# Object Oriented Programming with Java

Zheng Chen

## Class and Object

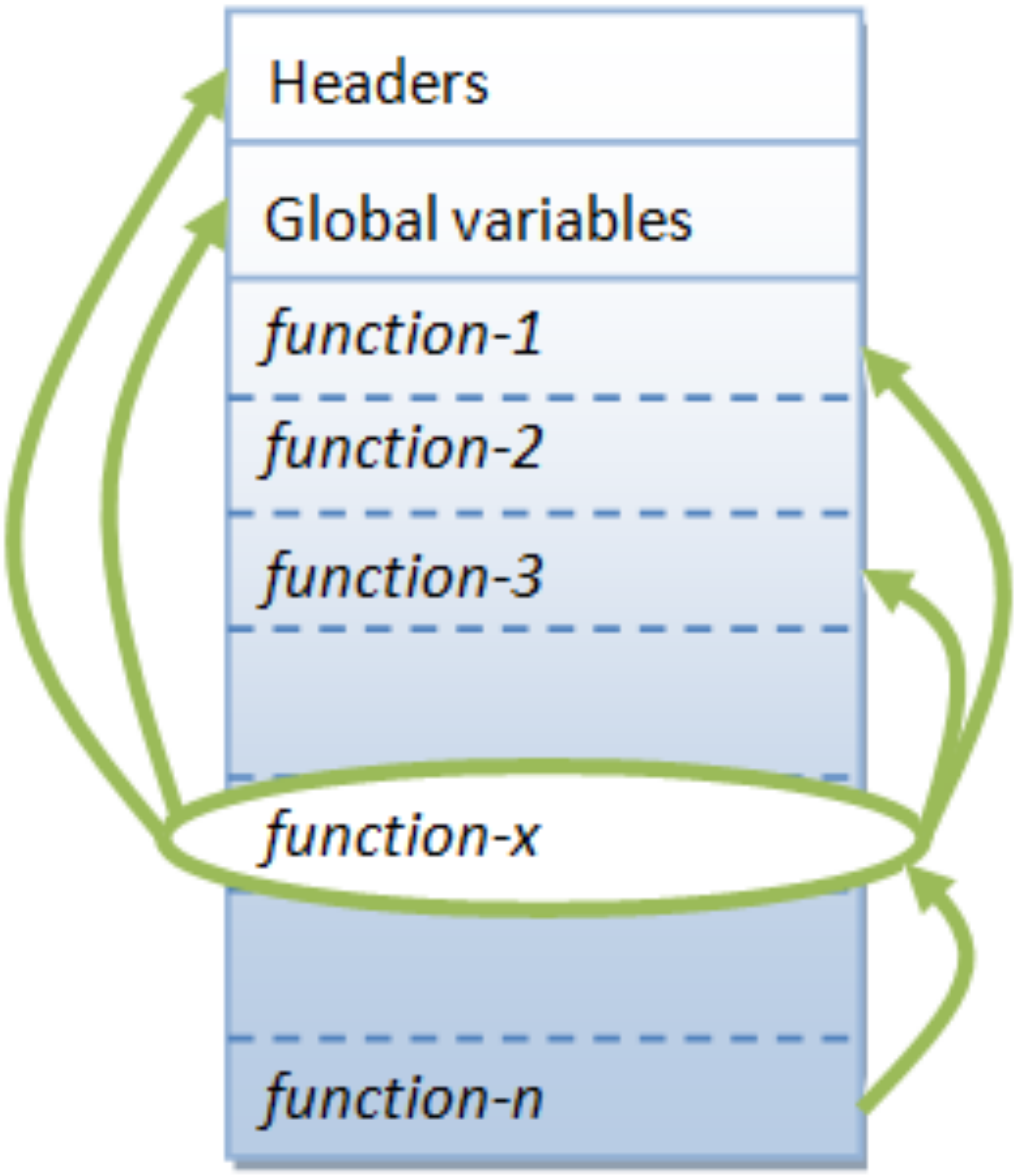
1. Object Oriented Programming
2. Class
3. Constructor
4. Method Overloading
5. Instance and Class Members
6. Object Life Cycle

## Why Object Oriented Programming?

- Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together.
- Can you "assemble" a software application?  The answer is obviously NO!
- Why re-invent the wheels? Why re-writing codes? Can you write better codes than those codes written by the experts?

## Traditional Procedural-Oriented languages

- Functions are less reusable, because it is likely to reference the global variables and other functions.
- The procedural languages are not suitable of high-level abstraction for solving real life problems.
- The traditional procedural-languages separate the data structures (variables) and algorithms (functions).



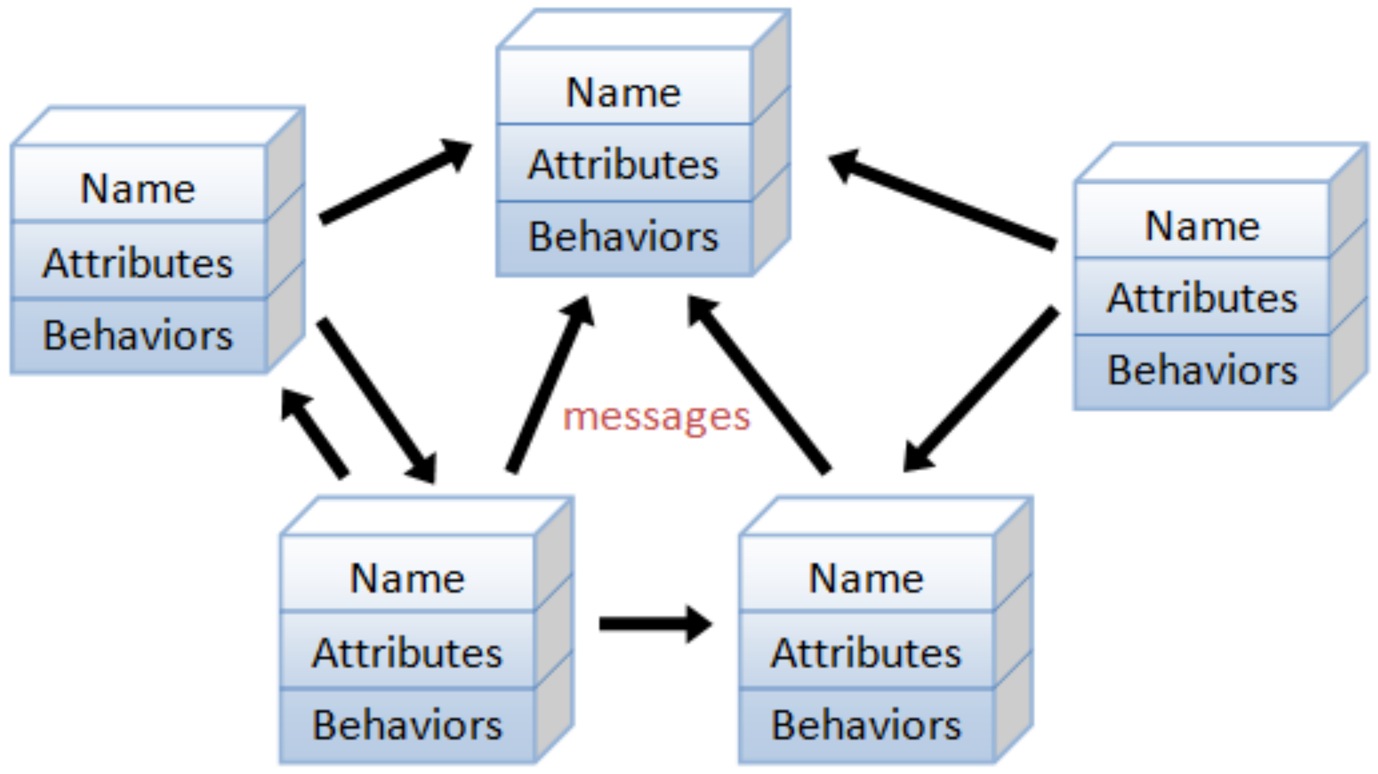
A function (in C) is not well-encapsulated

