



Classes and Objects in Java

Basics of Classes in Java



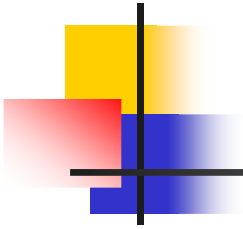
Contents

- Introduce to classes and objects in Java.
- Understand how some of the OO concepts learnt so far are supported in Java.
- Understand important features in Java classes.



Introduction

- Java is a true OO language and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the “state” and “behaviour” of the basic program components known as objects.

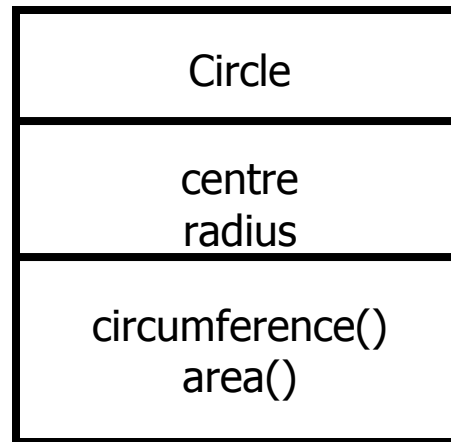


- Classes create objects and objects use methods to communicate between them. They provide a convenient method for packaging a group of logically related data items and functions that work on them.
- A class essentially serves as a template for an object and behaves like a basic data type "int". It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as encapsulation, inheritance, and polymorphism.



Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.





Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

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

- Bare bone class – no fields, no methods

```
public class Circle {  
    // my circle class  
}
```



Adding Fields: Class Circle with fields

- Add *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

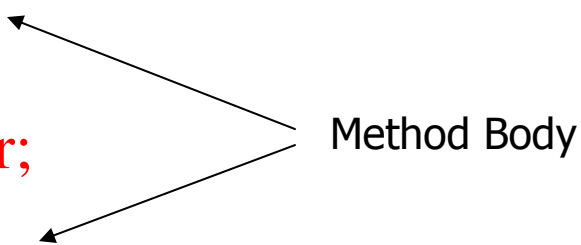
- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
    Method-body;
}
```




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 circumference() {  
        return 2*3.14*r;  
    }  
    public double area() {  
        return 3.14 * r * r;  
    }  
}
```



Method Body



Data Abstraction

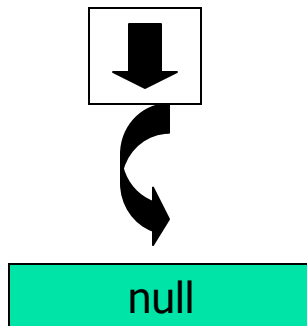
- Declare the Circle class, have created a new data type – **Data Abstraction**
- Can define variables (objects) of that type:

```
Circle aCircle;  
Circle bCircle;
```

Class of Circle cont.

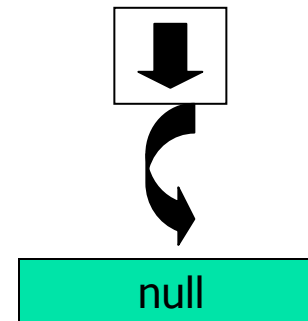
- aCircle, bCircle simply refers to a Circle object, not an object itself.

aCircle



Points to nothing (Null Reference)

bCircle

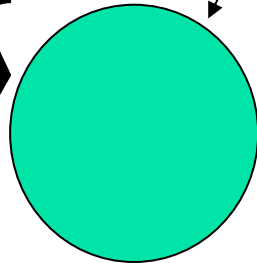


Points to nothing (Null Reference)

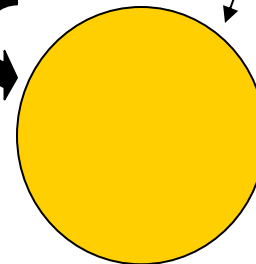
Creating objects of a class

- Objects are created dynamically using the *new* keyword.
- aCircle and bCircle refer to Circle

aCircle = new Circle() ;



bCircle = new Circle() ;





Creating objects of a class

```
aCircle = new Circle();  
bCircle = new Circle() ;
```

```
bCircle = aCircle;
```

