Bicycle class**. For accessing by other class **add public methods** that obtain the field values for us:

```
public class Bicycle {
    private int cadence; private int gear; private int speed;
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear; cadence = startCadence; speed = startSpeed;    }
}
```

```

public int getCadence() { return cadence; }
public void setCadence(int newValue) { cadence = newValue; }
public int getGear() { return gear; }
public void setGear(int newValue) { gear = newValue; }
public int getSpeed() { return speed; }
public void applyBrake(int decrement) { speed -= decrement; }
public void speedUp(int increment) { speed += increment; }
}

```

Types : All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

Defining Methods

Here is an example of a typical method declaration:

```

public double calculateAnswer(double wingSpan, int numberOfEngines,
    double length, double grossTons) {
    //do the calculation here
}

```

More generally, method declarations have **six** components, in order:

1. Modifiers—such as public, private etc.
2. The return type—the data type of the value returned by the method, or **void** if the method does not return a value.
3. The method name—the rules for field names apply to method names as well.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, ().
5. An **exception** list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

The signature of the method declared above is: **calculateAnswer(double, int, double, double)**

Typically, a method has a unique name within its class. **However, a method might have the same name as other methods due to *method overloading*.**

Overloading Methods

Java can distinguish between methods with **different *method signatures***. This means that methods within a class can have the same name if they have **different parameter lists**. For example :

```

public class DataArtist {
    ...
    public void draw(String s) { ... }
    public void draw(int i) { ... }
    public void draw(double f) { ... }
    public void draw(int i, double f) { ... }
}

```

- Overloaded methods are differentiated by the **number** and the **type** of the arguments passed into the method.
- The compiler **does not consider return type when differentiating methods**, so you cannot declare two methods with the same signature even if they have a different return type.

Constructors

Constructor declarations look like method declarations—except that they use the **name** of the class and **have no return type**. Constructor code gets executed when “**new**” is called

```
public Bicycle(int startCadence, int startSpeed, int startGear) {  
    gear = startGear;  cadence = startCadence;  speed = startSpeed;  
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8); // creates space in memory for the object and initializes fields.
```

Although Bicycle only has one constructor, it could have others, including a **no-argument constructor**:

```
public Bicycle() { gear = 1;  cadence = 10;  speed = 0; }
```

- The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of **Object**, which *does* have a no-argument constructor.
- You can use **access modifiers** in a constructor's declaration to control which other classes can call the constructor.

Arbitrary Number of Arguments in a method

You can use a construct called **varargs** to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. **It's a shortcut to creating an array manually**.

To use varargs, you follow the **type of the last parameter by an ellipsis (three dots, ...)**, then a space, and the **parameter name**. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {  
    int numberOfSides = corners.length;  
    double squareOfSide1, lengthOfSide1;  
    squareOfSide1 = (corners[1].x - corners[0].x)  
        * (corners[1].x - corners[0].x)  
        + (corners[1].y - corners[0].y)  
        * (corners[1].y - corners[0].y);  
    lengthOfSide1 = Math.sqrt(squareOfSide1);  
  
    // more method body code follows that creates and returns a
```

Inside the method, **corners** is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

```
// polygon connecting the Points  
}
```

You will most commonly see varargs with the printing methods; for example, this `printf` method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

Parameter Names

- A parameter can have the same name as one of the class's fields.
- If same, the parameter is said to **shadow** the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

```
public class Circle {  
    private int x, y, radius;  
    public void setOrigin(int x, int y) {  
        ...  
    }  
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields.

- Using the simple names x or y within the body of the method refers to the parameter, *not* to the field.
- To access the field, you must use a qualified name by using the `this` keyword.