# Object-Oriented Programming Languages

- The basic unit of OOP is class, which **encapsulates** both the static properties and dynamic operations within a "box", and specifies the public interface for using these boxes.

- Since classes are well-encapsulated, it is easier to reuse these classes.

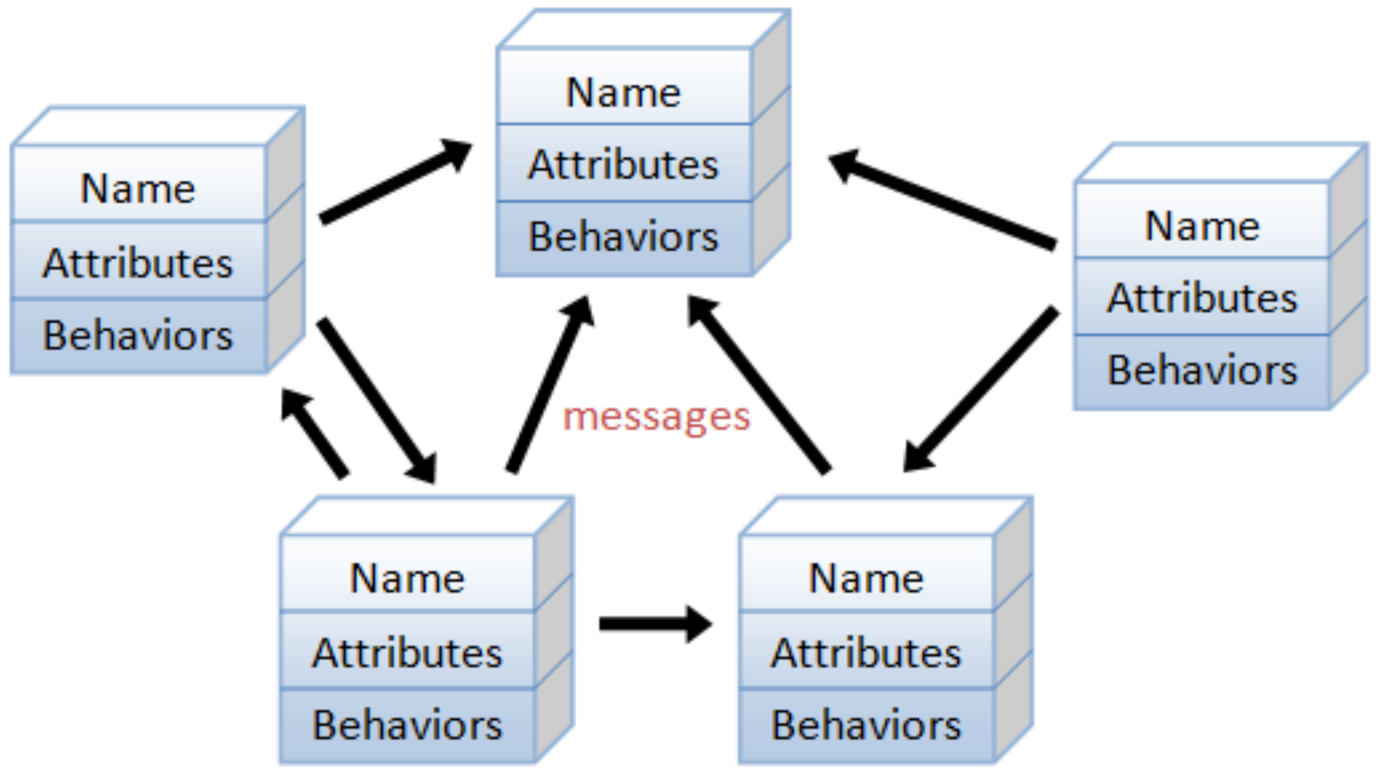
- In other words, OOP combines the data structures and algorithms of a software entity inside the same box.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

