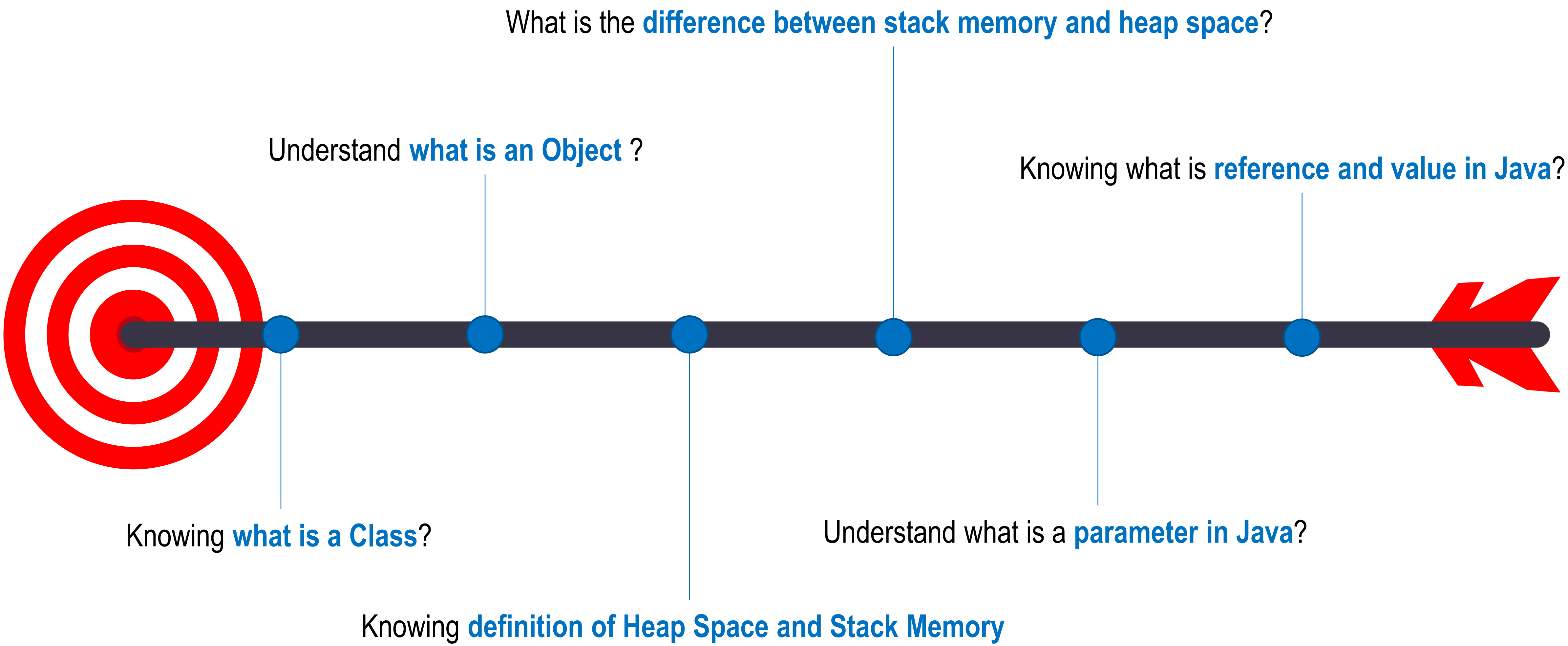
CLASSES AND OBJECT

Instructor: DieuNT1

Agenda

- **OOP Concepts**
  - ✓ What is a Class?
  - ✓ What is an Object
  - ✓ Define Classes for Objects
  - ✓ Constructors and Destructors
- **static Keyword in Java**
  - ✓ Static Variables
  - ✓ Static Methods
- **final Keyword in Java**
  - ✓ Final Class, Variables, Methods
  - ✓ Constants
- **Stack and Heap Memory**
- **Passing Objects to Methods**

Lesson Objectives



Knowing **what is a Class**?

Understand **what is an Object** ?

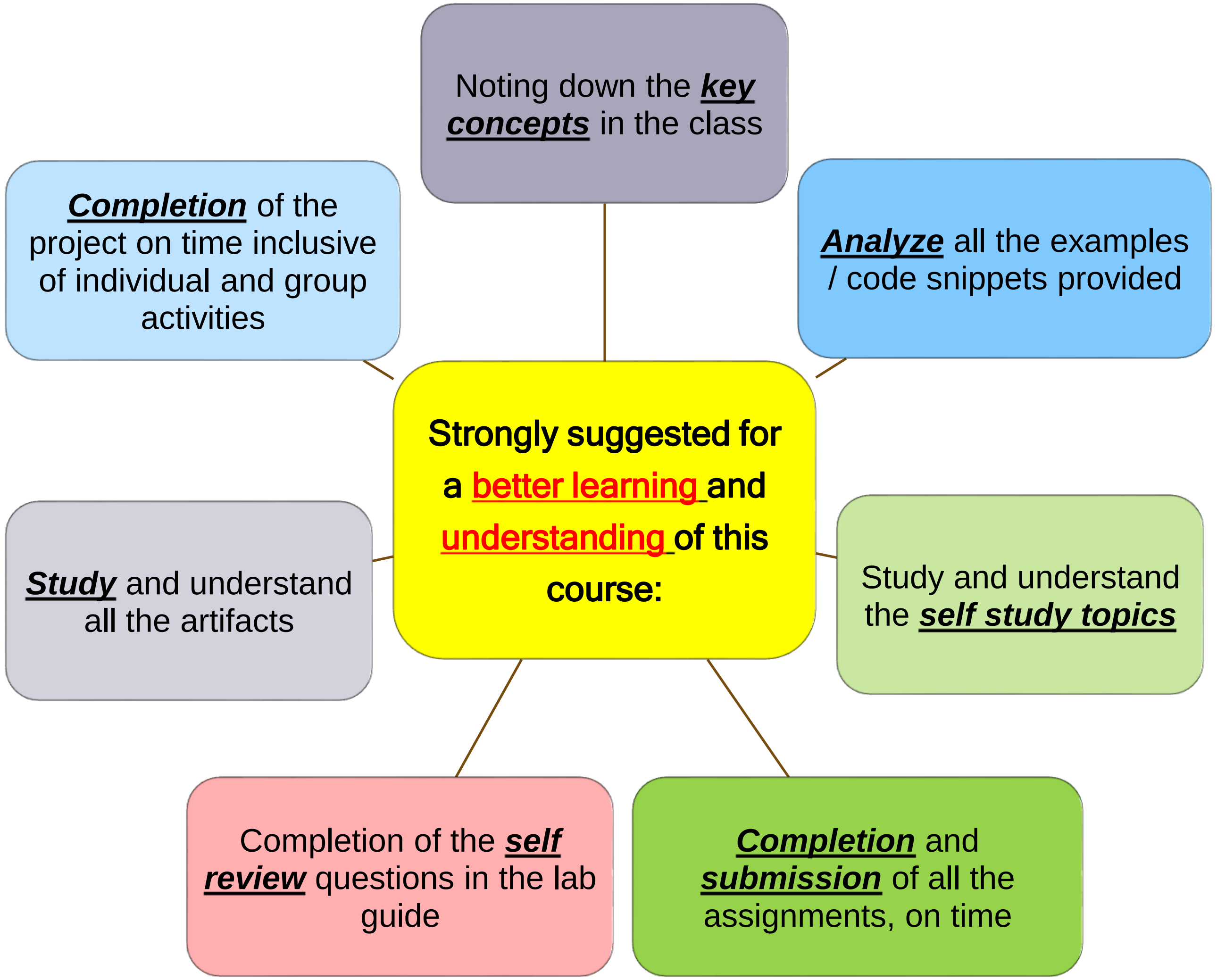
Knowing **definition of Heap Space and Stack Memory**

What is the **difference between stack memory and heap space**?

Understand what is a **parameter in Java**?

Knowing what is **reference and value in Java**?

Learning Approach



Strongly suggested for a better learning and understanding of this course:

Noting down the ***key concepts*** in the class

Analyze all the examples / code snippets provided

Study and understand the ***self study topics***

Completion and ***submission*** of all the assignments, on time

Completion of the ***self review*** questions in the lab guide

Study and understand all the artifacts

Completion of the project on time inclusive of individual and group activities

Section 1

OOPs Concepts

What is a Class?

- A class can be considered as a **blueprint** using which you can create as many objects.
- For example, create a class **House** that has three instance variables:

```java
public class House {
  String address;
  String color;
  double are;
  void openDoor() {
    // TODO
  }
  void closeDoor() {
    // TODO
  }
}
```

```java
public class HouseManagement {
  public static void main(String[] args) {
    House house1 = new House("Duytan", "Blue", 1000);
    House house2 = new House("Tonthatthuyet", "Green", 1200);
    System.out.println(house1.address + "\t" + house1.color +
            "\t" + house1.are);
    System.out.println(house2.address + "\t" + house2.color +
             "\t" + house2.are);
  }
}
```

- This is just a *blueprint*, it does not represent any House
- We have created two objects, while creating objects we provided separate properties to the objects using constructor.

What is an Object

Object

- ✓ An object is an instance of a class.
- ✓ You can create **many** instances of a class.

Objects have two characteristics

- ✓ Objects have unique identity
- ✓ They have states and behaviors.

