Object Oriented Programming

By Watcharin Sarachai

Topic

- Class Methods in Java
- new operator in Java
- Constructor Methods

Class Methods in Java

Class Methods in Java

- The Java Methods are declared within a **class**, and that they are used to perform certain actions:
  - Create a method named myMethod() in MyClass:

```java
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

Class Methods in Java

- myMethod() prints a text (*the action*), when it is called. To call a method, write the method's name followed by two parentheses `()` and a semicolon `;`
    - Inside main, call myMethod():

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Static vs. Non-Static

- We will often see Java programs that have either static or public attributes and methods.
- In the example above, we created a static method, which means that it can be accessed without creating an **object** of the **class**, unlike public, which can only be accessed by objects:
  - An example to demonstrate the differences between static and public methods:

```java
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    myPublicMethod(); // This would compile an error
  }
}
```

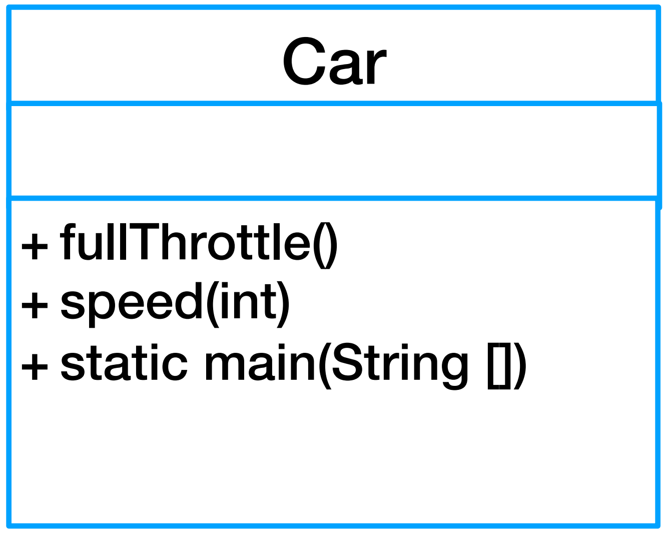
Static vs. Non-Static

```java
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Access Methods With an Object

- Create a Car object named myCar. Call the fullThrottle() and speed() methods on the myCar object, and run the program:

```java
package camt.ch02;
// Create a Car class
public class Car {
  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }
  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    // TODO: Create a myCar object

    // TODO: Call the fullThrottle() method

    // TODO: Call the speed(200) method

  }
}
```



Example explained

1. We created a custom Car class with the class keyword.

2. We created the fullThrottle() and speed() methods in the Car class.

3. The fullThrottle() method and the speed() method will print out some text, when they are called.

4. The speed() method accepts an int parameter called maxSpeed - we will use this in 8).

5. In order to use the Car class and its methods, we need to create an object of the Car Class.

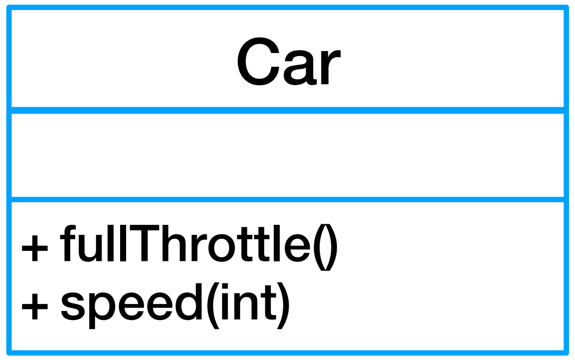
6. Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).

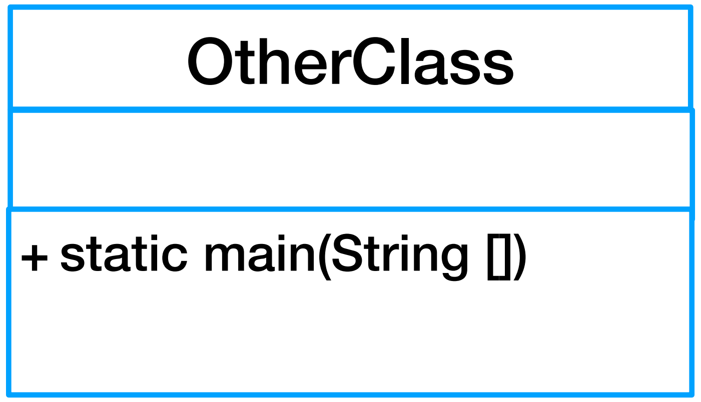
7. By using the new keyword we created a Car object with the name myCar.