## Object-Oriented Programming Languages

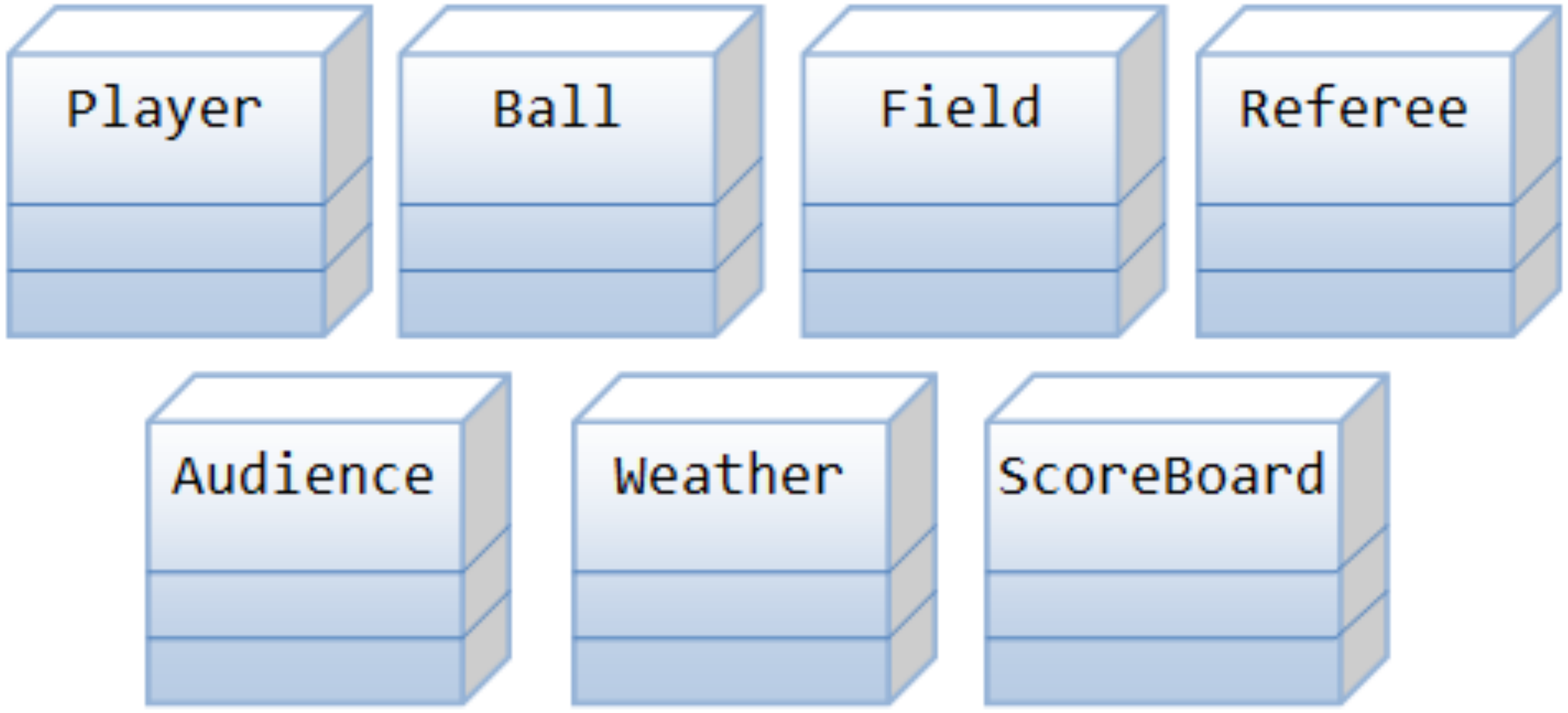
- OOP languages permit higher level of abstraction for solving real-life problems.
- The OOP languages let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

# A computer soccer games

- Player:
  - attributes include name, number, location in the field, and etc;
  - operations include run, jump, kick-the-ball, and etc.
- Ball:
- Referee:
- Field:
- Audience:
- Weather:



Classes (Entities) in a Computer Soccer Game

## OOP in Java

- In Java, a **class** is a definition of objects of the same kind.
    - In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.

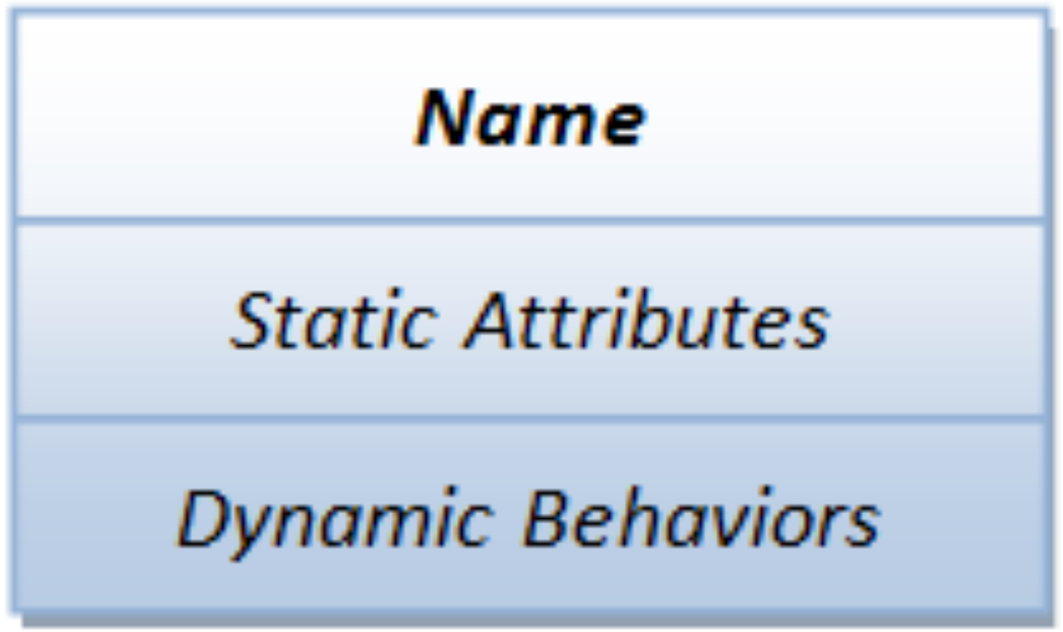
- An **instance** is a realization of a particular item of a class.
    - In other words, an instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition.

## Class and Object

1. Object Oriented Programming
2. **Class**
3. Constructor
4. Method Overloading
5. Instance and Class Members
6. Object Life Cycle