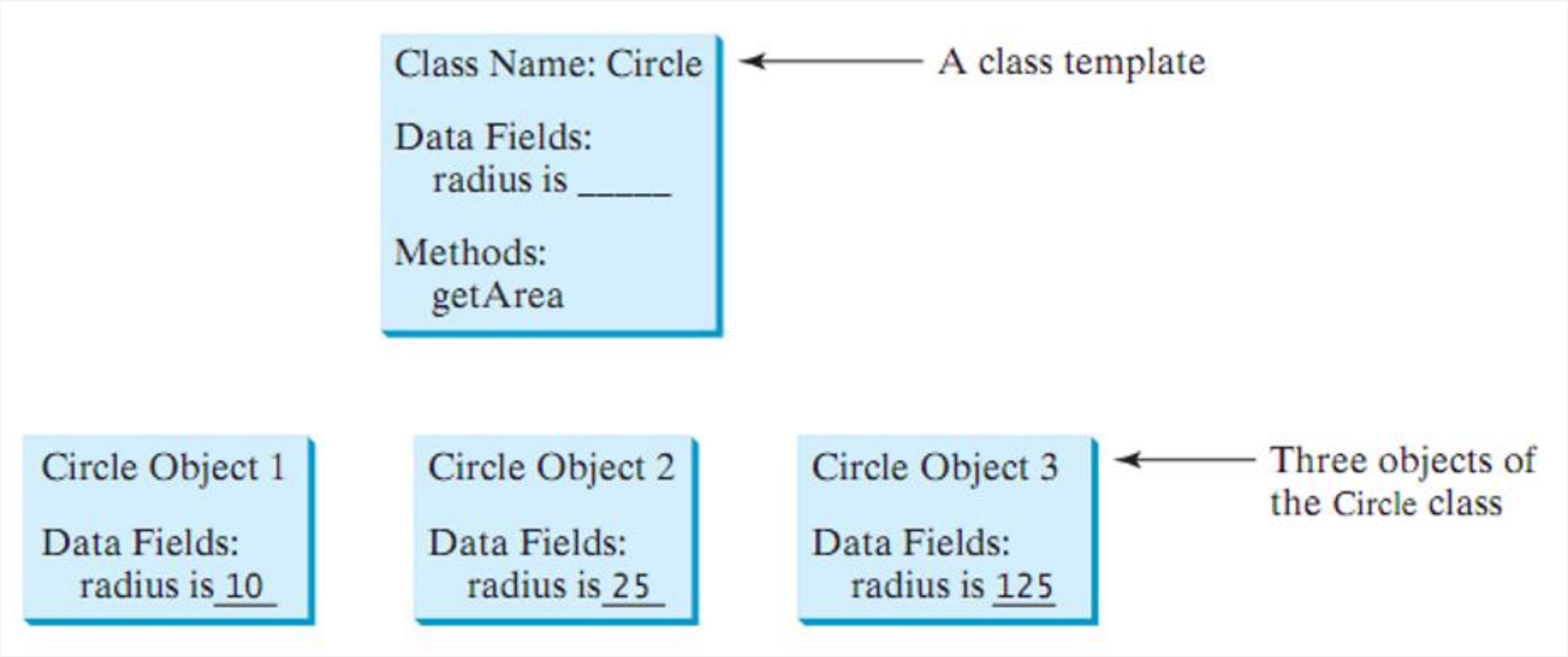
Example of states and behaviors

Objects represent identifiable real-world entities**. Eg**: house

- ✓ **States**: address, color, area
- ✓ **Behaviors**: open door, close door

Class, Object/Instance

- **Example:**



Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea

A class template

Circle Object 1

Data Fields:
radius is 10

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of
the Circle class

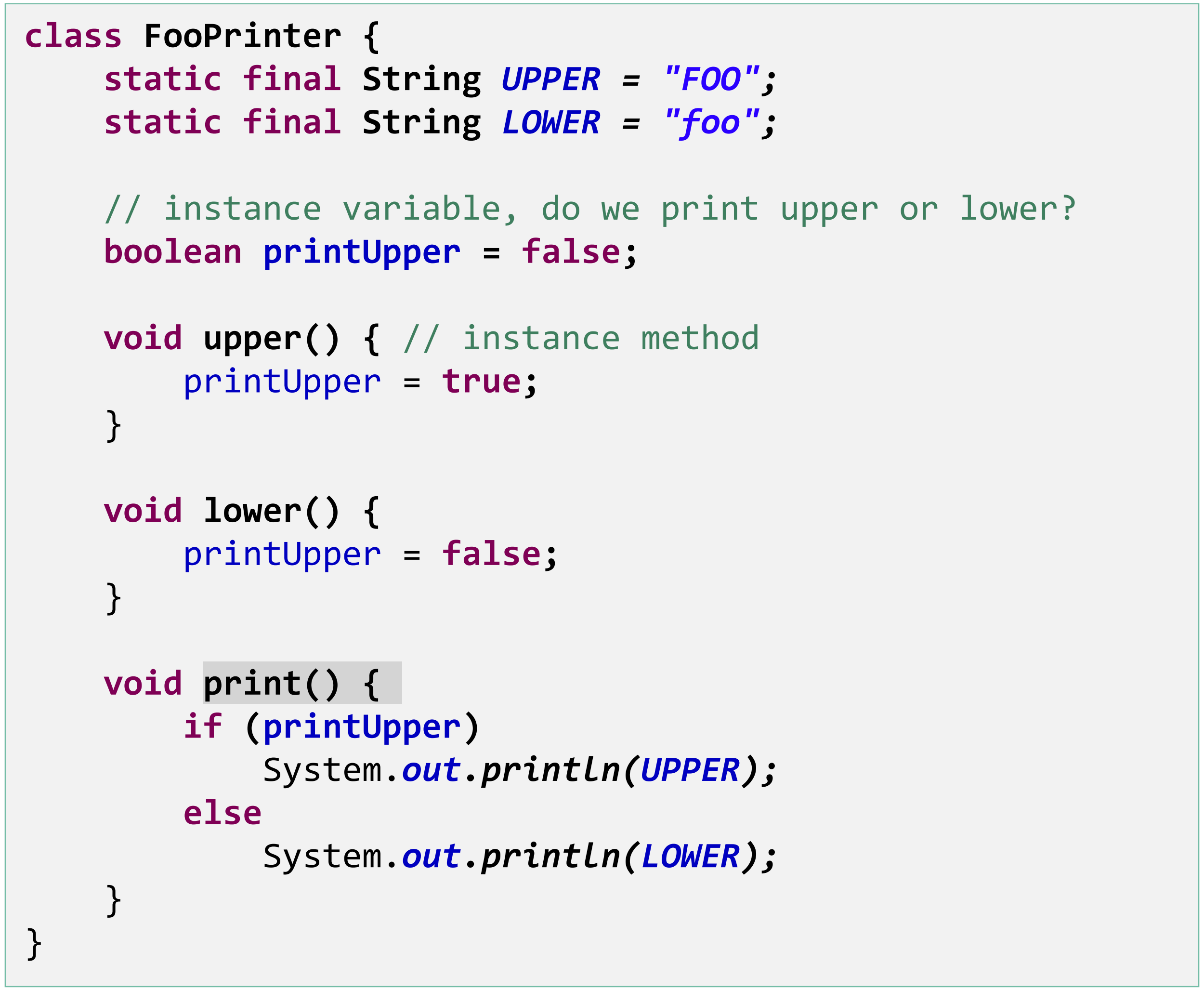
Define Classes for Objects

- Create new object type with `class` keyword.
- A class definition can contain:
  - ✓ instance variables (attribute/fields)
  - ✓ constructors
  - ✓ methods (instance method, static method)
- **Syntax:**

```
[public][<abstract><final>]class <ClassName>
[extends <SuperClass>] [implements <InterfaceName>]{
    <Attribute/Field declarations { initialization code }>
    <Constructors>
    <Methods>
}
```

Define Classes for Objects

- **Example:**

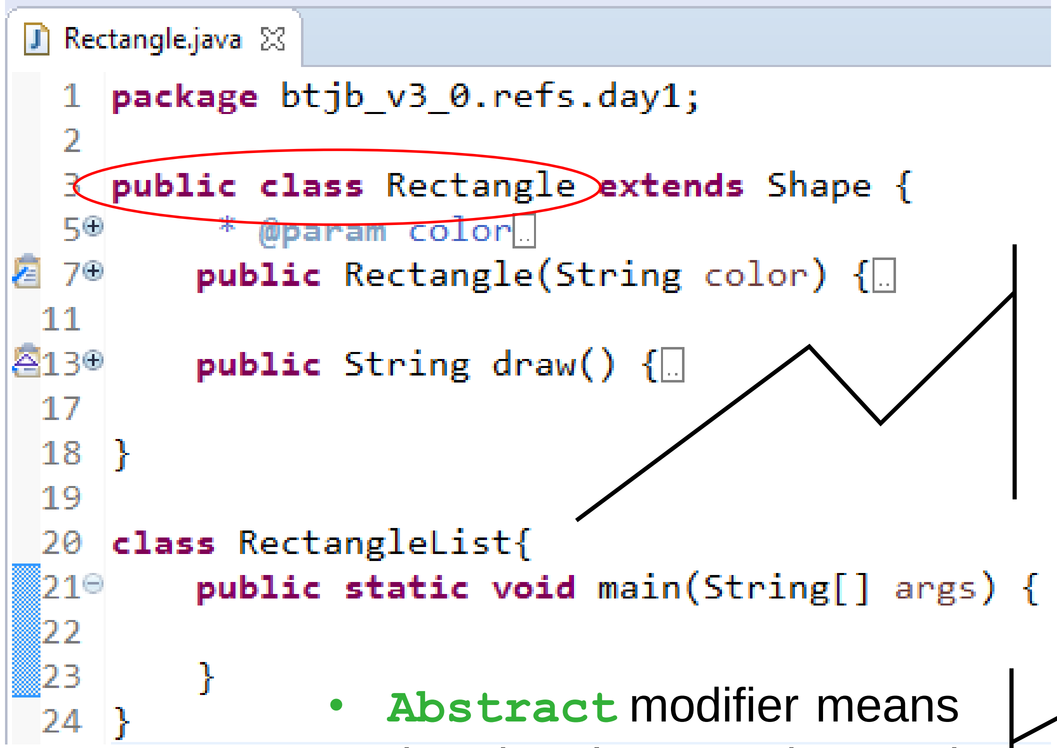


```
class FooPrinter {
    static final String UPPER = "FOO";
    static final String LOWER = "foo";

    // instance variable, do we print upper or lower?
    boolean printUpper = false;

    void upper() { // instance method
        printUpper = true;
    }

    void lower() {
        printUpper = false;
    }

    void print() {
        if (printUpper)
            System.out.println(UPPER);
        else
            System.out.println(LOWER);
    }
}
```

Define Classes for Objects

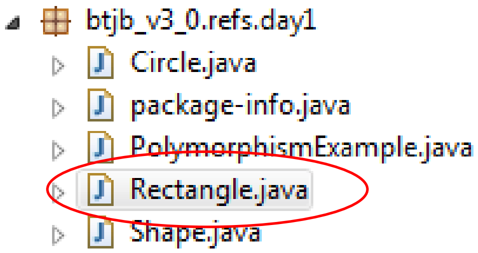
Class Modifiers

- **public**: that class is visible to all classes everywhere.
  - ✓ only one public class per file, must have same name as the file (this is how Java finds it!).



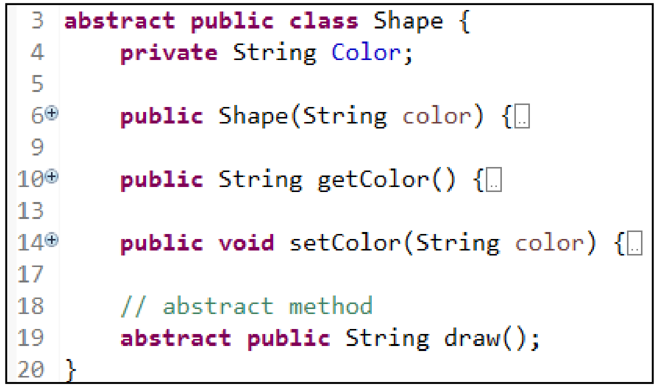
Rectangle.java

```
package btjb_v3_0.refs.day1;

public class Rectangle extends Shape {
         * @param color□
    public Rectangle(String color) {□

    public String draw() {□

}

class RectangleList{
    public static void main(String[] args) {

    }
}
```



```
btjb_v3_0.refs.day1
    Circle.java
    package-info.java
    PolymorphismExample.java
    Rectangle.java
    Shape.java
```

- ✓ If a class has no modifier (the **default**, also known as **package-private**)
- ✓ It is visible only within its own package.

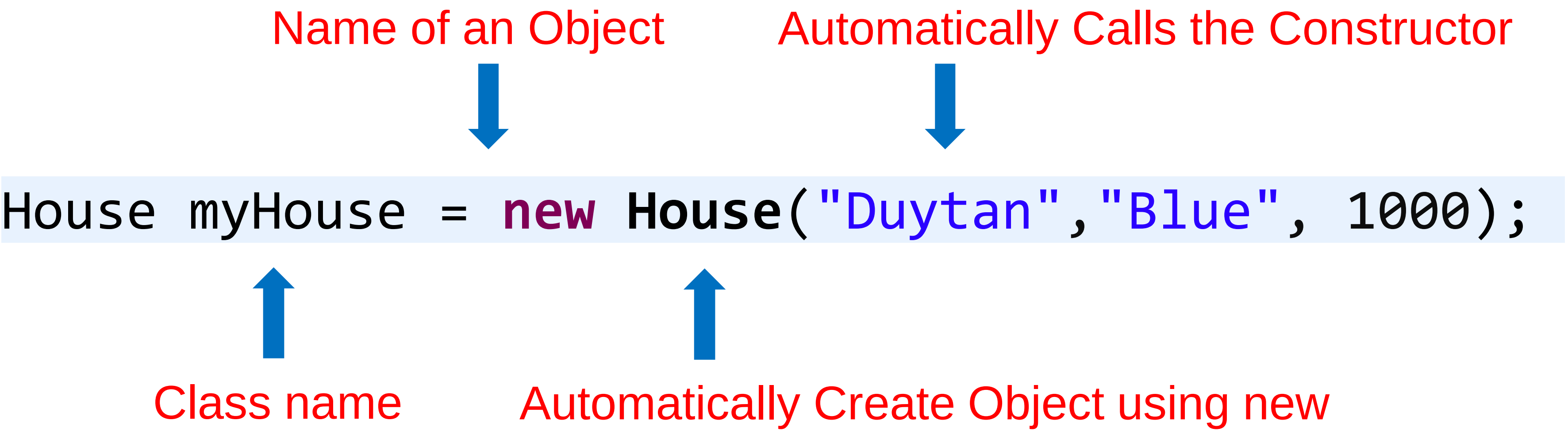


```
abstract public class Shape {
    private String Color;

    public Shape(String color) {□

    public String getColor() {□

    public void setColor(String color) {□

    // abstract method
    abstract public String draw();
}
```

- **Abstract** modifier means that the class can be used as a superclass only.