Passing Reference Data Type Arguments

- Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the **scope of the method**. When the method returns, the parameters are gone and any changes to them are lost.
- Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the **passed-in reference still references the same object as before**. However, the values of the object's fields *can* be changed in the method, if they have the proper access level. For example, consider a method in an arbitrary class that moves Circle objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {  
    // code to move origin of circle to x+deltaX, y+deltaY  
    circle.setX(circle.getX() + deltaX);  
    circle.setY(circle.getY() + deltaY);  
}
```

```
// code to assign a new reference to circle
```

```
circle = new Circle(0, 0);  
}
```

Let the method be invoked with these arguments: `moveCircle(myCircle, 23, 56)`

- The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively.
- These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.

Objects

Here's a small program, called CreateObjectDemo, that creates three objects: one Point object and two Rectangle objects. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

```
public class CreateObjectDemo {  
  
    public static void main(String[] args) {  
  
        // Declare and create a point object and two rectangle objects.  
        Point originOne = new Point(23, 94);  
        Rectangle rectOne = new Rectangle(originOne, 100, 200);  
        Rectangle rectTwo = new Rectangle(50, 100);  
  
        // display rectOne's width, height, and area  
        System.out.println("Width of rectOne: " + rectOne.width);  
        System.out.println("Height of rectOne: " + rectOne.height);  
        System.out.println("Area of rectOne: " + rectOne.getArea());  
  
        rectTwo.origin = originOne;    // set rectTwo's position  
  
        // display rectTwo's position  
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);  
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);  
  
        // move rectTwo and display its new position  
        rectTwo.move(40, 72);  
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);  
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);  
    }  
}
```

This program creates, manipulates, and displays information about various objects. Here's the output:

```
Width of rectOne: 100  
Height of rectOne: 200  
Area of rectOne: 20000  
X Position of rectTwo: 23  
Y Position of rectTwo: 94  
X Position of rectTwo: 40  
Y Position of rectTwo: 72
```

The following three sections use the above example to describe the **life cycle of an object** within a program

Creating Objects

```
Point originOne = new Point(23, 94);  
Rectangle rectOne = new Rectangle(originOne, 100, 200);  
Rectangle rectTwo = new Rectangle(50, 100);
```

Each of statements taken from the CreateObjectDemo program creates an object and assigns it to a variable.

Each of these statements has three parts (discussed in detail below):

- Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.
- Instantiation: The **new** keyword is a Java operator that creates the object.
- Initialization: The new operator is followed by a **call to a constructor**, which initializes the new object.

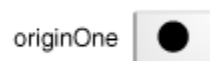
Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write: **type name;**

This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also **reserves the proper amount of memory** for the variable.

You can also declare a **reference** variable on its own line. For example: **Point originOne;**

If you declare **originOne** like this, its value will be undetermined until an object is actually created and assigned to it. **Simply declaring a reference variable does not create an object**. A variable in this state, which currently references no object, can be illustrated as follows (the variable name, originOne, plus a reference pointing to nothing):



Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. This reference is usually assigned to a variable of the appropriate type as shown below. **The new operator also invokes the object constructor.**

```
Point originOne = new Point(23, 94);
```

The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example: **int height = new Rectangle().height;**

Initializing an Object : Here's the code for the Point class:

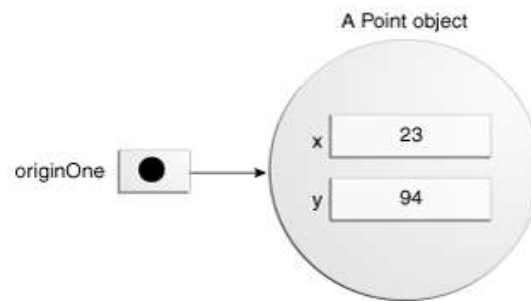
```
public class Point {  
    public int x = 0;    public int y = 0;  
    //constructor  
    public Point(int a, int b) { x = a; y = b; }  
}
```

```
Point originOne = new Point(23, 94);
```

Here's the code for the Rectangle class,
which contains four constructors:

```
public class Rectangle {  
    public int width = 0;    public int height = 0;    public Point origin;  
  