## Class

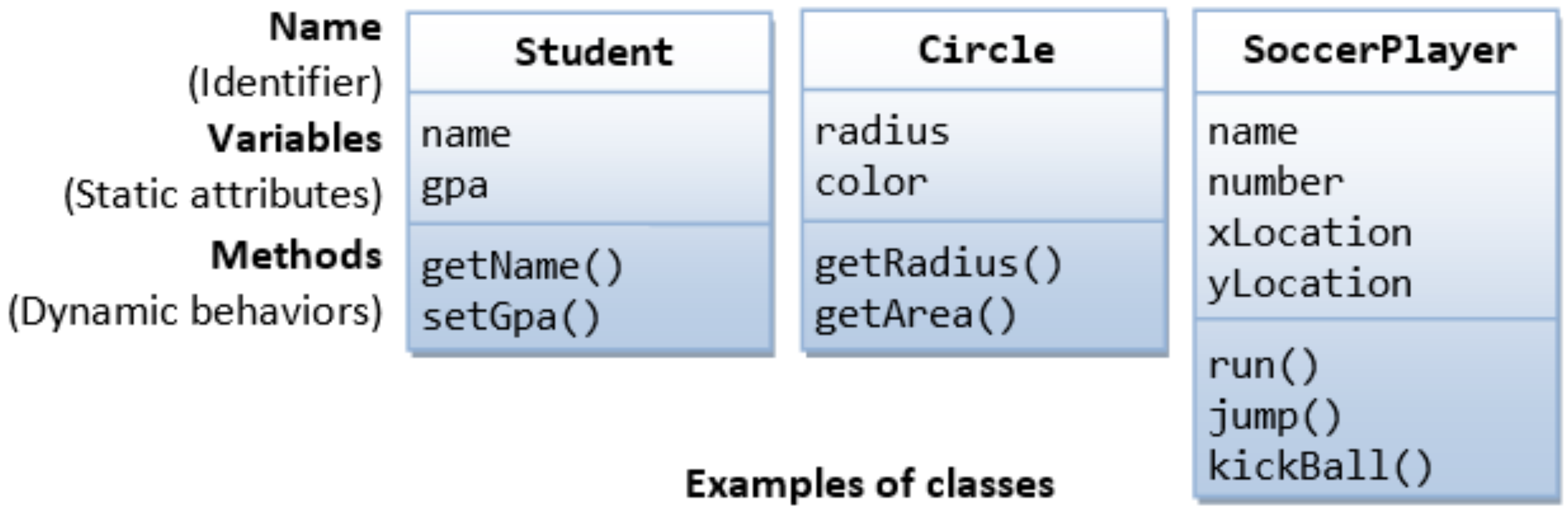
- A class can be visualized as a three-compartment box, as illustrated:
  - Name (or identity): identifies the class.
  - Variables (or attribute, state, field): contains the static attributes of the class.
  - Methods (or behaviors, function, operation): contains the dynamic behaviors of the class.



A class is a 3-compartment box

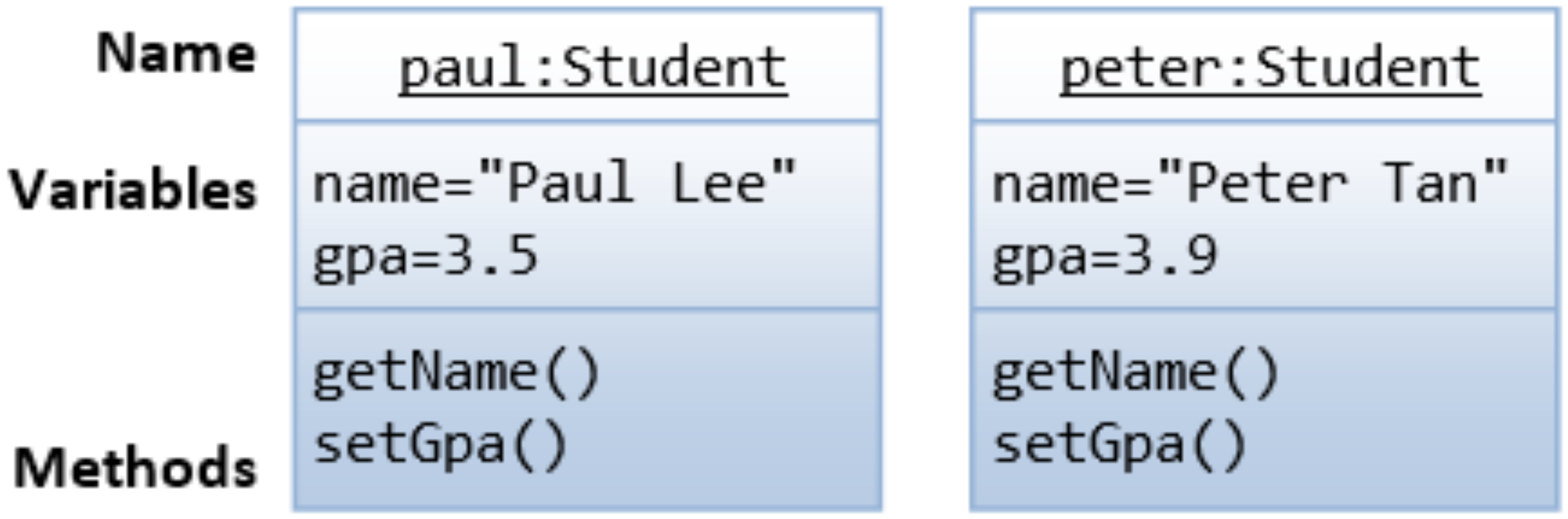
- In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

## Class Example



Examples of classes

## Instance Example



**Two instances - paul and peter - of the class Student**

## Class Conception Summarization

- A class is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.

- A class is a 3-compartment box containing the name, variables and the methods.

- A class encapsulates the data structures (in variables) and algorithms (in methods). The values of the variables constitute its state. The methods constitute its behaviors.

- An instance is an instantiation (or realization) of a particular item of a class.

## Class Definition in Java

```
[AccessControlModifier] class ClassName { // class name
   dataType varName1;              // variables
   dataType varName2;

   dataType methodName1() { ...... }  // methods
   dataType methodName1() { ...... }
}
```

## Class Definition Example

```
public class Circle {        // class name
   double radius;              // variables
   String color;

   double getRadius() { ...... }  // methods
   double getArea() { ...... }
}
```

## Creating Instances of a Class

- To create an instance of a class, you have to:

  1. **Declare** an instance identifier (instance name) of a particular class.

  2. **Construct** the instance (i.e., allocate storage for the instance and initialize the instance) using the "new" operator.

  3. **Assign** the instance to the instance identifier.

## Creating Instances Example

```
// Declare 3 instances of the class Circle, c1, c2, and c3
Circle c1, c2, c3;  // They hold a special value called null
c1 = new Circle(); // Instantiation via new operator
c2 = new Circle(2.0);
c3 = new Circle(3.0, "red");

// You can Declare and Construct in the same statement
Circle c4 = new Circle();
```

When an instance is declared but not constructed,
it holds a special value called null.

## Dot (.) Operator

- The variables and methods belonging to a class are formally called member variables and member methods. To reference a member variable or method, you must:

  1. First identify the instance you are interested in, and then,

  2. Use the dot operator (.) to reference the desired member variable or method.

## Dot (.) Operator Example

```
// Declare and construct c1 and c2 of the class Circle
Circle c1 = new Circle ();
Circle c2 = new Circle ();
// Invoke member methods for c1 via dot operator
System.out.println(c1.getArea());
System.out.println(c1.getRadius());
// Reference member variables for c2 via dot operator
c2.radius = 5.0;
c2.color = "blue";
```

## Member Variables

- A member variable has a name (or identifier) and a type; and holds a value of that particular type (as descried in the earlier chapter).