Creating an Object

- Defining a class does not create an object of that class - this needs to happen explicitly:



Name of an Object

Automatically Calls the Constructor

```
House myHouse = new House("Duytan","Blue", 1000);
```

Class name

Automatically Create Object using new

- In general, **an object must be created** before any *methods can be called*.
    - ✓the exceptions are *static* methods.

What does it mean to create an object?

```
public class SimpleClass {
    public static void main(String[] args) {
        FooPrinter foo = new FooPrinter();
        foo.print();
        foo.upper();
        foo.print();
    }
}
```

```
Output:
foo
FOO
```

- An object is a chunk of memory:
  - ✓holds field values
  - ✓holds an associated object type
- All objects of the same type share code
  - ✓they all have same object type, but can have different field values.

Constructors

A **constructor** is invoked to create an object using the **new** operator.

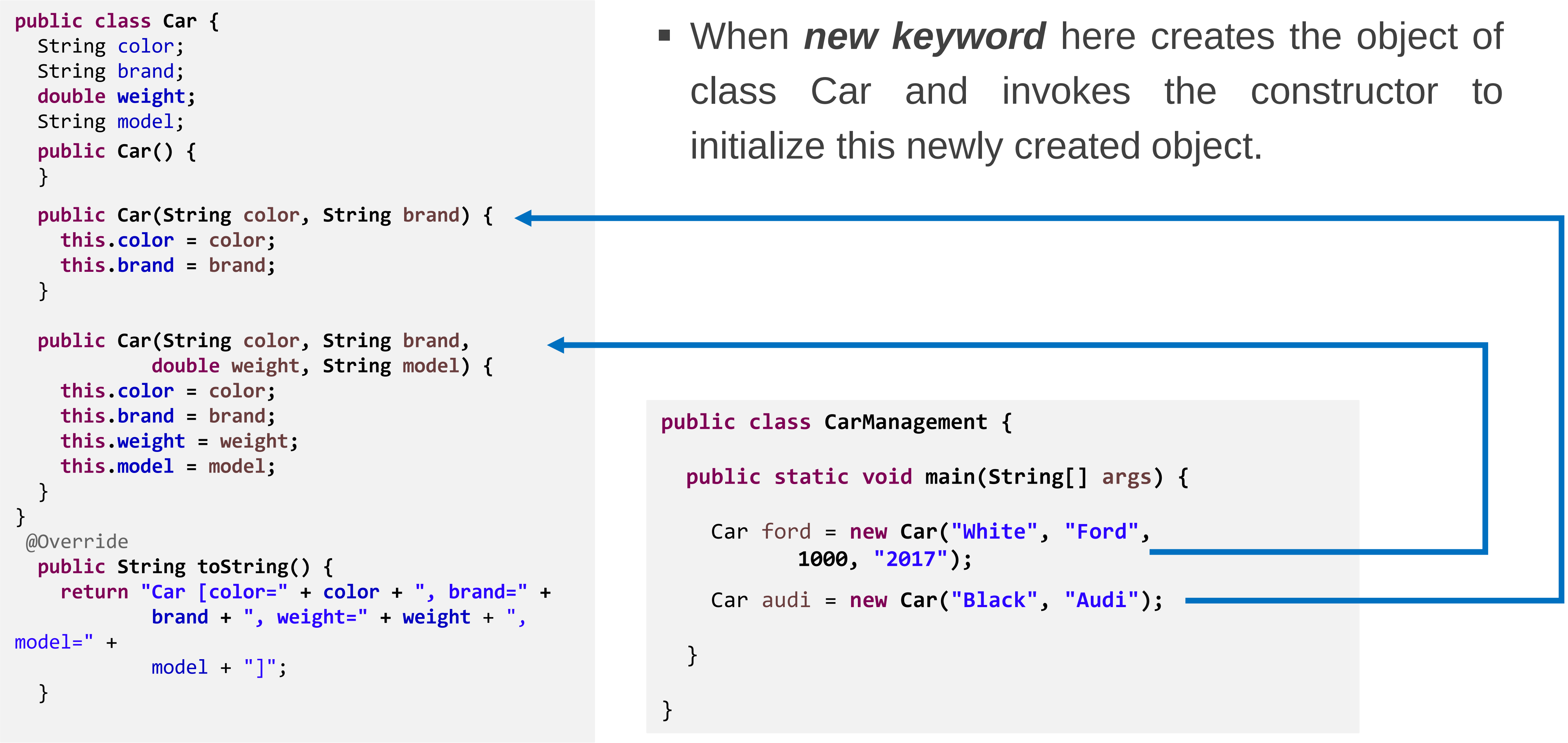
- Constructor is a block of code that initializes the newly created object.
  - ✓ Constructor has same name as the class
  - ✓ People often refer constructor as special type of method in Java. It doesn’t have a return type
- You can create multiple constructors, each must accept **different parameters**.
- If you don't write any constructor, the compiler will (in effect) write one for you:

```
FooPrinter(){}
```

- If you include any constructors in a class, the compiler will not create a default constructor!

How does a constructor work

- When ***new keyword*** here creates the object of class Car and invokes the constructor to initialize this newly created object.



```
public class Car {
  String color;
  String brand;
  double weight;
  String model;
  public Car() {
  }

  public Car(String color, String brand) {
    this.color = color;
    this.brand = brand;
  }

  public Car(String color, String brand,
            double weight, String model) {
    this.color = color;
    this.brand = brand;
    this.weight = weight;
    this.model = model;
  }
}
 @Override
  public String toString() {
    return "Car [color=" + color + ", brand=" +
            brand + ", weight=" + weight + ",
model=" +
            model + "]";
  }
```

```
public class CarManagement {

  public static void main(String[] args) {

    Car ford = new Car("White", "Ford",
           1000, "2017");

    Car audi = new Car("Black", "Audi");

  }

}
```

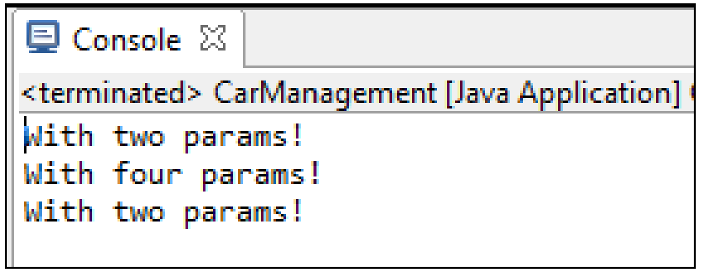
Multiple (overload) Constructors

- Must accept different parameters.
- One constructor can call another, use ***this***, not the classname:

```java
public class Car {
  String color;
  String brand;
  double weight;
  String model;

  public Car() {
    System.out.println("No params!");
  }

  public Car(String color, String brand) {
    this.color = color;
    this.brand = brand;
    System.out.println("With two params!");
  }

  public Car(String color, String brand,
                      double weight, String model) {
    this(color, brand);
    this.weight = weight;
    this.model = model;
    System.out.println("With four params!");
  }
}
```

```java
public class CarManagement {

  public static void main(String[] args) {
    Car ford = new Car("White", "Ford", 1000, "2017");

    Car audi = new Car("Black", "Audi");

  }

}
```

What will print out?



```
<terminated> CarManagement [Java Application]
With two params!
With four params!
With two params!
```

Destructors

Nope!

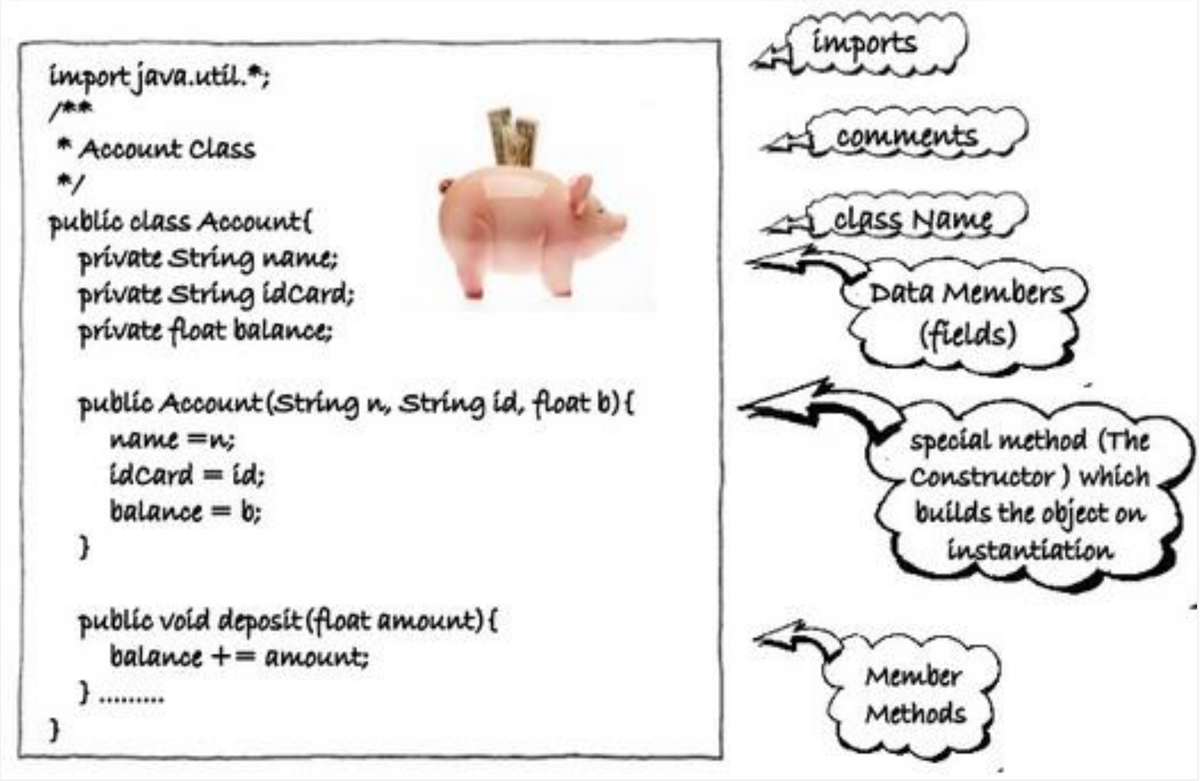
There is a **finalize()** method that is called when an object is destroyed:

- You don't have control over when the object is destroyed (it might never be destroyed).
- The JVM garbage collector takes care of destroying objects automatically (you have limited control over this process).