    // four constructors  
    public Rectangle() { origin = new Point(0, 0); }  
    public Rectangle(Point p) { origin = p; }  
    public Rectangle(int w, int h) { origin = new Point(0, 0);    width = w;    height = h; }  
    public Rectangle(Point p, int w, int h) { origin = p; width = w; height = h; }  
  
    // a method for moving the rectangle  
    public void move(int x, int y) { origin.x = x; origin.y = y; }  
  
    // a method for computing the area of the rectangle  
    public int getArea() { return width * height; }  
}
```

The result of executing this statement
can be illustrated in the next figure:

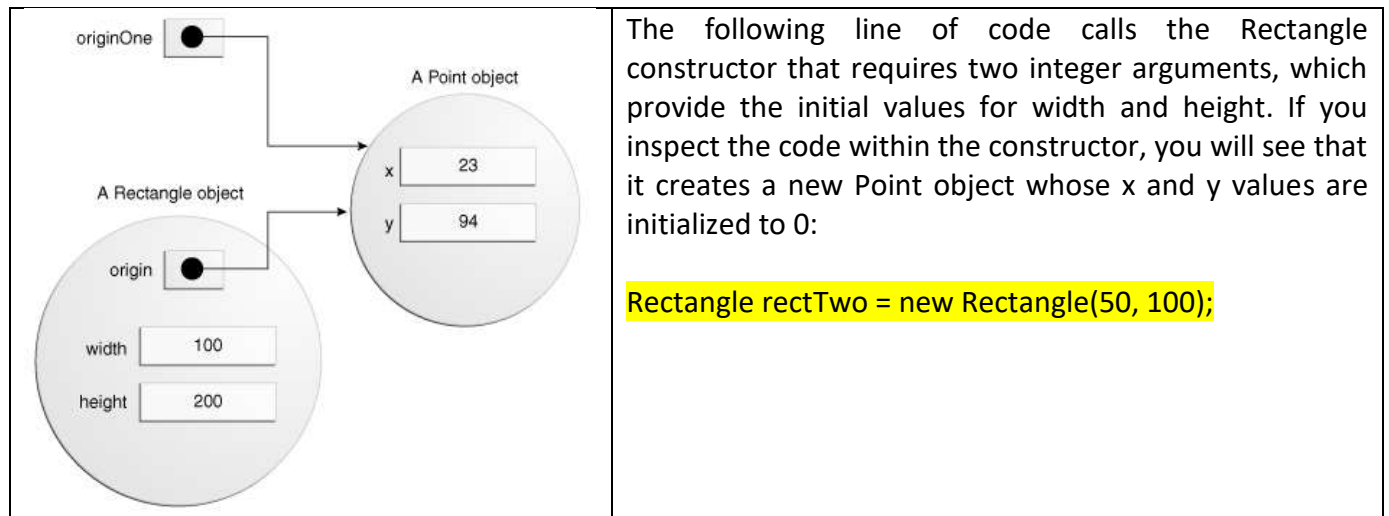


Each constructor lets you
provide initial values for the
rectangle's origin, width, and
height, using both **primitive**
and **reference** types.

When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the same Point object—an object can have multiple references to it, as shown in the next figure:



Using Objects

Referencing an Object's Fields

Object fields are accessed by their name. Code that is outside the object's class must use an **object reference** or expression, followed by the **dot (.) operator**, followed by a simple field name, as in:

`objectReference.fieldName`

```
System.out.println("Width of rectOne: " + rectOne.width);  
System.out.println("Height of rectOne: " + rectOne.height);
```

You can use any expression that returns an object reference. Recall that the `new` operator returns a reference to an object. So you could use the value returned from `new` to access a new object's fields:

```
int height = new Rectangle().height;
```

Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

`objectReference.methodName(argumentList);` or: `objectReference.methodName();`

The Rectangle class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the CreateObjectDemo code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());    rectTwo.move(40, 72);
```

The new operator returns an object reference, so you can use the value returned from new to invoke a new object's methods: `new Rectangle(100, 50).getArea()`

Remember, invoking a method on a particular object is the same as sending a message to that object.