  **[AccessControlModifier] type varName**
  **[= initialValue] [, type varName2 [= initialValue2]] ... ;**

- Example

  ```
  private double radius;
  public int length = 1, width = 1;
  ```

## Member Methods

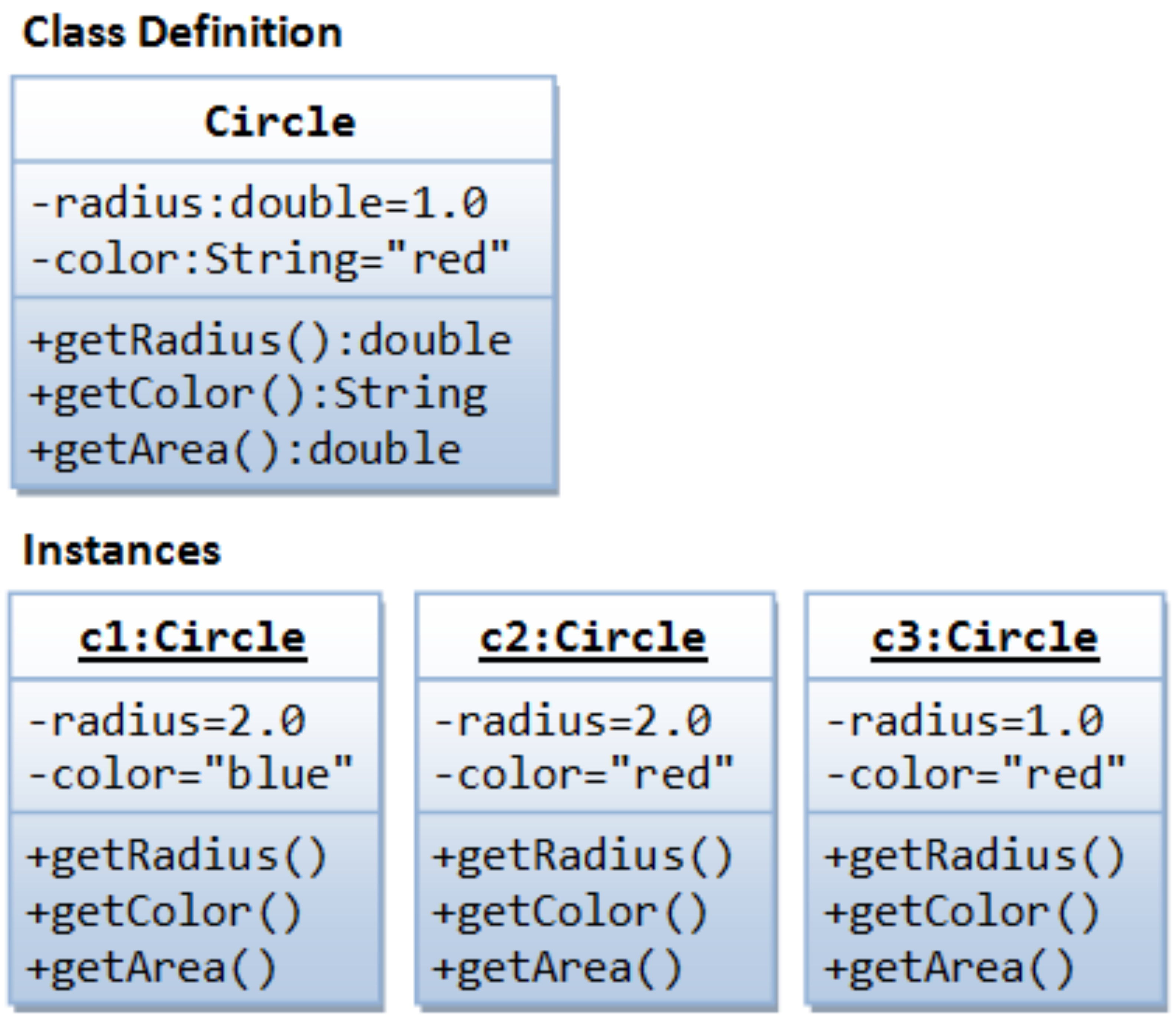
- A method receives arguments from the caller, performs the operations defined in the method body, and returns a piece of result (or void) to the caller.

  **[AccessControlModifier] returnType methodName ([parameterList]) { … }**

- Example

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

## An OOP Example



```
public class Circle {
   // Private instance variables
   private double radius;
   private String color;

   // Public methods
   public double getRadius() {
      return radius;
   }
   public String getColor() {
      return color;
   }
   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```

Circle.java

```
public class TestCircle {
   public static void main(String[] args) {
      // Declare and Construct an instance of the Circle class
      Circle c1 = new Circle();
      c1.radius=2.0;     //illegal access
      c1.color=“blue"   //and break encapsulation
      System.out.println("The radius is: " + c1.getRadius());
      // use dot operator to invoke member methods
      System.out.println("The color is: " + c1.getColor());
      System.out.printf("The area is: %.2f%n", c1.getArea());
      ……
     }
}
```

TestCircle.java

## Encapsulation

- Encapsulation is all about wrapping variables and methods in one single unit.

- Encapsulation is also known as data hiding. Because, when you design your class you may (and you should) make your variables hidden from other classes and provide methods to manipulate the data instead.

- Your class should be designed as a **black-box**. In other words – you give the class some data and get new data as response, without caring about the internal mechanisms used for data processing.

## Encapsulation - 2

- To achieve encapsulation:
  - Declare the variables of a class as **private**.
  - Provide public **setter** and **getter** methods to modify and view the variables values.
- The benefits of encapsulation:
  - The fields of a class can be made read-only or write-only.
  - A class can have **total control** over what is stored in its fields.
  - The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

## Getters and Setters

- To allow other classes to read the value of a private variable says xxx, we provide a get method (or getter or accessor method) called getXxx().
    - A get method needs not expose the data in raw format.
    - It can process the data and limit the view of the data others will see.
    - The getters shall not modify the variable.
- To allow other classes to modify the value of a private variable says xxx, we provide a set method (or setter or mutator method) called setXxx().
    - A set method could provide data validation (such as range checking), or transform the raw data into the internal representation.