Instance variable (Field)

- Instance variable in Java is used by objects to store their states
- **Fields** (data members) can be any primitive or reference type
- **Syntax**:

```
[Access modifier] <Data type> <field_name>;
```



```
import java.util.*;
/**
 * Account Class
 */
public class Account{
   private String name;
   private String idCard;
   private float balance;

   public Account(String n, String id, float b){
      name =n;
      idCard = id;
      balance = b;
   }

   public void deposit(float amount){
      balance += amount;
   } .........
}
```

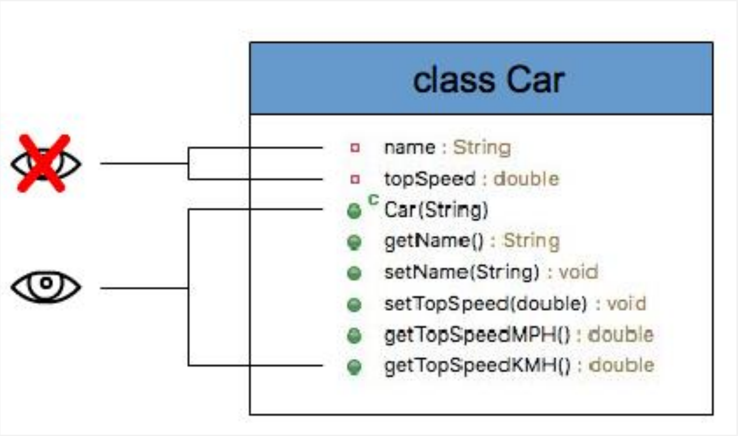
imports
comments
class Name
Data Members (fields)
special method (The Constructor) which builds the object on instantiation
Member Methods

Instance variable (Field)

- The following table shows the **access** to members permitted by each **modifier**:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- **Example:**



class Car

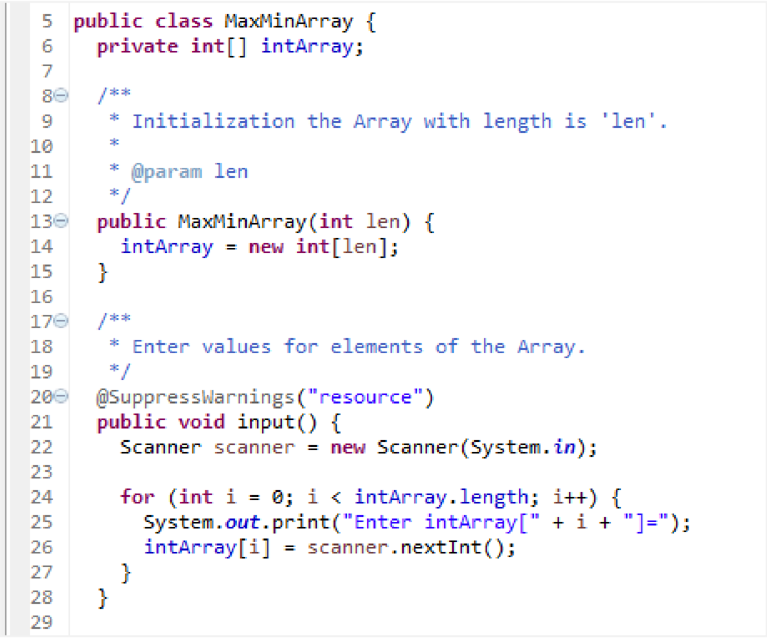
name : String
topSpeed : double
Car(String)
getName() : String
setName(String) : void
setTopSpeed(double) : void
getTopSpeedMPH() : double
getTopSpeedKMH() : double

Instance method

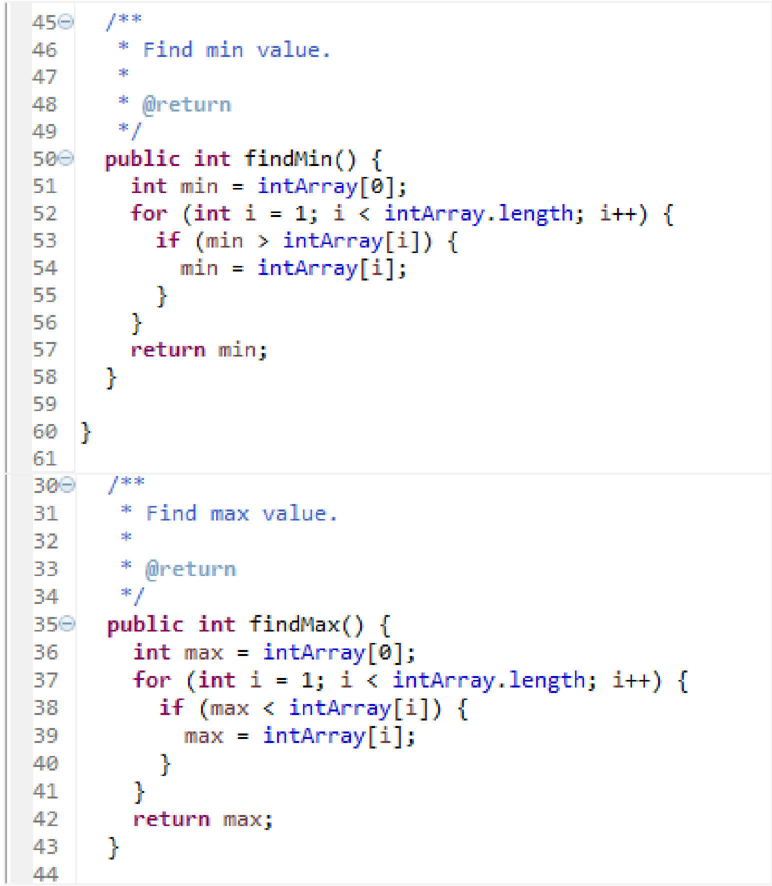
- **Instance method** are methods which require an object of its class to be created before it can be called.
- **Access modifiers**: same idea as with fields.
  - ✓`private`/`protected`/`public`/`no modifier`:
- **No access modifier:**
  - ✓`abstract:` no implementation given, must be supplied by subclass.
  - ✓`final`: the method cannot be changed by a subclass/cannot be overridden by subclasses (no alternative implementation can be provided by a subclass).

Instance method

- Example, create **MinMaxArray** class have:
  - ✓ an instance variable is intArray
  - ✓ three instance methods are input(), findMax(), findMin() as bellow



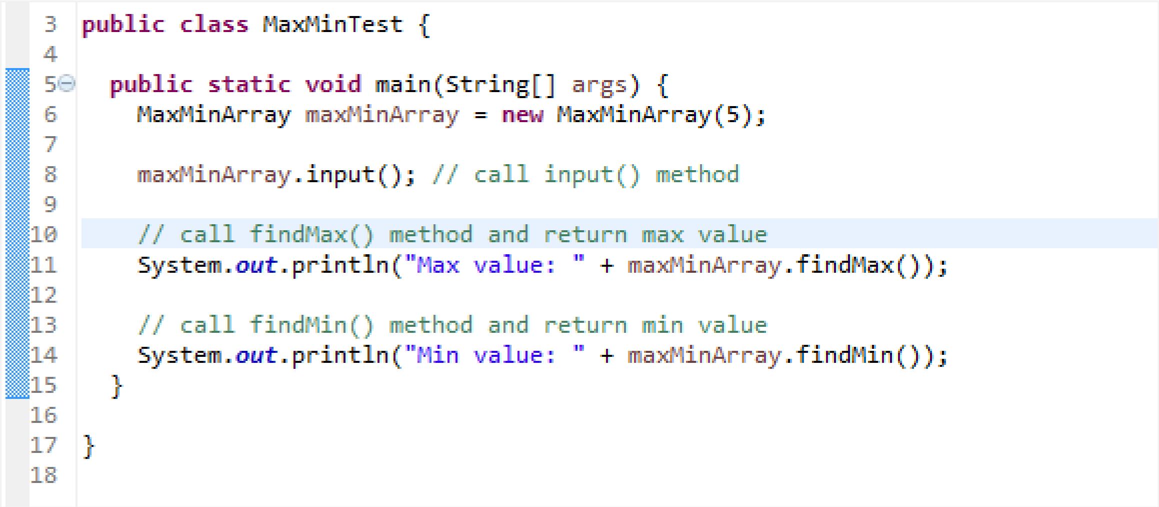
```java
public class MaxMinArray {
  private int[] intArray;

  /**
   * Initialization the Array with length is 'len'.
   *
   * @param len
   */
  public MaxMinArray(int len) {
    intArray = new int[len];
  }

  /**
   * Enter values for elements of the Array.
   */
  @SuppressWarnings("resource")
  public void input() {
    Scanner scanner = new Scanner(System.in);

    for (int i = 0; i < intArray.length; i++) {
      System.out.print("Enter intArray[" + i + "]=");
      intArray[i] = scanner.nextInt();
    }
  }

```



```java
  /**
   * Find min value.
   *
   * @return
   */
  public int findMin() {
    int min = intArray[0];
    for (int i = 1; i < intArray.length; i++) {
      if (min > intArray[i]) {
        min = intArray[i];
      }
    }
    return min;
  }

}

  /**
   * Find max value.
   *
   * @return
   */
  public int findMax() {
    int max = intArray[0];
    for (int i = 1; i < intArray.length; i++) {
      if (max < intArray[i]) {
        max = intArray[i];
      }
    }
    return max;
  }

```

Instance method

- Create **MinMaxTest** class with main() method:
  - ✓Create an object named minMaxArray
  - ✓Call 3 methods and see the output



```java
public class MaxMinTest {

	public static void main(String[] args) {
		MaxMinArray maxMinArray = new MaxMinArray(5);

		maxMinArray.input(); // call input() method

		// call findMax() method and return max value
		System.out.println("Max value: " + maxMinArray.findMax());

		// call findMin() method and return min value
		System.out.println("Min value: " + maxMinArray.findMin());
	}

}
```

Output:

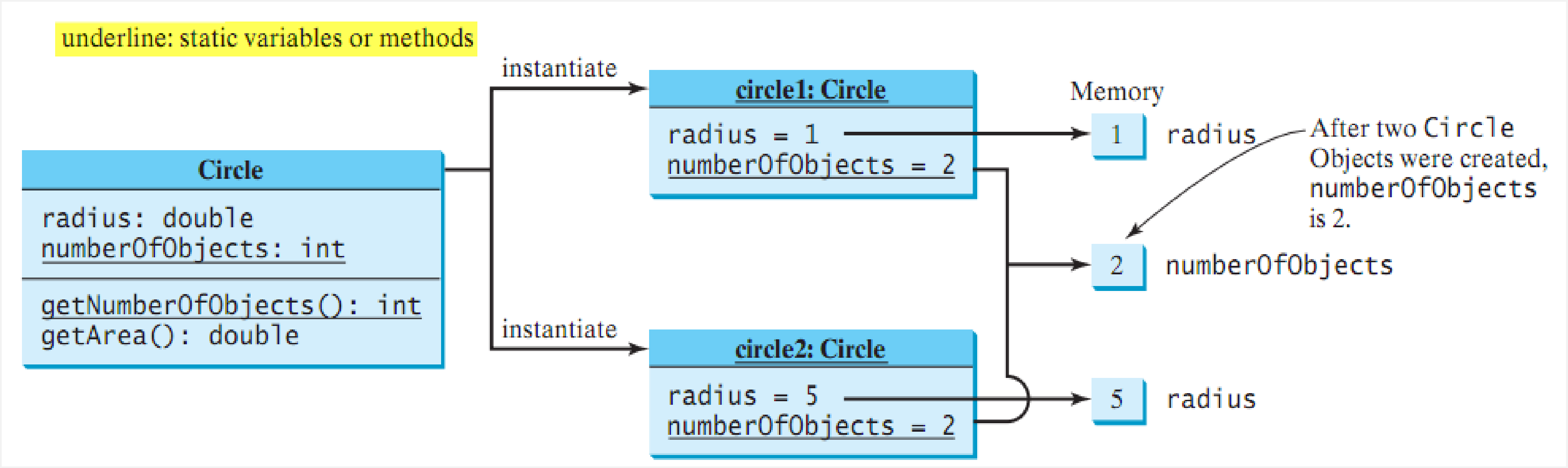
```
Enter intArray[0]=4
Enter intArray[1]=2
Enter intArray[2]=-2
Enter intArray[3]=8
Enter intArray[4]=3
Max value: 8
Min value: -2
```