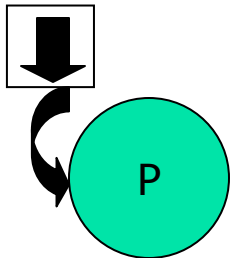
Creating objects of a class

```
aCircle = new Circle();  
bCircle = new Circle();
```

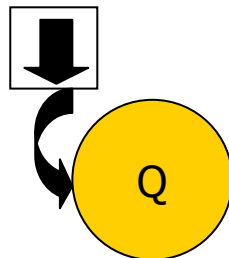
```
bCircle = aCircle;
```

Before Assignment

aCircle

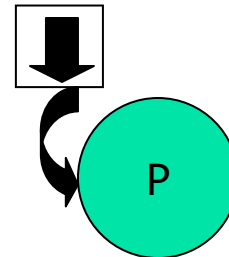


bCircle

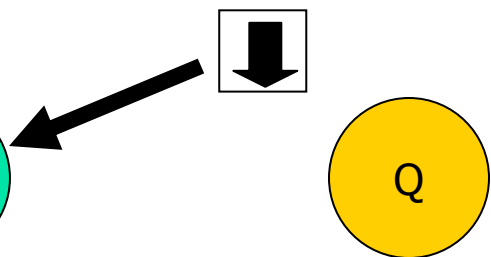


Before Assignment

aCircle





bCircle





Automatic garbage collection

- 
- The object  does not have a reference and cannot be used in future.
 - The object becomes a candidate for automatic **garbage collection**.
 - Java automatically collects garbage periodically and releases the memory used to be used in the future.



Accessing Object/Circle Data

- Similar to C syntax for accessing data defined in a structure.

ObjectName.VariableName

ObjectName.MethodName(parameter-list)

```
Circle aCircle = new Circle();
```

```
aCircle.x = 2.0 // initialize center and radius
```

```
aCircle.y = 2.0
```

```
aCircle.r = 1.0
```


Executing Methods in Object/Circle

- Using Object Methods:

sent 'message' to aCircle

```
Circle aCircle = new Circle();  
  
double area;  
aCircle.r = 1.0;  
area = aCircle.area();
```



Using Circle Class

```
// Circle.java: Contains both Circle class and its user class
//Add Circle class code here
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.circumference();
        System.out.println("Radius="+aCircle.r+" Area="+area);
        System.out.println("Radius="+aCircle.r+" Circumference =" +circumf);
    }
}
```

```
[raj@mundroo]~: java MyMain
Radius=5.0 Area=78.5
Radius=5.0 Circumference =31.400000000000002
```



What is a Constructor?

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).



- Like any other method

Defining a Constructor

```
public class ClassName {  
  
    // Data Fields...  
  
    // Constructor  
    public ClassName()  
    {  
        // Method Body Statements initialising Data Fields  
    }  
  
    //Methods to manipulate data fields  
}
```

- Invoking:

- There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.



Defining a Constructor: Example

```
public class Counter {  
    int CounterIndex;  
  
    // Constructor  
    public Counter()  
    {  
        CounterIndex = 0;  
    }  
    //Methods to update or access counter  
    public void increase()  
    {  
        CounterIndex = CounterIndex + 1;  
    }  
    public void decrease()  
    {  
        CounterIndex = CounterIndex - 1;  
    }  
    int getCounterIndex()  
    {  
        return CounterIndex;  
    }  
}
```



Multiple Constructors

- Sometimes want to initialize in a number of different ways, depending on circumstance.
- This can be supported by having multiple constructors having different input arguments.



Multiple Constructors

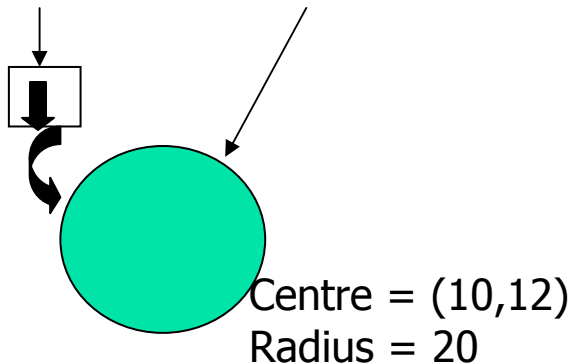
```
public class Circle {  
    public double x,y,r; //instance variables  
    // Constructors  
    public Circle(double centreX, double centreY, double radius) {  
        x = centreX; y = centreY; r = radius;  
    }  
    public Circle(double radius) { x=0; y=0; r = radius; }  
    public Circle() { x=0; y=0; r=1.0; }  
  
    //Methods to return circumference and area  
    public double circumference() { return 2*3.14*r; }  
    public double area() { return 3.14 * r * r; }  
}
```

