

Garbage Collection

In Java, memory occupied by an object is automatically reclaimed when the object is no longer needed. This is achieved by a process called **Garbage Collection**. The programmer does not have to worry about releasing or reclaiming memory used by object. This greatly reduces bugs and helps programmers be more productive.

The Java interpreter knows which objects it has allocated. It also knows which variables or objects refer to which other objects. So the interpreter can determine which objects are no longer referenced by any variable and it then destroys those objects.

The Java garbage collector runs as a low-priority thread, so it does most of its work when nothing else is going on. The only time that it must run even when some high-priority thread is going on is when available memory is dangerously low. But this doesn't happen often because the low-priority thread is running in the background and cleans unused objects.

Finalizers

Constructors are used to create objects: obtain memory, obtain resources, initialize object data...

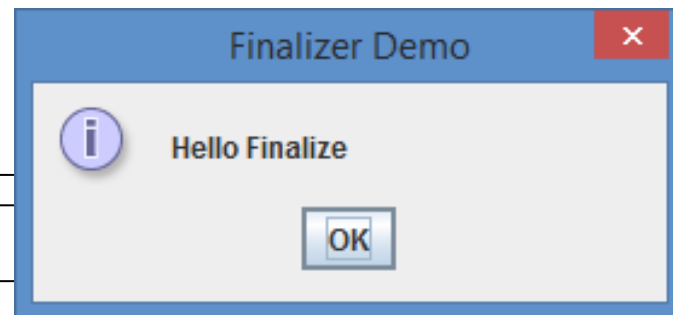
Finalizers are used to return allocated resources back to the system such as file, print and network connections. Remember that the garbage collector automatically reclaims memory. So you don't need to worry about reclaiming memory, except in some situations where you may have to help the garbage collector in identifying unused objects.

A class's finalizer is called just before the garbage collector destroys the object. It always has the name `finalize`, returns no values, has return type `void` and takes no parameters. If you don't define a `finalize` method for your class, a default one is created that does nothing.

Finalizers 2

Finalizers are a bit similar to C++'s destructor functions which are used to return resources to the system. Finalizers are not as useful and necessary as C++'s destructors and are not often used in normal Java programming. The following is an example demonstrating how finalizers are used:

```
import javax.swing.*;
public class EmployeeTest{
    public static void main(String[] args){
        String output="Hello Finalize";
        Employee e= new Employee("X", "YZ");
        JOptionPane.showMessageDialog(null,
            output,"Finalizer Demo",
            JOptionPane.INFORMATION_MESSAGE);
        e=null;        //mark for garbage collection
        System.gc();   //suggest that GC be called
        System.exit(0);
    }
}
```



Finalizers 3

```
class Employee{
    public Employee(String fName, String lName){
        this.fName=fName;
        this.lName=lName;
        System.out.println("Constuctor:" + fName+
            " " +lName);
    }
    protected void finalize(){
        System.out.println("Finalizer Called");
    }
    private String fName, lName; }

```

Finalizer methods are usually declared as `protected` so that subclasses can directly access and run them. You could also declare them as `Public` but information hiding and encapsulation may be compromised. The output of this program is a frame showing the message: "Hello Finalize"

Inheritance

Inheritance allows new classes to be created by reusing existing classes, thus saving time in software development. New classes acquire proven and debugged properties of existing classes.

In Java, the keyword `extends` is used to inherit a new class from an existing class:

```
class Child extends Parent {...}
```

The new class `Child` is the subclass and the `Parent` is the superclass.

Unlike C++, Java does not support multiple inheritance, but it supports interfaces which allow Java achieve many of the advantages of multiple inheritance without the associated problems.

Every object of the subclass is also an object of the super-class but not the other way round. A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package.

Inheritance

The **direct superclass** is the superclass from which the subclass explicitly inherits. An **indirect superclass** is any class above the direct superclass in the **class hierarchy**. In Java, the class hierarchy begins with class Object (in package java.lang), which *every* class in Java directly or indirectly **extends**.