Section 2

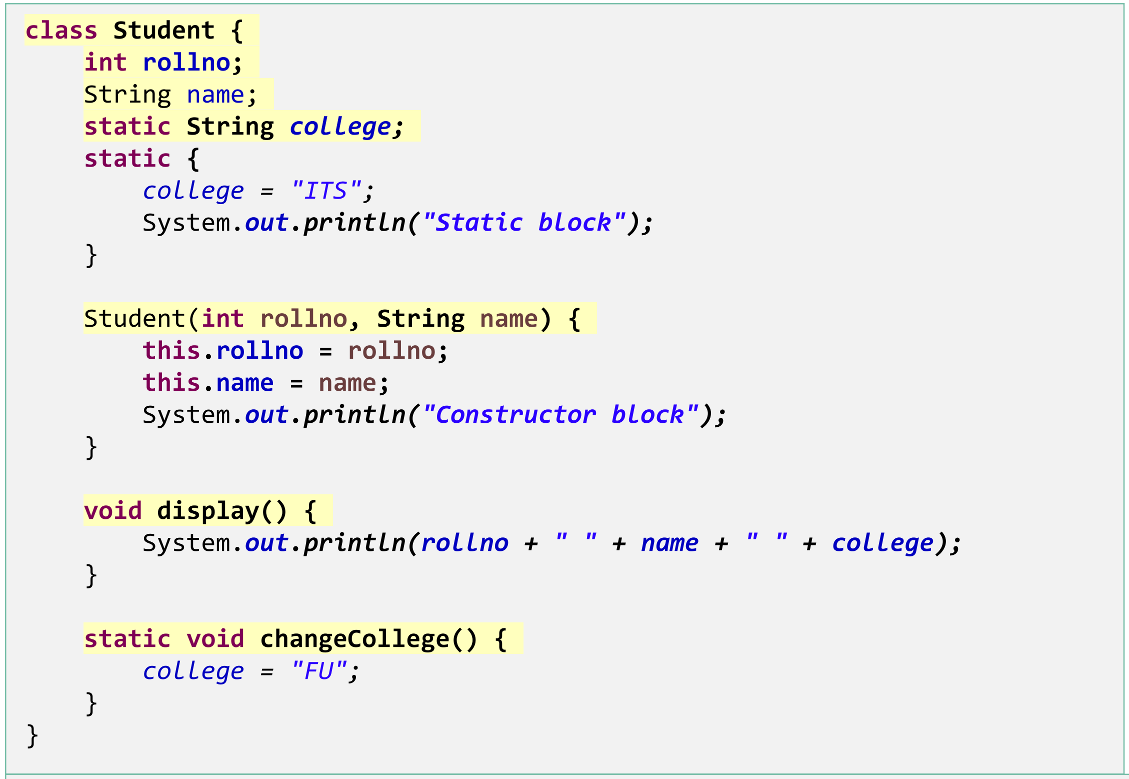
static Keyword in Java

Static variables

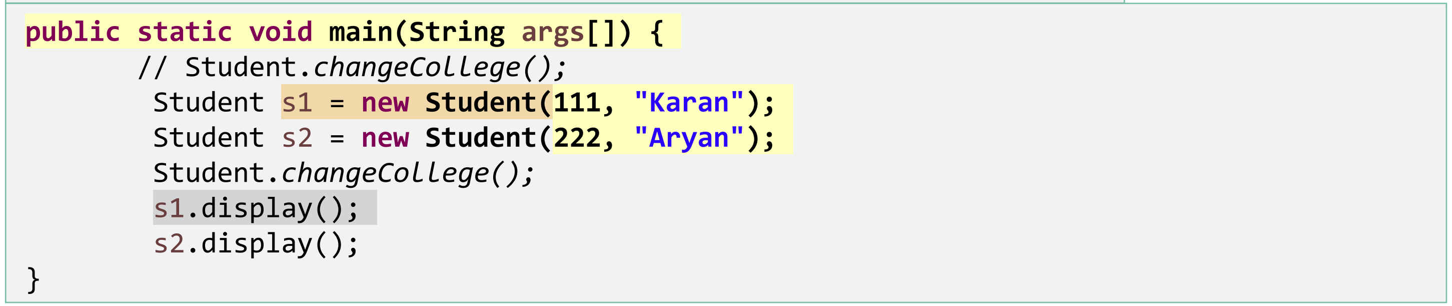
Static variables	Instance variables
▪ Fields declared static are called class fields (class variables). ▪ There is only one copy of a static field, no matter how many objects are created.	▪ Instance variables: tied to specific object, they are not shared between objects .



Static variables Example



```java
class Student {
    int rollno;
    String name;
    static String college;
    static {
        college = "ITS";
        System.out.println("Static block");
    }

    Student(int rollno, String name) {
        this.rollno = rollno;
        this.name = name;
        System.out.println("Constructor block");
    }

    void display() {
        System.out.println(rollno + " " + name + " " + college);
    }

    static void changeCollege() {
        college = "FU";
    }
}
```



```java
public static void main(String args[]) {
       // Student.changeCollege();
        Student s1 = new Student(111, "Karan");
        Student s2 = new Student(222, "Aryan");
        Student.changeCollege();
        s1.display();
        s2.display();
}
```

Output:

```
111          Karan        FU
222          Aryan        FU
```

Static vs Non-Static Variables

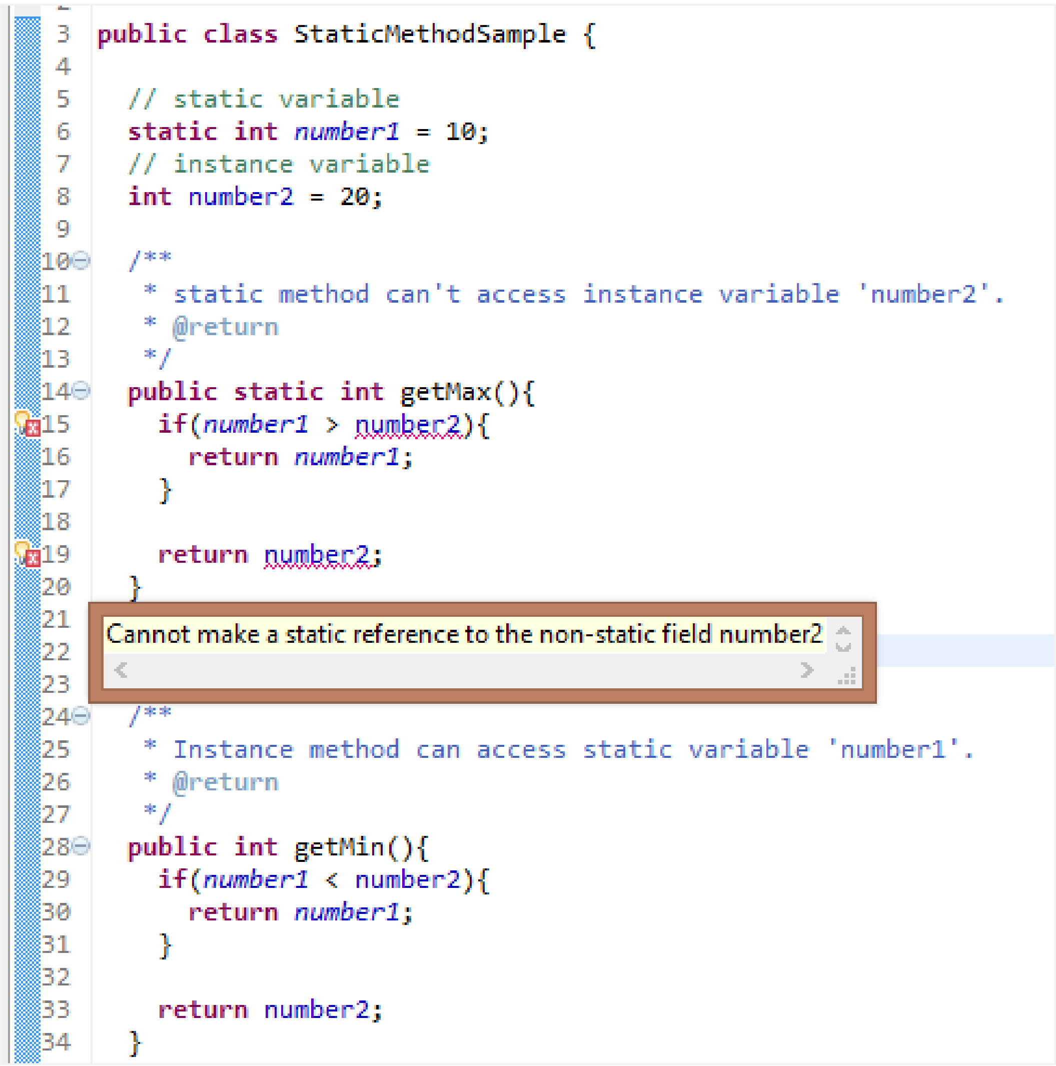
Static Variables	Non-Static Variables
✓ They can access them using class names.	✓ They can be accessed only using objects.
✓ They can access them with static methods as well as non-static methods.	✓ They can be accessed only using non-static methods.
✓ They are allocated memory only once while loading the class.	✓ A memory per object is allocated.
✓ These variables are shared by all the objects or instances of the class.	✓ Each object has its own copy of the non-static variables.
✓ Static variables have global scope.	✓ They have local scope.

Static methods

- Static methods are the methods in Java that can be called without creating an object of class.
  - ✓Instance method can access the instance methods and instance variables directly.
  - ✓Instance method can access static variables and static methods directly.
  - ✓Static methods can access the static variables and static methods directly.
  - ✓Static methods can't access instance methods and instance variables directly.
- **Syntax:**

```
static return_type method_name();
```

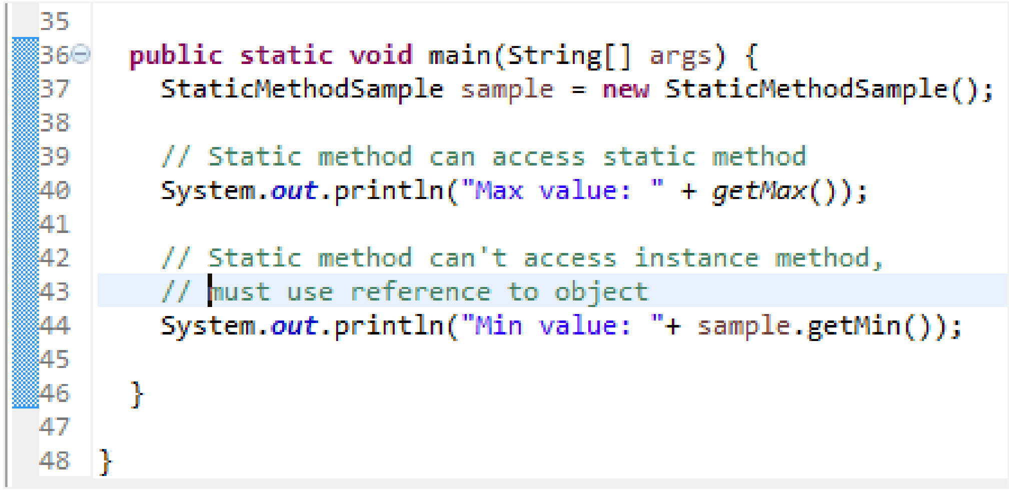
Static methods



```
public class StaticMethodSample {

  // static variable
  static int number1 = 10;
  // instance variable
  int number2 = 20;

  /**
   * static method can't access instance variable 'number2'.
   * @return
   */
  public static int getMax(){
    if(number1 > number2){
      return number1;
    }

    return number2;
  }
```

Cannot make a static reference to the non-static field number2

```
  /**
   * Instance method can access static variable 'number1'.
   * @return
   */
  public int getMin(){
    if(number1 < number2){
      return number1;
    }

    return number2;
  }
```



```
  public static void main(String[] args) {
    StaticMethodSample sample = new StaticMethodSample();

    // Static method can access static method
    System.out.println("Max value: " + getMax());

    // Static method can't access instance method,
    // must use reference to object
    System.out.println("Min value: "+ sample.getMin());

  }

}
```

Static vs Non-Static Methods

Static Methods	Non-Static Methodsnon-Static Methods
✓ These methods support early or compile-time binding.	✓ They support late, run-time, or dynamic binding.
✓ These methods can only access static variables of other classes as well as their own class.	✓ They can access both static as well as non-static members.
✓ You can’t override static methods.	✓ They can be overridden.
✓ **Less memory** consumption since they are allocated memory only once when the class is being loaded.	✓ **Memories** are allocated for each object.