Initializing with constructors

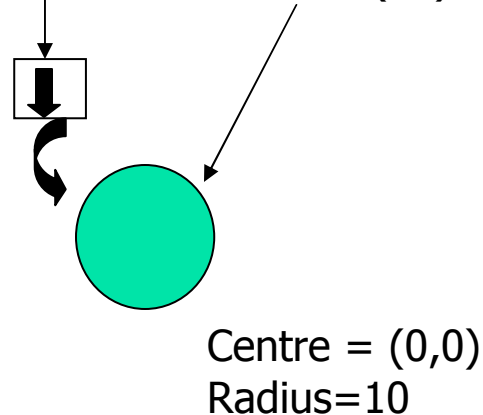
```
public class TestCircles {
```

```
    public static void main(String args[]){  
        Circle circleA = new Circle( 10.0, 12.0, 20.0);  
        Circle circleB = new Circle(10.0);  
        Circle circleC = new Circle();  
    }  
}
```

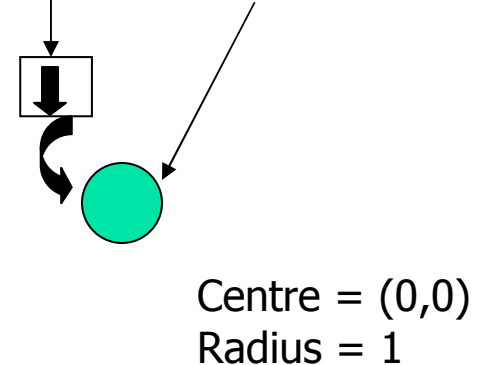
circleA = new Circle(10, 12, 20)



circleB = new Circle(10)



circleC = new Circle()





Method Overloading

- Constructors all have the same name.
- Methods are distinguished by their signature:
 - name
 - number of arguments
 - type of arguments
 - position of arguments
- That means, a class can also have multiple usual methods with the same name.
- Not to confuse with *method overriding* (coming up), *method overloading*:



Polymorphism

- Allows a single *method or operator* associated with different meaning depending on the type of data passed to it. It can be realised through:
 - Method Overloading
 - Operator Overloading (Supported in C++, but not in Java)
- Defining the same *method* with different argument types (method overloading) - polymorphism.
- The method body can have different logic depending on the data type of arguments.

A Program with Method

Overloading

```
// Compare.java: a class comparing different items
class Compare {
    static int max(int a, int b)
    {
        if( a > b)
            return a;
        else
            return b;
    }
    static String max(String a, String b)
    {
        if( a.compareTo (b) > 0)
            return a;
        else
            return b;
    }

    public static void main(String args[])
    {
        String s1 = "Melbourne";
        String s2 = "Sydney";
        String s3 = "Adelaide";

        int a = 10;
        int b = 20;

        System.out.println(max(a, b)); // which number is big
        System.out.println(max(s1, s2)); // which city is big
        System.out.println(max(s1, s3)); // which city is big
    }
}
```



The New *this* keyword

- *this* keyword can be used to refer to the object itself.

It is generally used for accessing class members (from its own methods) when they have the same name as those passed as arguments.

```
public class Circle {  
    public double x,y,r;  
    // Constructor  
    public Circle (double x, double y, double r) {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
}
```

```
//Methods to return circumference and area
```

```
}
```



Static Members

- Java supports definition of global methods and variables that can be accessed without creating objects of a class. Such members are called Static members.
- Define a variable by marking with the **static** methods.
- This feature is useful when we want to create a variable common to all instances of a class.
- One of the most common example is to have a variable that could keep a count of how many objects of a class have been created.
- Note: Java creates only one copy for a static variable which can be used even if the class is never instantiated.