```
public class Circle {

    private double radius;//radius in cm
    private String color;

    public double getRadius() {
        return radius;
    }

    public double getDiameter() {
        return radius * 2;
    }

    public double getRadiusInMeter() {
        return radius / 100;
    }
    …
}
```

```
public class Circle {

    private double radius;//radius in cm
    private String color;

    public void setRadius(double radius) {
        if (radius > 0) {//check validation
            this.radius = radius;
        }
    }
    …
}
```

## Class and Object

1. Object Oriented Programming
2. Class
3. **Constructor**
4. Method Overloading
5. Instance and Class Members
6. Object Life Cycle

## Constructors

- A constructor looks like **a special method** that has the same method name as the class name.
- A constructor is used to **construct** and **initialize** all the member variables.
- Constructor has no return type.

## Constructor Syntax

```
class ClassName {
  [AccessControlModifier] ClassName([paras]){
    …
  }
}
```

## No argument Constructors

- A constructor with no parameter is called the **default** constructor. It initializes the member variables to their default value.

```
Public class MyClass {
   Int num;
   MyClass() {
      num = 100;
   }
}
```

## Parameterized Constructors

- It is used to provide different values to distinct objects.

```
class MyClass {
   int x;
   MyClass( int i ) {
      x = i;
   }
}
```

## Call Constructors

- To construct a new instance of a class, you need to use a special "**new**" operator followed by a call to one of the constructors.
- Example

```
Circle c1 = new Circle();
Circle c2 = new Circle(2.0);
Circle c3 = new Circle(3.0, "red");
```

## Some important rules

- Constructor name must be **same** as the class name.
- It must not have any return value explicitly.
- A java constructor cannot be abstract, final and synchronized.
- Every class has the constructor whether it’s a regular class or an abstract class.
- If you don’t implement any constructor within the class, the compiler will do it for you.
- Constructor overloading is possible but overriding is not possible. Which means we can have the overloaded constructor in our class, but we can’t override a constructor.

## Some important rules - 2

- If Superclass doesn’t have a no-arg(default) constructor, then the compiler would not insert a default constructor in a child class as it does in a standard scenario.

- Interfaces do not have constructors.

- A constructor can also invoke another constructor of the same class. By using this(). If you want to invoke the parameterized constructor, then do it like the following: this(parameter list).

- Constructors cannot be inherited.

## Use this() to invoke a constructor

```
public class Circle {

    private double radius;//radius in cm
    private String color;

    public Circle() {
        radius = 1;
        color = "red";
    }

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

}
```

## Use this() to invoke a parameterized constructor

```
public class Circle {

    private double radius;//radius in cm
    private String color;

    public Circle() {
        this(1, "red");
    }

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

}
```

## Class and Object

1. Object Oriented Programming
2. Class
3. Constructor
4. **Method Overloading**
5. Instance and Class Members
6. Object Life Cycle

## Method Overloading

- Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.
- In order to overload a method, the argument lists of the methods must differ in:

    1. Number of parameters.
    2. Data type of parameters.
    3. Sequence of Data type of parameters.

## Method Overloading Examples

| | |
|---|---|
| add(int, int)<br>add(int, int, int) | add(int, int)<br>add(int, float) |
| add(int, float)<br>add(float, int) | ~~int add(int, int)~~<br>~~float add(int, int)~~ |

## Constructor Overloading

- Java constructor can be overloaded like conventional methods.

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. The compiler differentiates them by the number of parameters in the list and their types.

## Constructor overloading example

```
public class Circle {

    private double radius;//radius in cm
    private String color;

    public Circle() {
        this(1, "red");
    }

    public Circle(double radius) {
        this(radius, "red");
    }

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

}
```

## Class and Object

1. Object Oriented Programming
2. Class
3. Constructor
4. Method Overloading
5. **Instance and Class Members**
6. Object Life Cycle

## Instance and Class Members

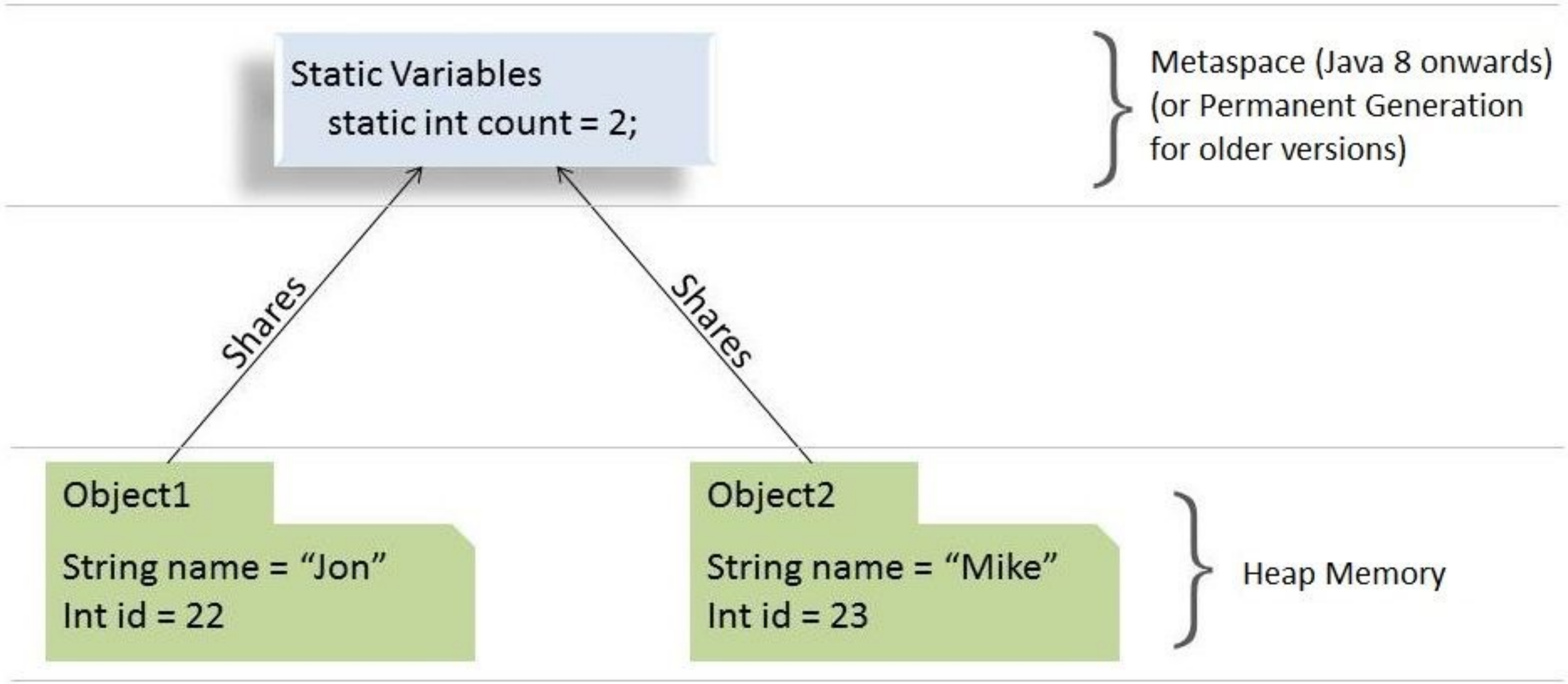
- The **static** word can be used to attach a Variable or Method to a Class. The variable or Method that are marked static belongs to the Class rather than to any particular instance.

- Static variable and Methods can be used **without** having an instance of the Class.