Static Blocks

- **Static blocks** in Java are used to initialize static variables.
  - ✓ They are executed only once when the class is loaded and hence, are perfect for this job.
  - ✓ Can include more than one static block in the class.
  - ✓ Static blocks can only access static variables.
- **Example**:

```java
public class Test {
     static int i = 10;
     static int j;
     static {
          System.out.println("Initializing the Static Variable using Static Block ...");
          j = i * 5;
     }
}
class Main {
     public static void main(String args[]) {
          System.out.println("Value of i is: " + Test.i);
          System.out.println("Value of j is: " + Test.j);
     }
}
```

Static Blocks

- **Output:**

```
Initializing the Static Variable using Static Block ...
Value of i is: 10
Value of j is: 50
```

- **Explain:**
  - ✓ You saw the creation of *two static variables* called i and j inside the Test class.
  - ✓ It went on to initialize variable j using a static block. In the main method, you must use the class name to print the values of i and j static variables.
  - ✓ You can see that the **static block gets executed before the execution of the main method**. When the static block is executed, *it prints the first line regarding the initialization and then initializes the variable j*.
  - ✓ **Then**, the **main method gets executed** which **prints the values of both the variables**.

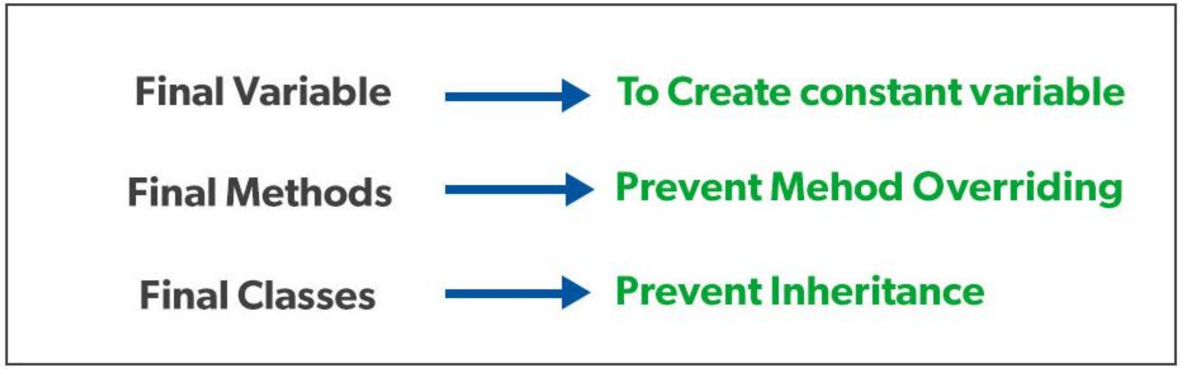
Section 3

final Keyword in Java

final Keyword in Java

final is a non-access modifier applicable only to a variable, it is used in different contexts (a method, or a class).

- The following are different contexts where final is used.



Final Variable → To Create constant variable

Final Methods → Prevent Mehod Overriding

Final Classes → Prevent Inheritance

Final variables

- The keyword **final** means: once the value is set, it cannot be changed. This also means that **you must initialize a final variable.**
  - ✓ They must be ***static*** if they belong to the ***class***.
  - ✓ ***Not be static*** if they belong to the ***instance*** of the class.
- **Examples**:

```
final int THRESHOLD = 5;     // Final variable
final int THRESHOLD;         // Blank final variable
static final double PI = 3.141592653589793; // Final static variable PI
static final double PI;      // Blank final static variable
```

If the **final variable is a reference**, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final collection.

Final variables

- There are some **ways to initialize** a final variable:
  - ✓ A blank final variable can be initialized inside an instance-initializer block or inside the constructor.
  - ✓ A **blank final static variable** can be initialized inside a static block.

If you have more than one constructor in your class then it must be initialized in all of them, otherwise, a compile-time error will be thrown.

```java
public class FinalSample {
    // a final variable and direct initialize
    final int THRESHOLD = 5;
    // a blank final variable
    final int CAPACITY;
    // another blank final variable
    final int MINIMUM;
    // a final static variable PI and direct initialize
    static final double PI = 3.141592653589793;
    // a blank final static variable
    static final double EULERCONSTANT;
    // instance initializer block for initializing CAPACITY
    {
        CAPACITY = 25;
    }
    // static initializer block for initializing EULERCONSTANT
    static {
        EULERCONSTANT = 2.3;
    }
// constructor for initializing MINIMUM
// Note that if there are more than one constructor,
// you must initialize MINIMUM in them also
    public FinalSample() {
        MINIMUM = -1;
    }
}
```

Final classes

When a class is declared with *final* keyword, it is called a final class.
A final class **cannot be extended** (inherited).

- **There are two uses of a final class:**
  - ✓ **Usage 1:** One is definitely to prevent inheritance, as final classes cannot be extended.
  - ✓ For example, all Wrapper Classes like Integer, Float, etc. are final classes. We can not extend them.

```
final class Bike{}

// COMPILE-ERROR! Can't subclass A
class Honda extends Bike {
  void run() {
          System.out.println("running safely with 100kmph");
}
}
```

  - ✓ **Usage 2:** The other use of final with classes is to create an immutable class like the predefined String class. One can not make a class immutable without making it final.

Final methods

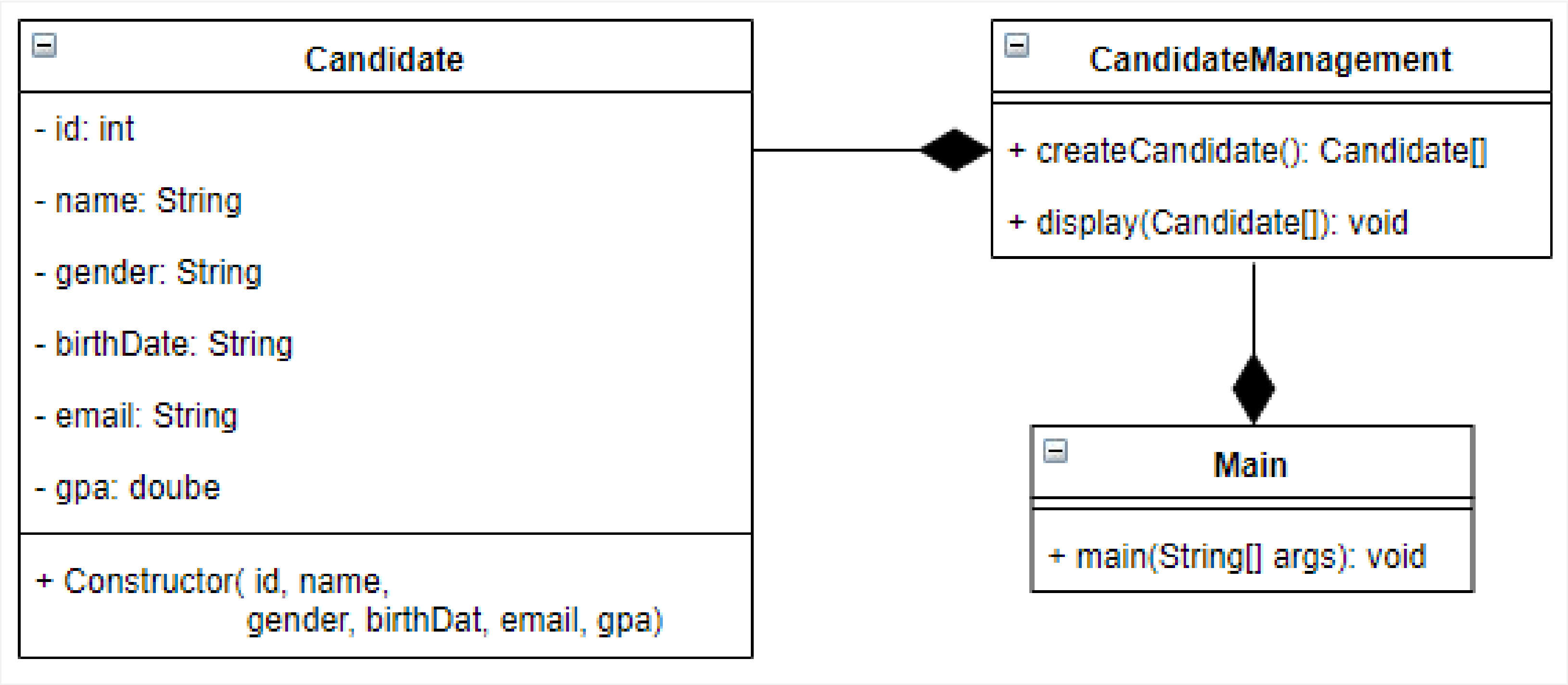
When a method is declared with *final* keyword, it is called a final method.
A final method cannot be overridden.

- **Example:**

```
class Bike {
  final void run(){System.out.println("running");}
}

class Honda extends Bike {

  // Compile-error! We can not override
   void run() {
              System.out.println("running safely with 100kmph");
   }
}
```

Practical time

- Implement the class diagram below by java:

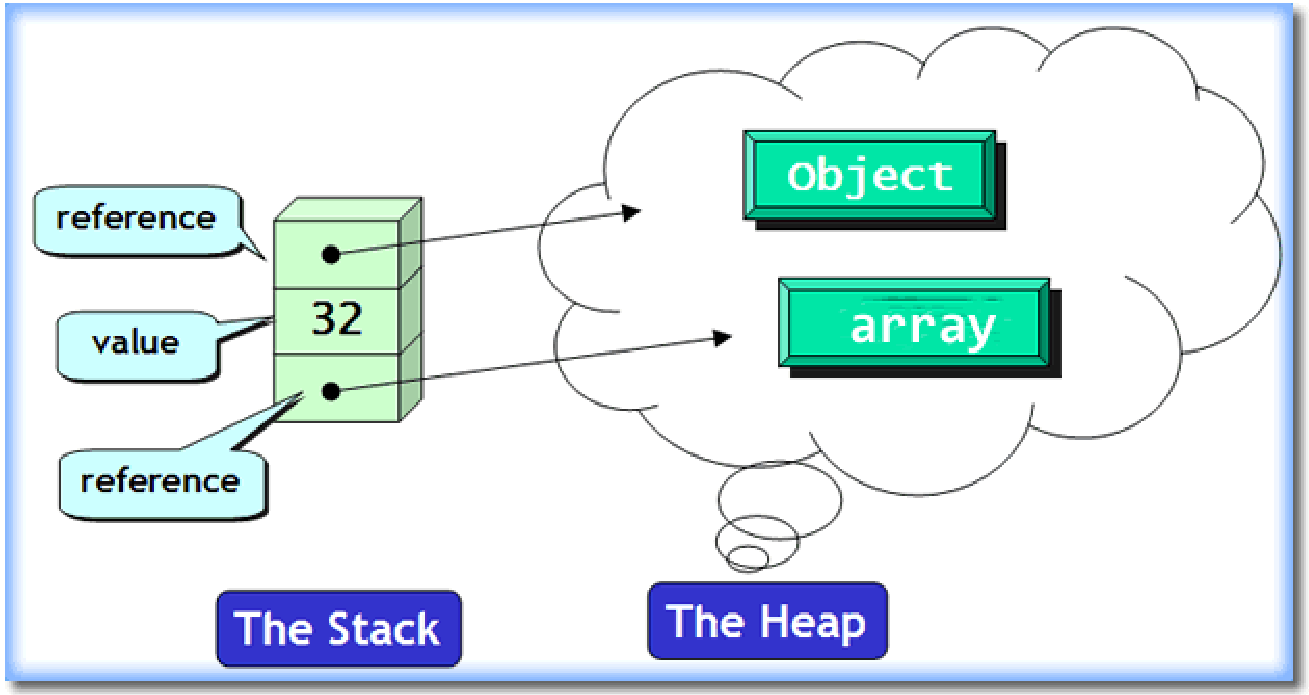

Candidate
- id: int
- name: String
- gender: String
- birthDate: String
- email: String
- gpa: doube
+ Constructor(id, name, gender, birthDat, email, gpa)
CandidateManagement
+ createCandidate(): Candidate[]
+ display(Candidate[]): void
Main
+ main(String[] args): void

Section 2

HEAP SPACE VS STACK MEMORY

Introduction

- To run an application in an optimal way, JVM divides memory into **stack** and **heap memory**.
- **Whenever we declare new variables and objects, call a new method, declare a String, or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space**.