Static Variables

- Define using *static*:

```
public class Circle {  
    // class variable, one for the Circle class, how many circles  
    public static int numCircles;  
  
    //instance variables,one for each instance of a Circle  
    public double x,y,r;  
    // Constructors...  
}
```

- Access with the class name (ClassName.StatVarName):

```
nCircles = Circle.numCircles;
```

Static Variables - Example

- Using *static variables*:

```
public class Circle {  
    // class variable, one for the Circle class, how many circles  
    private static int numCircles = 0;  
    private double x,y,r;  
  
    // Constructors...  
    Circle (double x, double y, double r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
        numCircles++;  
    }  
}
```

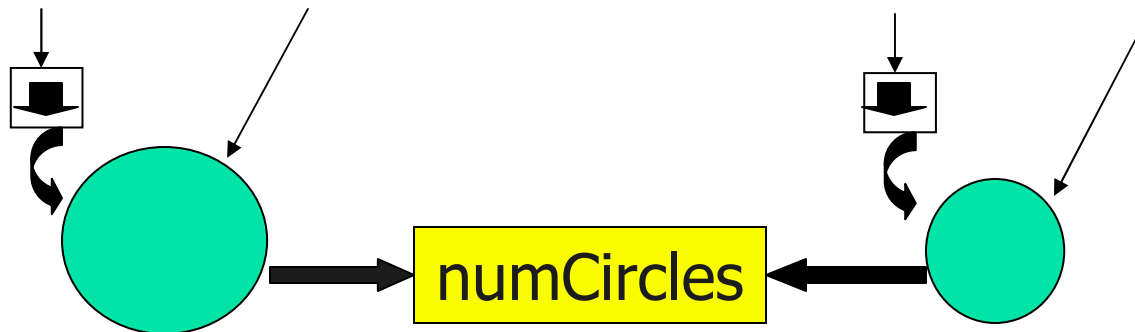
Class Variables - Example

- Using *static variables*:

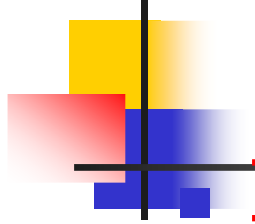
```
public class CountCircles {  
  
    public static void main(String args[]){  
        Circle circleA = new Circle( 10, 12, 20);    // numCircles = 1  
        Circle circleB = new Circle( 5, 3, 10);      // numCircles = 2  
    }  
}
```

circleA = new Circle(10, 12, 20)

circleB = new Circle(5, 3, 10)



Instance Vs Static Variables



Instance variables : One copy per **object**.
Every object has its own instance variable.

- E.g. x, y, r (centre and radius in the circle)

- **Static** variables : One copy per **class**.

- E.g. numCircles (total number of circle objects created)



Static Methods

- A class can have methods that are defined as static (e.g., main method).
- Static methods can be accessed without using objects. Also, there is NO need to create objects.
- They are prefixed with keyword “static”
- Static methods are generally used to group related library functions that don't depend on data members of its class. For example, Math library functions.

Comparator class with Static methods

// Comparator.java: A class with static data items comparison methods

```
class Comparator {
```

```
    public static int max(int a, int b)
```

```
    {
```

```
        if( a > b)
```

```
            return a;
```

```
        else
```

```
            return b;
```

```
    }
```

Directly accessed using ClassName (NO Objects)

```
    public static String max(String a, String b)
```

```
    {
```

```
        if( a.compareTo (b) > 0)
```

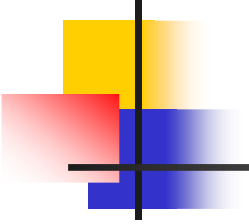
```
            return a;
```

```
        else
```

```
            return b;
```

```
    }
```

```
}
```



```
class MyClass {  
    public static void main(String args[])  
    {  
        String s1 = "Melbourne";  
        String s2 = "Sydney";  
        String s3 = "Adelaide";  
  
        int a = 10;  
        int b = 20;  
  
        System.out.println(Comparator.max(a, b)); // which number is big  
        System.out.println(Comparator.max(s1, s2)); // which city is big  
        System.out.println(Comparator.max(s1, s3)); // which city is big  
    }  
}
```



Static methods restrictions

- They can only call other static methods.
- They can only access static data.
- They cannot refer to “this” or “super” (more later) in anyway.