```
public class JavaStaticExample {
   static int i = 10;
   static void method() {
         System.out.println("Inside Static method");
   }

   public static void main(String[] args) {
         // Accessing Static method
         JavaStaticExample.method();
         // Accessing Static Variable
         System.out.println(JavaStaticExample.i);
         // Accessing static method using reference. Warning by compiler
          JavaStaticExample obj1 = new JavaStaticExample();
         System.out.println(obj1.i);
         }
}
```

## Static Example


Static Variables
static int count = 2;
Metaspace (Java 8 onwards) (or Permanent Generation for older versions)
Shares
Shares
Object1
String name = "Jon"
Int id = 22
Object2
String name = "Mike"
Int id = 23
Heap Memory

## Differences between static and non static

| Static Members | Non Static Members |
|---|---|
| Static members can be accessed using class name | Non static members can be accessed using instance of a class |
| Static members can be accessed by static and non static methods | Non static members cannot be accessed inside a static method. |
| Static variables reduce the amount of memory used by a program. | Non static variables do not reduce the amount of memory used by a program |
| Static variables are shared among all instances of a class. | Non static variables are specific to that instance of a class. |
| Static variable is like a global variable and is available to all methods. | Non static variable is like a local variable and they can be accessed through only instance of a class. |

## Static Block

- The static block is a block of statement inside a Java class that will be executed when a class is first loaded into the JVM
- A static block helps to initialize the static data members, just like constructors help to initialize instance members

```
class className{
    static {
    //Code goes here
    }
}
```

## Static Block Example

```
public class Test {
    static int i;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    public static void main(String args[]) {
        System.out.print("Gona print Test.i:");
        System.out.println(Test.i);
    }
}
```

## Class and Object

1. Object Oriented Programming
2. Class
3. Constructor
4. Method Overloading
5. Instance and Class Members
6. **Object Life Cycle**

## Life Cycle of an Object

1. Before an object can be created from a class, the class must be **loaded**. To do that, the Java runtime locates the class on disk (in a .class file) and reads it into memory.

2. An object is **created** from a class when you use the **new** keyword. To initialize the class, Java allocates memory for the object and sets up a reference to the object so that the Java runtime can keep track of it.

3. The object lives its life, providing access to its public methods and fields to whoever wants and needs them.

4. When it's time for the object to die, the object is removed from memory, and Java drops its internal reference to it. You don't have to destroy objects yourself. A special part of the Java runtime called the "**garbage collector**" takes care of destroying all objects when they are no longer in use.

## Garbage Collection

- Garbage means **unreferenced** objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- 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.

## How can an object be unreferenced?

- By nulling a reference:
  - Employee e=new Employee();
  - e=null;
- By assigning a reference to another:
  - Employee e=new Employee();
  - e=new Employee();
  - //now the first object referred by e is available for garbage collection
- By anonymous object:
  - new Employee();

## finalize() method

- The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

  **protected void finalize(){}**

- The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

- The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

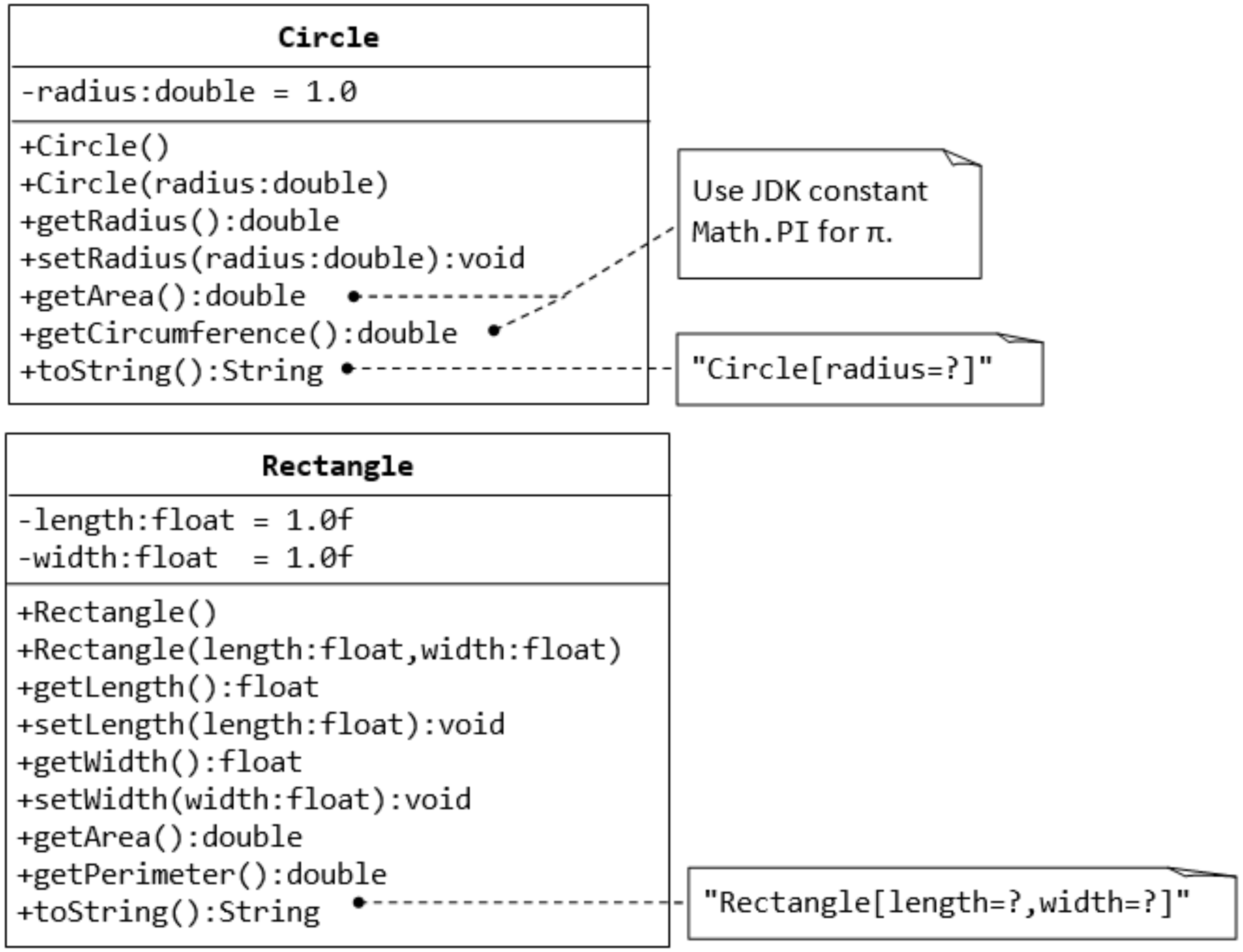
  **public static void gc(){}**

- Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