Stack Memory

Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains ***primitive values*** that are specific to a method and ***references*** to objects that are in a heap, referred from the method.

Key Features of Stack Memory

- ✓ It grows and shrinks as new methods are called and returned respectively
- ✓ Variables inside stack **exist only as long as the method that created them is running**
- ✓ It's automatically allocated and deallocated when method finishes execution
- ✓ If this memory is full, Java throws ***java.lang.StackOverFlowErro***r
- ✓ Access to this memory is fast when compared to heap memory
- ✓ This memory is **threadsafe** as each thread operates in its own stack

IMPORTANT *Access to this memory is in **Last-In-First-Out (LIFO)** order.*

Heap Space in Java

Heap space in Java is used for ***dynamic memory allocation for Java objects*** and JRE classes at the runtime. ***New objects*** are always created in heap space and the references to this objects are stored in stack memory.

- **Key Features of Java Heap Memory**
  - ✓ If heap space is full, Java throws ***java.lang.OutOfMemoryError***
  - ✓ Access to this memory is comparatively slower than stack memory
  - ✓ This memory, **isn't automatically deallocated**. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
  - ✓ Unlike stack, a heap isn't threadsafe and mory is relatively needs to be guarded by properly synchronizing the code

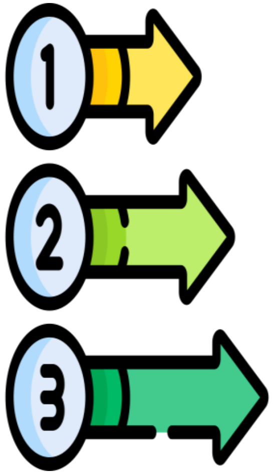
Heap Space vs Stack Memory

- Based on what we’ve learned so far, let’s analyze a simple Java code to assess how to manage memory here:

```
class Person {
      int id;
      String name;
      public Person(int id, String name) {
                this.id = id; this.name = name;
      }
}
public class PersonBuilder {
      private static Person buildPerson(int id, String name) {
                return new Person(id, name);
      }

     public static void main(String[] args) {
      int id = 23;
      String name = "John";
      Person person = buildPerson(id, name);
     }
}
```

Heap Space vs Stack Memory

- **Let’s analyze this step-by-step:**



✓ When we enter the *main()* method, a space in stack memory is created to store primitives and references of this method.

- Stack memory directly stores the primitive value of integer *id.*
- The reference variable *person* of type *Person* will also be created in stack memory, which will point to the actual object in the heap.

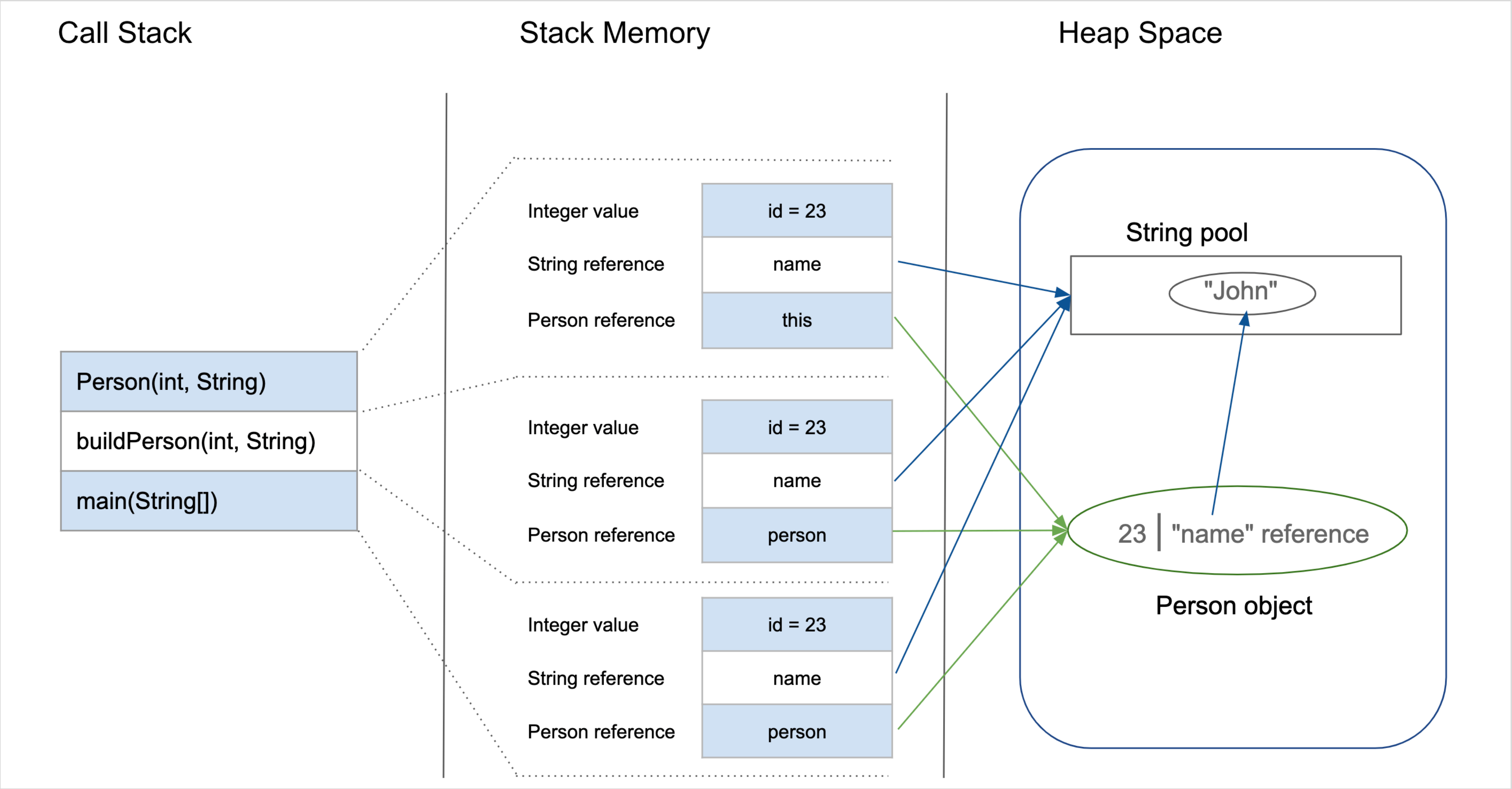
✓ The call to the parameterized constructor *Person(int, String)* from *main()* will allocate further memory on top of the previous stack. This will store:

- The **this object reference of the calling object in stack memory**
- The **primitive value id** in the stack memory
- The **reference variable of String argument name**, which will point to the actual string from string pool in heap memory

✓ The main method is further calling the *buildPerson()* static method, for which further allocation will take place in stack memory on top of the previous one. This will again store variables in the manner described above.

✓ However, heap memory will store all instance variables for the newly created object person of type Person.

Heap Space vs Stack Memory



Section 3

Passing Objects to Methods

Method Parameters

01

- Parameters (also called arguments) **is variable that declare** in the method definition.

02

- Parameters are **always classified** as "variables" not "fields".

03

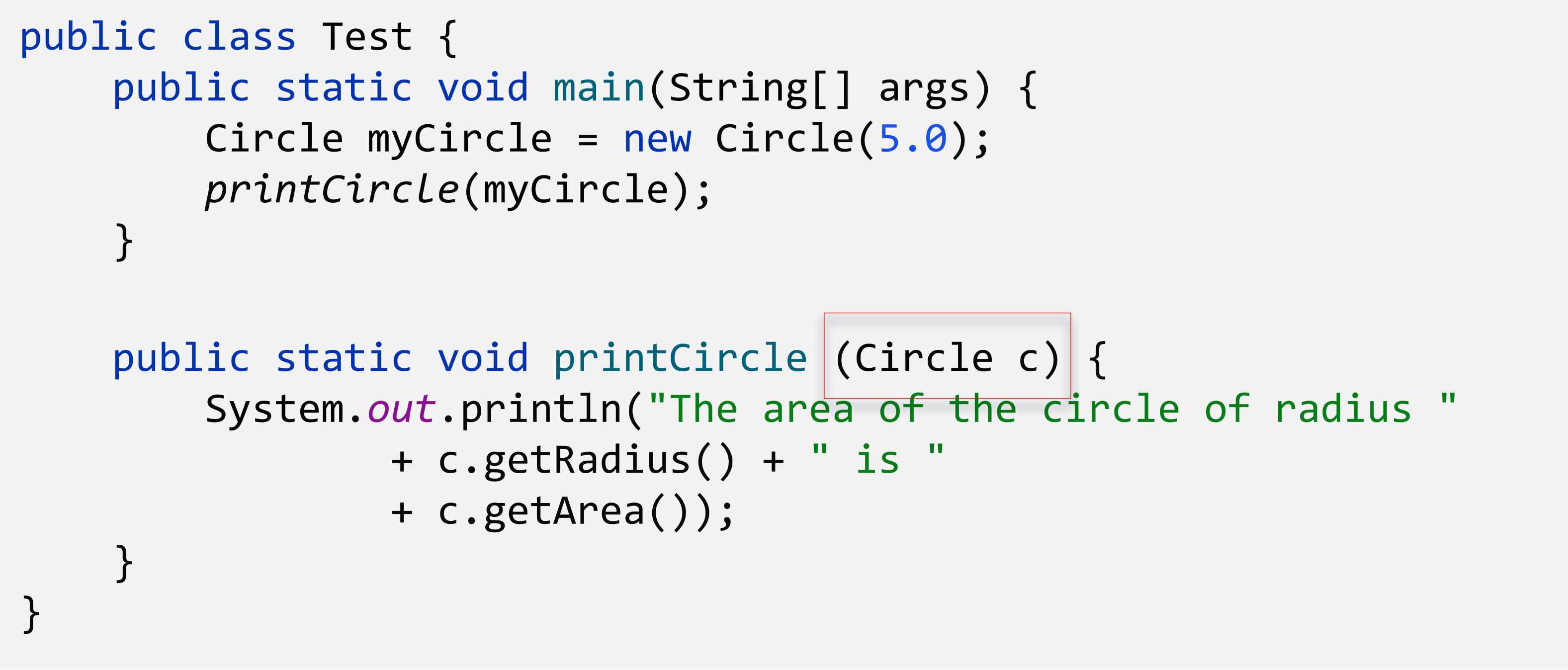
- **Two ways to pass arguments to methods**
  - Pass-by-value
  - Pass-by-reference