Java supports only **single inheritance**, in which each class is derived from exactly *one* direct superclass. Unlike C++, Java does *not* support multiple inheritance (a class is derived from more than one direct superclass).

All public and protected superclass members retain their original access modifier when they become members of the subclass—public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass.

A superclass's private members are not accessible outside the class itself. Rather, they're *hidden* in its subclasses and can be accessed only through the public or protected methods inherited from the superclass.

Inheritance 2

Every class in Java must inherit from a superclass; if a new class does not explicitly extend another class, Java implicitly uses the `Object` class as the superclass for the new class. Class `Object` provides a set of methods that can be used with any object of any class.

Consider the following example which is taken from the textbook:

```
class Point {
    protected int x, y;
    public Point() {
        setPoint(0, 0);
    }
    public Point(int a, int b) {
        setPoint(a, b);
    }
    //see next page...
```

Inheritance 3

```
    public void setPoint(int a, int b) {
        x=a; y=b;
    }
    public int getX() { return x;}
    public int getY() { return y;}
    public String toString() {
        return "[" + x + ", " + y + "];"
    }
}
Public class Circle extends Point{
    protected double radius;
    public Circle() {
        setRadius(0);
    }
    //see next page...
```


Inheritance 4

```
public Circle(double r, int a, int b){
    super(a, b);
    setRadius(r);
}
public void setRadius(double r) {
    radius= (r >=0.0 ? r : 0.0);
}
public double getRadius() { return radius;}
public double area() {
    return Math.PI * radius * radius;
}
public String toString(){
    return "Center= " + "[" + x + ", " + y +
    "; Radius= " + radius; }
}
```

Inheritance 5


```
import java.text.DecimalFormat;
import javax.swing.JOptionPane;
Public class InheritanceTest {
    public static void main(String[] args) {
        Point pointRef, p;
        Circle circleRef, c;
        String output;
        p=new Point(30, 50);
        c=new Circle(2.7, 120, 89);
        output="Point p: " + p.toString() +
            "\nCircle c: " +c.toString();
        pointRef=c;    //since a circle is-a point
        output+="\nCircle c (via pointRef): " +
            pointRef.toString();
        circleRef=(Circle) pointRef; //downcast
```

Inheritance 6

```
output+="\nCircle c (via circleref): " +
        circleRef.toString();
DecimalFormat precision2=new DecimalFormat("0.00");
output+="\nArea of c (via circleRef): " +
        precision2.format(circleRef.area());
if(p instanceof Circle) {
    circleRef=(Circle) p;
    output+="\nCast Successful";
}
else
    output+="\np does not refer to a Circle";
JOptionPane.showMessageDialog(null, output,
    "Demonstarting the \"is-a \" relationship",
    JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
```

Inheritance 7

Demonstrating the "is-a " relationship ✕

 Point p: [30, 50]
Circle c: Center= [120, 89; Radius= 2.7]
Circle c (via pointRef): Center= [120, 89; Radius= 2.7]
Circle c (via circleref): Center= [120, 89; Radius= 2.7]
Area of c (via circleRef): 22.90
p does not refer to a Circle

Inheritance 7

In this example, class Circle inherits from class Point and adds members specific to itself:

- Circle overrides Point's `toString` method (polymorphism)
- Point and Circle both have a default constructor as well as a parameterized constructor
- Subclass Circle needs to call superclass Point's parameterized constructor using `super` along with any required arguments, and this statement must come before any other statements.
- Default constructors are invoked automatically.
- Superclass objects or references can be used to refer to subclass objects because of the is-a relationship, hence the statement `pointRef=c;`

Inheritance 8

- Explicit casting is needed to make a Circle object to refer to a Point object, hence the statement:

```
circleRef= (Circle) pointRef;
```

- Attempting to cast a Point object to a Circle object is an error, so the statement: `circleRef=(Circle) p;` is illegal because `p` refers to a Point object.
- The operator `instanceof` is used to check whether the object to which it refers is a Circle.
- Superclass constructors are not inherited; subclass constructors can call superclass constructors using the `super` reference.