The Garbage Collector

The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value **null**. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

Returning a Value from a Method

- You declare a method's return type in its method declaration.
- Within the body of the method, you use the **return** statement to return the value. Any method declared **void** doesn't return a value. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this: **return;**
- If you try to return a value from a method that is declared **void**, you will get a compiler error.
- The **data type** of the return value must match the method's declared return type.

```
// a method for computing the area of the rectangle
public int getArea() { return width * height; }
```

The getArea method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate Bicycle objects, we might have a method like this:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                             Environment env) {
    Bicycle fastest;
    // code to calculate which bike is faster, given each bike's gear
    // and cadence and given the environment (terrain and wind)
    return fastest;
}
```

Covariant return types in Java (not in syllabus)

- Before JDK 5.0, it was not possible to **override** a method by **changing the return type**. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be **exactly same** as that of parent class method. Overriding method was said to be **invariant** with respect to return type.
- Java 5.0 onwards** it is possible to have **different return type** for an overriding method in child class, but child's return type should be **sub-type of parent's return type**. Overriding method becomes **variant** with respect to return type.

Below is the simple example to understand the co-variant return type with method overriding.

// Two classes used for return types.

class A {} and class B extends A {}

<pre>class Base { A fun() { System.out.println("Base fun()"); return new A(); } }</pre>	<pre>class Derived extends Base { B fun() { System.out.println("Derived fun()"); return new B(); } }</pre>	<pre>public class Main { public static void main(String args[]) { Base base = new Base(); base.fun(); Derived derived = new Derived(); derived.fun(); } }</pre>
--	---	--

Output:
Base fun()
Derived fun()

Note : If we swap return types of Base and Derived, then **above program would not work**.

This technique, called **covariant return type**, means that the return type is allowed to vary in the same direction as the subclass.

Note: You also can use **interface** names as **return** types. In this case, the object returned must implement the specified interface.

Advantages:

- It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
- We get a liberty to have more specific return types when overriding methods.
- Help in preventing run-time ClassCastExceptions on returns

Using the this Keyword

Within an instance method or a constructor, **this** is a reference to the **current object** — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

The most common reason for using the `this` keyword is because a field is shadowed by a method or constructor parameter. For example, the `Point` class was written like this

```
public class Point {
    public int x = 0; public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a; y = b; }
}
```

But it could have been written like this:

```
public class Point {
    public int x = 0; public int y = 0;
    //constructor
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor `x` is a local copy of the constructor's first argument. To refer to the `Point` field `x`, the constructor must use `this.x`.

Using `this` with a Constructor

Within a constructor, you can also use the `this` keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*.

```
public class Rectangle {
    private int x, y; private int width, height;

    public Rectangle() { this(0, 0, 1, 1); }
    public Rectangle(int width, int height) { this(0, 0, width, height); }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x; this.y = y; this.width = width;
        this.height = height;
    }
    ...
}
```

If present, the invocation of another constructor must be the first line in the constructor.

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 Rectangle at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates.

Copy Constructor in Java

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

Following is an example Java program that shows a simple use of copy constructor.

// filename: Main.java

```

class Complex {

    private double re, im;
    // A normal parametrized constructor
    public Complex(double re, double im) {    this.re = re;    this.im = im;    }

    // Copy constructor
    Complex(Complex c) {
        System.out.println("Copy constructor called");    re = c.re;    im = c.im;
    }
    // Overriding the toString of Object class
    @Override
    public String toString() {    return "(" + re + " + " + im + "i";    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}

```

Output:
Copy constructor called
(10.0 + 15.0i)

Now try the following Java program:

// filename: Main.java

```

class Complex {

    private double re, im;
    public Complex(double re, double im) {    this.re = re;    this.im = im;    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(c1); // compiler error here
    }
}

```

As no copy constructor

Java destructor – Why is it missing?