```
public class TestGarbage{
    public void finalize(){
        System.out.println("object is garbage collected");
    }
    public static void main(String args[]){
        TestGarbage s1=new TestGarbage();
        TestGarbage s2=new TestGarbage();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

Garbage Collection Example

## Homework: Circle and Rectangle



```
Circle
-radius:double = 1.0
+Circle()
+Circle(radius:double)
+getRadius():double
+setRadius(radius:double):void
+getArea():double
+getCircumference():double
+toString():String
```

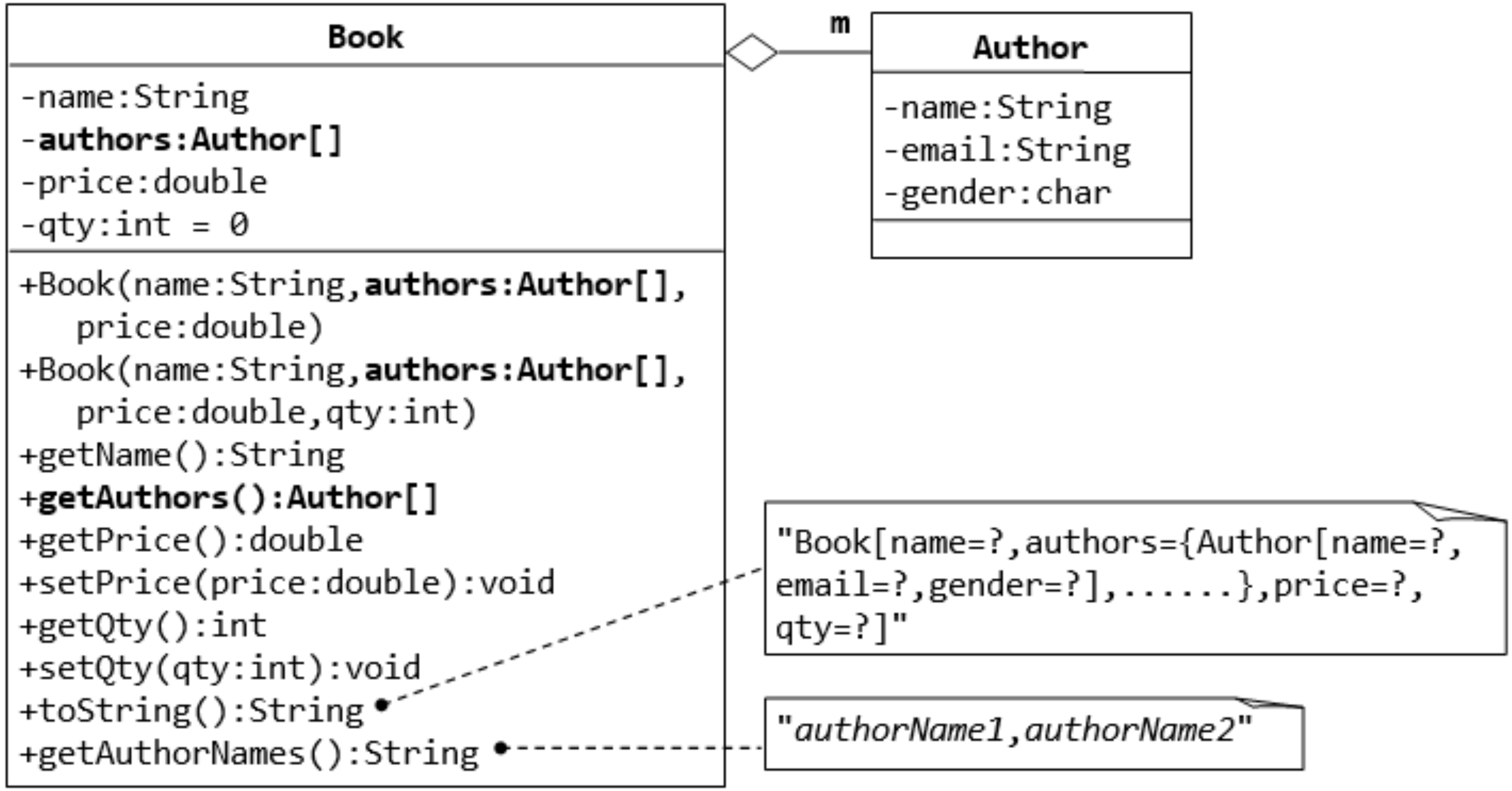
Use JDK constant Math.PI for π.

"Circle[radius=?]"

```
Rectangle
-length:float = 1.0f
-width:float  = 1.0f
+Rectangle()
+Rectangle(length:float,width:float)
+getLength():float
+setLength(length:float):void
+getWidth():float
+setWidth(width:float):void
+getArea():double
+getPerimeter():double
+toString():String
```

"Rectangle[length=?,width=?]"

# Homework 2: Book and Authors


Book
-name:String
-**authors:Author[]**
-price:double
-qty:int = 0
+Book(name:String,**authors:Author[]**, price:double)
+Book(name:String,**authors:Author[]**, price:double,qty:int)
+getName():String
**+getAuthors():Author[]**
+getPrice():double
+setPrice(price:double):void
+getQty():int
+setQty(qty:int):void
+toString():String
+getAuthorNames():String
m
Author
-name:String
-email:String
-gender:char
"Book[name=?,authors={Author[name=?,email=?,gender=?],......},price=?,qty=?]"
"*authorName1,authorName2*"