Class , Object & Casting

Class

- Class - A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support
- Class defines structure and behavior (data & code) that will be shared by a set of objects

Example

```
  class MyClass  {    }
```

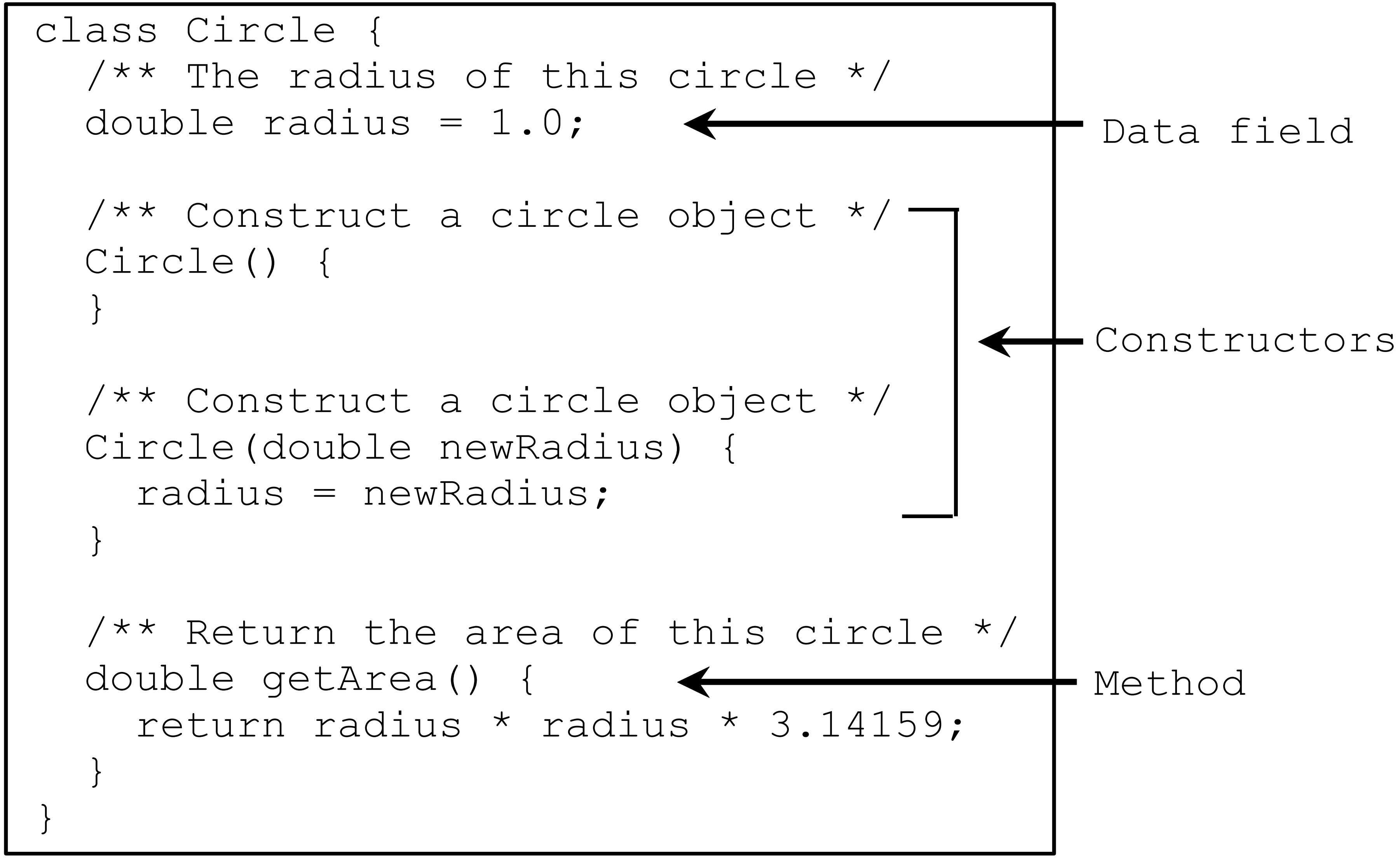
Object

- An object is a region of storage that defines both state & behavior.
  - **State** is represented by a set of variables & the values they contain.
  - **Behavior** is represented by a set of methods & the logic they implement.
- Thus, an object is a combination of a data & the code that acts upon it.
- Objects are the basic runtime entities in an object-oriented system.
- Objects are instance of a class.

Example:

```
person p1,p2;
p1 = new person();
p2 = new person();
```

Classes



```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Constructors.

- A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created.
  - A constructor with no parameters is referred to as a *no-arg constructor*.
  - Constructors must have the same name as the class itself.
  - Constructors do not have a return type—not even void.
  - Constructors are invoked using the new operator when an object is created.
  - Constructors play the role of initializing objects.

Example:

```
class A{
private int Age;
public A(String name){
    System.out.println("Passed Name is: "+ name );
}
public void setAge(int age ){
Age = age;
}
public int getAge(){
return Age;
}
}
```

```
public class JavaApplication3 {
   public static void main(String[] args) {
     /* Object creation */
      A a1 =new A("Rahim");
      a1.setAge(25);
      System.out.println("Age is: "+a1.getAge() );
   }
}
```

```
Output:
Passed Name is: Rahim
Age is: 25
```

Method Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

```
int i = 5
double d = 5.0
```

Why overload a method?

- So you can use the same names for methods that do essentially the same thing
    - Example: println(int), println(double), println(boolean), println(String), etc.
- So you can supply defaults for the parameters:

```
int increment(int amount) {
    count = count + amount;
    return count;
}
int increment() {
    return increment(1);
}
```

- Notice that one method can call another of the same name

- So you can supply additional information:

```
void printResults() {
    System.out.println("total = " + total + ", average = " + average);
}
void printResult(String message) {
    System.out.println(message + ": ");
    printResults();
}
```

Constructor Overloading

- Constructors are methods that can be overloaded, just like any other method in a class.
- In most situations, you will want to generate objects of a class from different sets of initial defining data
- The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
public class MyClass
{
    int x;
    MyClass()
    {
       System.out.println("Inside MyClass() constructor.");
        x=0;
    }
    MyClass(int i)
    {
  System.out.println("Inside MyClass(int) constructor.");
        x=i;
    }
     MyClass(double d)
    {
System.out.println("Inside MyClass(double) constructor.");
        x=(int)d;
    }
```

```
void getXvalue()
    {
        System.out.println("The value of the instance
             variable of the object  is “ +x +".");
    }    }

public class MyClassTest
{

    public static void main(String[] args)
    {
        MyClass first=new MyClass();
        MyClass second=new MyClass(52);
        MyClass third=new MyClass(13.6);
        first.getXvalue();
        second.getXvalue();
        third.getXvalue();
    }
}
```

```
Inside MyClass() constructor.
Inside MyClass(int) constructor.
Inside MyClass(double) constructor.
The value of the instance variable of the object  is 0 .
The value of the instance variable of the object  is 52.
The value of the instance variable of the object  is 13.
```

```
public class MyClass
{
   int x, y;
   MyClass(){
  System.out.println("Inside MyClass() constructor.");
       x=0;
       y=0;
    }
  MyClass(int i, int j)  {
  System.out.println("Inside MyClass(int) constructor.");
      x=i;
      y=j;    }

    MyClass(MyClass obj)
    {
 System.out.println("Inside MyClass(MyClass) constructor.");
       x=obj.x;
       y=obj.y;
    }
```

```
void getXYvalues()
    {
System.out.println("The value of the instance variables of the object are "+x+" and "+y+".");
    }
}

public class MyClassTest
{
    public static void main(String[] args)
    {
        MyClass first=new MyClass();
        MyClass second=new MyClass(52, 18);
        MyClass third=new MyClass(second);
        first.getXYvalues();
        second.getXYvalues();
        third.getXYvalues();
    }
}
```

```
Inside MyClass() constructor.
Inside MyClass(int) constructor.
Inside MyClass(MyClass) constructor.
The value of the instance variable of the object  is 0 and 0.
The value of the instance variable of the object  is 52 and 18.
The value of the instance variable of the object  is 52 and 18.
```

The this Keyword

- The <u>this</u> keyword refers to the current object in a method or constructor.
- The <u>this</u> keyword is the name of a reference that refers to an object itself.
- The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name

```
public class MyClass {
  int x;

  // Constructor with a parameter
  public MyClass(int x) {
    this.x = x;
  }

  // Call the constructor
  public static void main(String[] args) {
    MyClass myObj = new MyClass(5);
    System.out.println("Value of x = " + myObj.x);
  }
}
```

If you omit the keyword in the example above, the output would be "0" instead of "5"

Reference the Hidden Data Fields

```java
public class Foo {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Garbage Collection

- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management
- 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.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts

Array Of Object

```
public class Simple {

public int member;

public void foo() {
        System.out.println("foo");
}

public static void main(String args[]) {
        Simple[] samp = new Simple[10];

        for (int i = 0; i < 10; i++) {
        samp[i] = new Simple();
        samp[i].foo();
}   }   }
```

Array Of Object Example:

```
class ArrayOfObject{
                 int id;
                 String name;
                 double marks;
  void getInput() {
         Scanner in = new Scanner(System.in);
         System.out.println("Enter Student Name");
         name = in.nextLine();
         System.out.println("Enter Student id");
         id = in.nextInt();
         System.out.println("Enter Student Marks");
         marks = in.nextDouble();}
 void Show() {
       System.out.println("Id is :" + id);
       System.out.println("Name is :" + name);
       System.out.println("marks is : " + marks);
  }}
```

```
public class JavaApplication3 {
    public static void main(String[] args) {
      ArrayOfObject samp[] = new ArrayOfObject[3];
      for (int i = 0; i < samp.length; i++) {
                  samp[i] = new ArrayOfObject();
                  samp[i].getInput();     }
        for (int i = 0; i < samp.length; i++) {
                  samp[i].Show();
    }
}
}
```

Type Casting

- The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.
- **Widening Type Casting:** In **Widening Type Casting**, Java automatically converts one data type to another data type:

```
int num = 10;
 System.out.println("The integer value: " + num);

 // convert into double type
 double data = num;
 System.out.println("The double value: " + data);
```

- In the case of Widening Type Casting, the lower data type (having smaller size) is converted into the higher data type (having larger size). Hence there is no loss in data. This is why this type of conversion happens automatically.

Narrowing Type Casting

- In Narrowing Type Casting, we manually convert one data type into another using the parenthesis.

```
double num = 10.99;
  System.out.println("The double value: " + num);

  // convert into int type
  int data = (int)num;
  System.out.println("The integer value: " + data);
```

- In the case of Narrowing Type Casting, the higher data types (having larger size) are converted into lower data types (having smaller size). Hence there is the loss of data