Java lacks a destructor element, and instead uses a garbage collector for resource deallocation. The reason is that all Java objects are heap allocated and garbage collected. Java does support finalizers, but they are meant to be used only as a safeguard for objects holding a handle to native resources like sockets, file handles, window handles, etc.

The closest equivalent to a destructor in Java is the `Object.finalize()` method. It's usually not a good option.

When the garbage collector collects an object for garbage collection, its operation differs in following situations:

1. For an object without a finalizer.	• GC simply marks the memory region as free.
2. For an object without a finalizer.	• GC copies it into a temporary location • It is enqueued into a waiting-to-be-finalized queue • A Finalizer thread polls the queue with very low priority and runs the finalizer.

There is practically no guarantee that your finalizers will ever run due to following reasons :

- All Finalizers are run in the same thread, so if any of the finalizers hangs, the whole finalizer thread hangs.
- If an exception is thrown, this might kill the finalizer thread (although the JVM is free to respawn a new one) and in the worst case no further pending objects would be finalized.
- When the application exits, the JVM stops without waiting for the pending objects to be finalized.

So use of `finalize()` methods should be avoided. They are not a reliable mechanism for resource clean up and it is possible to cause problems in the garbage collector by abusing them.

If you require a deallocation call in your object, say to release resources, use an explicit method call. You should try to clean up in the logical places in your code using the `try{} finally{}` statements!

Destructor and Garbage Collector : Let's take a look on some definitions, to understand exactly what the destructor and the garbage collector do.

- **Destructor**: It is a special method called when the object's lifecycle is over, in order to free memory and deallocate resources. It is very prominent in languages with manual memory management (say C++) and extremely important to use it in order to avoid memory leaks.
- **Garbage Collector**: The garbage collector is a program that runs on the JVM and recovers memory by deleting objects that are not used anymore or are not accessible from the code (and are considered garbage, hence the name). It runs automatically and periodically checks the references

in contrast to the objects in the **memory heap**. If an unreferenced object is found the garbage collector gets rid of it and frees the memory.

The `Object.finalize()` method

- The **`finalize()`** method is inherited in all Java objects. This method is **NOT** a destructor!
- It will be called just before an object's final destruction by the garbage collector to ensure proper clean up.
- It is supposed to be used in order to provide **additional safety** in cases when you need to be sure that the use of external resources (like opening and closing a file, or a socket, or a stream) will be done correctly, which means closing everything before the program shutdown.
- To add a finalizer to a class, you simply define the **`finalize()`** and inside it specify those actions that must be performed before an object is destroyed.
- You can invoke the finalizers to run **either** by using the method itself (**`Object.finalize()`**) **or** by using the method **`System.runFinalizersOnExit(true)`**. The Java runtime calls that method whenever it is **about to recycle an object of that class**.
- **They also add overhead to the GC**, making your program **slower**! Due to the GC, it is not known when the finalizer will run, and this can potentially create problems.

Let's take a look on a safer way to use `finalize()`. Say we are opening a few streams, and use them, and in the end we call **`System.runFinalizersOnExit(true)`** in order to close all the remaining open streams by having implemented a **`finalize()`** method in this class. The signature of the `finalize` method is shown below :

```
// The finalize() method. We are using it to close all the open streams and NOT to destroy the object itself, as this is useless and dangerous. The GC will take care of that.
```

```
@Override
public void finalize() throws Throwable {
    .....
}
```

In the main method we call **`System.runFinalizersOnExit(true)`** at end to call the `finalize()`.

```
public static void main(String[] args) {
    .....
    .....
    // Here we are forcing the program to run all the implemented finalize() methods when the
    // program finishes. Again, not recommended!
    System.runFinalizersOnExit(true);
    System.out.println("Program exiting. The finalizers should run now.");
}
}
```

As seen, when the program is about to end, the finalizers start running and all the open streams are closed.