Visibility Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods, *encapsulation*, safely sealing data within the *capsule* of the class
- Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)



Visibility Modifiers: Public, Private, Protected

- *Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
 - Default (No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Visibility

```
public class Circle {  
    private double x,y,r;
```

```
    // Constructor
```

```
    public Circle (double x, double y, double r) {  
        this.x = x;  
        this.y = y;  
        this.r = r;
```

```
    }
```

```
    //Methods to return circumference and area
```

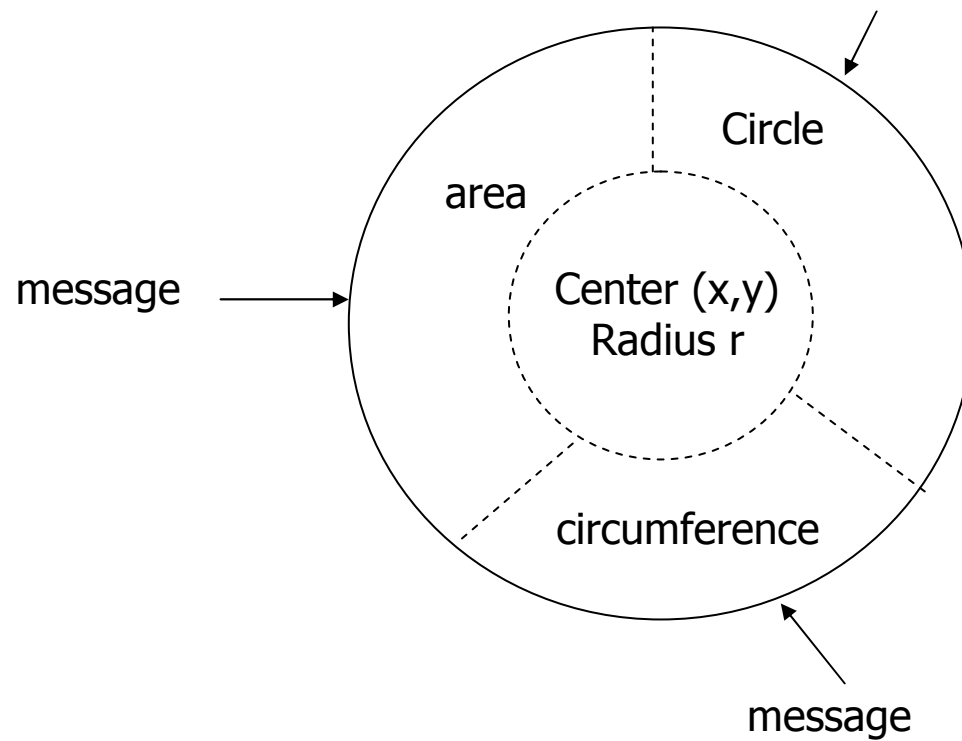
```
    public double circumference() { return 2*3.14*r;}  
    public double area() { return 3.14 * r * r; }
```

```
}
```


Visibility

Circle

Construction time message



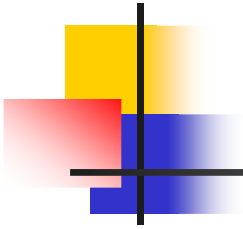
Accessors – “Getters/Setters”

```
public class Circle {  
    private double x,y,r;  
  
    //Methods to return circumference and area  
    public double getX() { return x;}  
    public double getY() { return y;}  
    public double getR() { return r;}  
    public double setX(double x) { this.x = x;}  
    public double setY(double y) { this.y = y;}  
    public double setR(double r) { this.r = r;}  
}
```



Objects Cleanup/Destructor

- Unlike c and c++, memory deallocation is automatic in java, don't worry about it
 - no dangling pointers and no memory leak problem.
- Java allows you to define *finalizer* method, which is *invoked* (if defined) just before the object destruction.
- In way, this presents an opportunity to perform record-maintenance operation or cleanup any special allocations made by the user.



```
//      done with this circle
protected void finalize() throws IOException {
    Circle.numCircles = Circle.numCircles--;
    System.out.println("number of circles:" + Circle.num_circles);
}
```