Value and Reference Parameters

- **Pass-by-value**
  - ✓ Copy of argument's value is passed to called method
  - ✓ In Java, every primitive is pass-by-value
- **Pass-by-reference**
  - ✓ Caller gives called method direct access to caller's data
  - ✓ Called method can manipulate this data
  - ✓ Improved performance over pass-by-value
  - ✓ In Java, every object/arrays is (are) pass-by-reference

Passing Objects to Methods

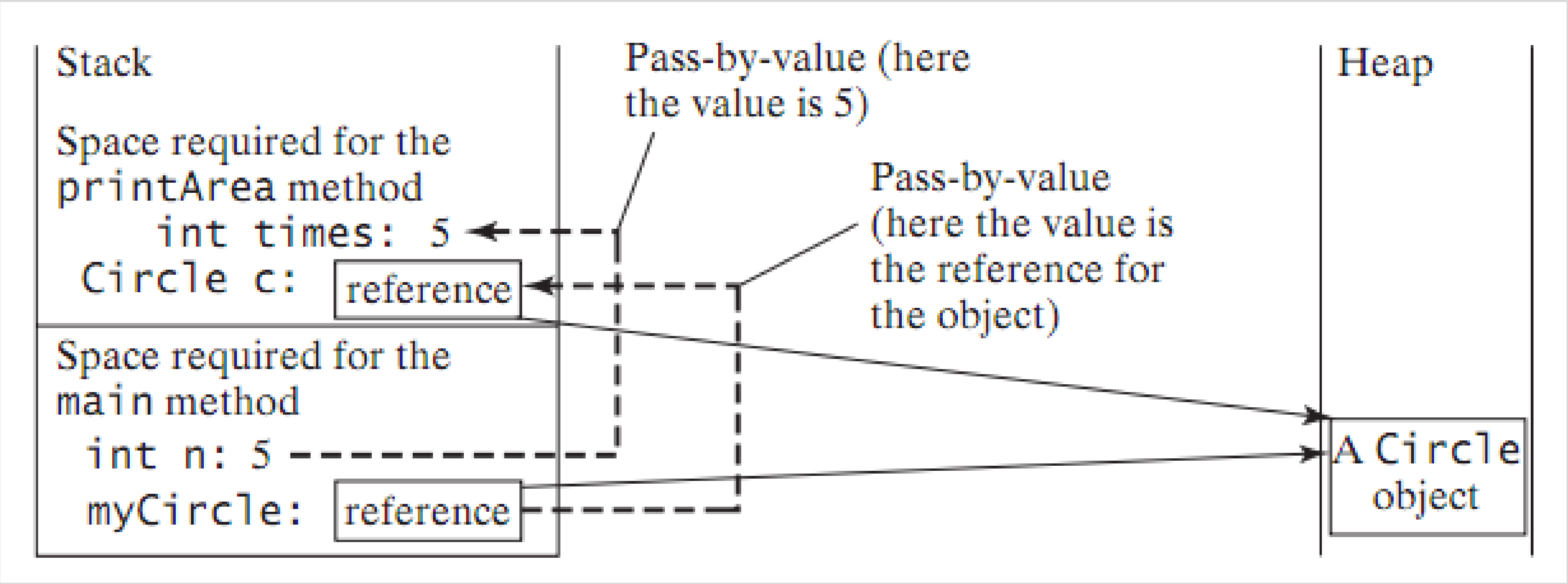
- You can pass objects to methods: Passing an object is actually passing the reference of the object.



```
public class Test {
    public static void main(String[] args) {
        Circle myCircle = new Circle(5.0);
        printCircle(myCircle);
    }

    public static void printCircle (Circle c) {
        System.out.println("The area of the circle of radius "
                + c.getRadius() + " is "
                + c.getArea());
    }
}
```

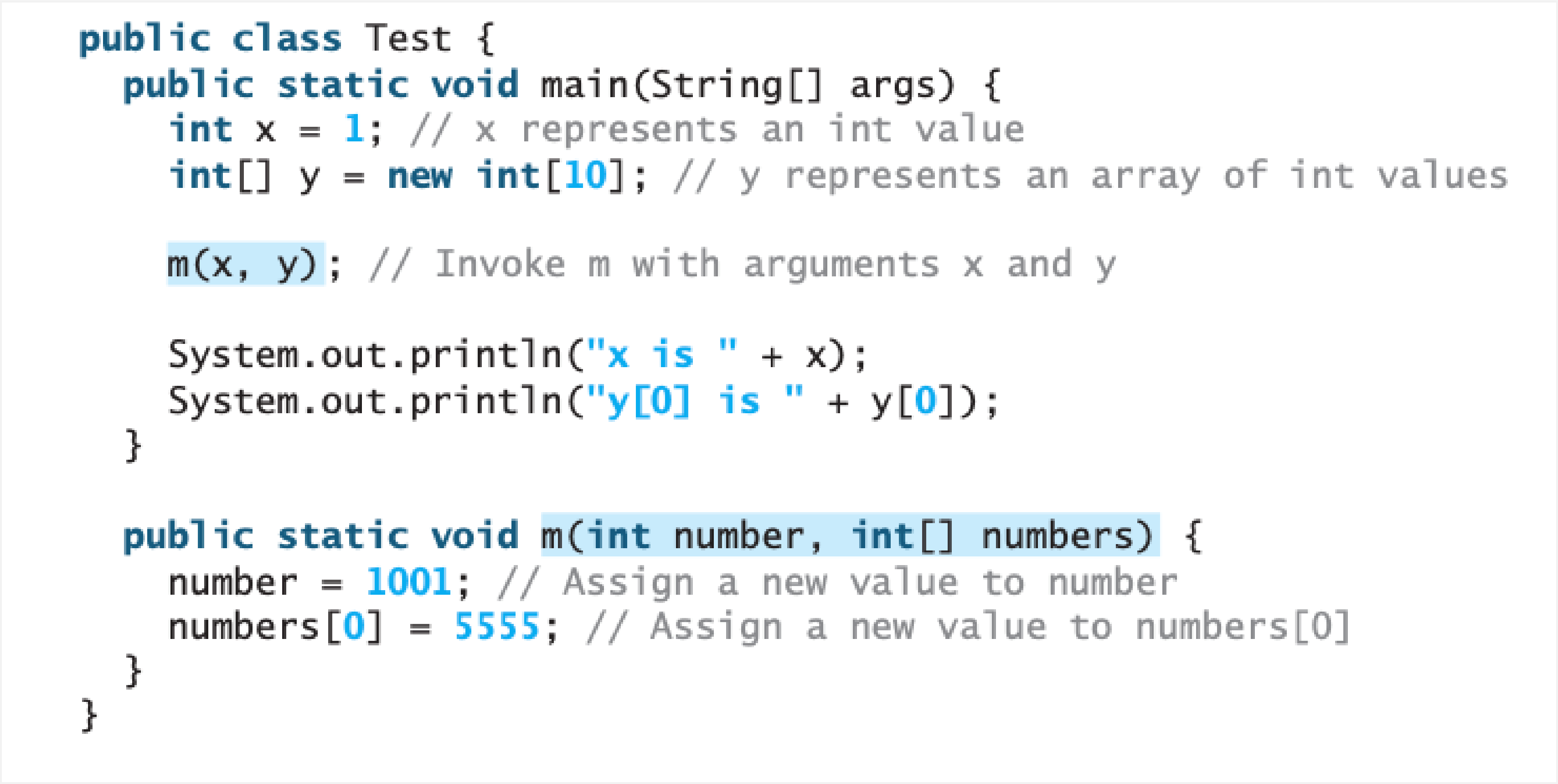
Passing Objects to Methods

- Passing objects passes their reference
  - ✓**c** and **myCircle** refer to same object
  - ✓Changes via **c** affect **myCircle** outside method


Stack
Space required for the printArea method
int times: 5
Circle c: reference
Space required for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

Passing Objects to Methods

- **Example:**



```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

- **What are output?**

Array of objects

- Create arrays of objects:

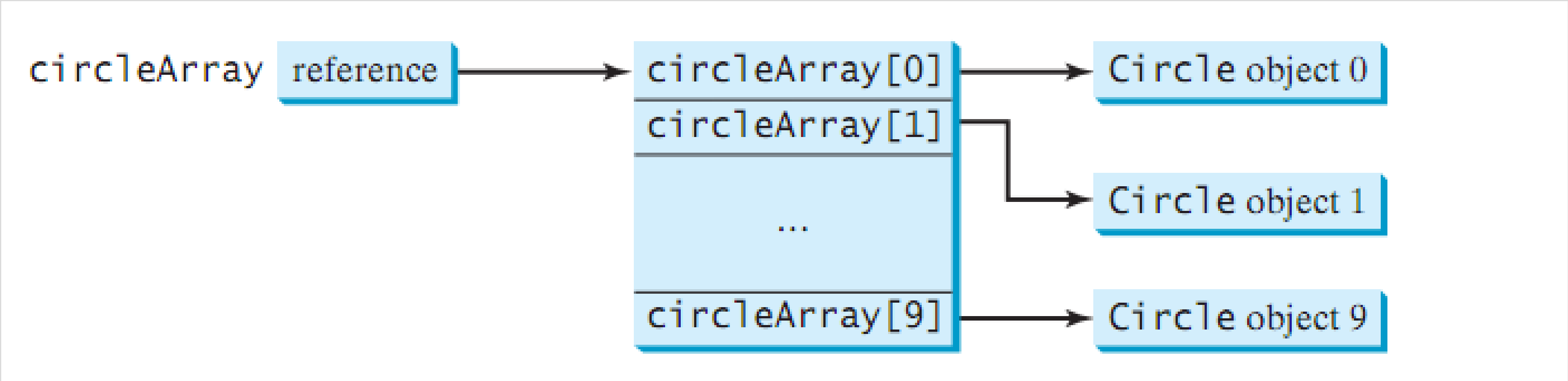
```
Circle[] circleArray = new Circle[10];
```

- Use for loop to initialize the **circleArray**:

```
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle();
}
```

Array of objects

- Array of objects is **array of references**
- Accessing involves two reference levels:
  - ✓**circleArray** references entire array
  - ✓**circleArray**[1] references a Circle



In an array of objects, an element of the array contains a reference to an object.

Array of objects

- **Example:**

```
public class TotalArea {
    public static void main(String[] args) {
        Circle[] circleArray;
        circleArray = createCircleArray();
        printCircleArray(circleArray);
    }

    /** Create an array of Circle objects */
    public static Circle[] createCircleArray() {
        Circle[] circleArray = new Circle[5];

        for (int i = 0; i < circleArray.length; i++) {
            circleArray[i] = new Circle(Math.random() * 100);
        }
        return circleArray;
    }
```

```
    public static void printCircleArray(Circle[] circleArray) {
        System.out.printf("%-30s%-15s\n", "Radius", "Area");

        for (int i = 0; i < circleArray.length; i++) {
            System.out.printf("%f-30f%-15f\n",
                    circleArray[i].getRadius(), circleArray[i].getArea());
        }
        System.out.println("————————————————————————————————————");
        // Compute and display the result
        System.out.printf("%-30s%-15f\n",
                "The total area of circles is",sum(circleArray));
    }

    public static double sum(Circle[] circleArray) {
        double sum = 0; // Initialize sum
        for (int i = 0; i < circleArray.length; i++)
            sum += circleArray[i].getArea();
        return sum;
    }
}
```

THANK YOU!