8. Then, we call the fullThrottle() and speed() methods on the myCar object, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of **200** inside the speed() method.

Using Multiple Classes

- It is a good practice to create an object of a class and access it in another class.
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:
  - Car.java
  - OtherClass.java





```java
public class Car {

  public void fullThrottle() {

    System.out.println("The car is going as fast as it can!");

  }


  public void speed(int maxSpeed) {

    System.out.println("Max speed is: " + maxSpeed);

  }

}
```

```java
class OtherClass {

  public static void main(String[] args) {

    Car myCar = new Car();     // Create a myCar object

    myCar.fullThrottle();      // Call the fullThrottle() method

    myCar.speed(100);          // Call the speed() method

  }

}
```

new operator in Java

new Operator in Java

- When you are declaring a class in java, you are just creating a new data type.
- A class provides the blueprint for objects. You can create an object from a class.
- However obtaining objects of a class is a two-step process :

  1. Declaration

  2.Instantiation and Initialization

new Operator in Java

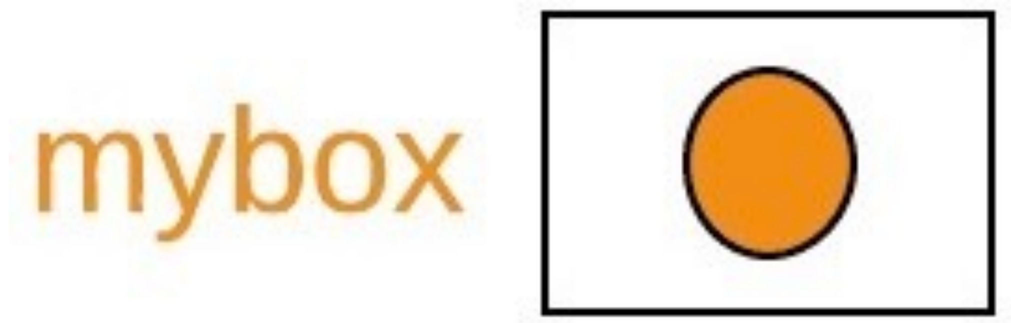
1. Declaration: First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Below is general syntax of declaration with an example :

Syntax :
```
class-name var-name;
```

Example :
```
// declare reference to an object of class Box
Box mybox;
```

new Operator in Java

1. Declaration: A variable in this state, which currently references no object, can be illustrated as follows (*the variable name, **mybox**, plus a reference pointing to nothing*):



new Operator in Java

1. Instantiation and Initialization:

- Second, we must acquire an actual, physical copy of the object and assign it to that variable.

- We can do this using the new operator.
  The new operator instantiates a class by dynamically allocating (*i.e, allocation at run time*) memory for a new object and returning a reference to that memory.

- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

new Operator in Java

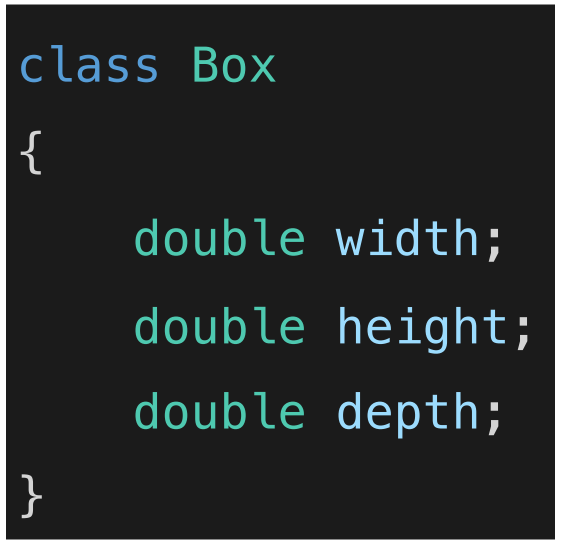
1.Instantiation and Initialization:

- The new operator is also followed by a call to a **class constructor**, which initializes the **new object**. A **constructor** defines what occurs when an object of a class is created.

- Constructors are an important part of all classes and have many significant attributes. In below example we will use the default constructor. Below is general syntax of instantiation and initialization with an example :

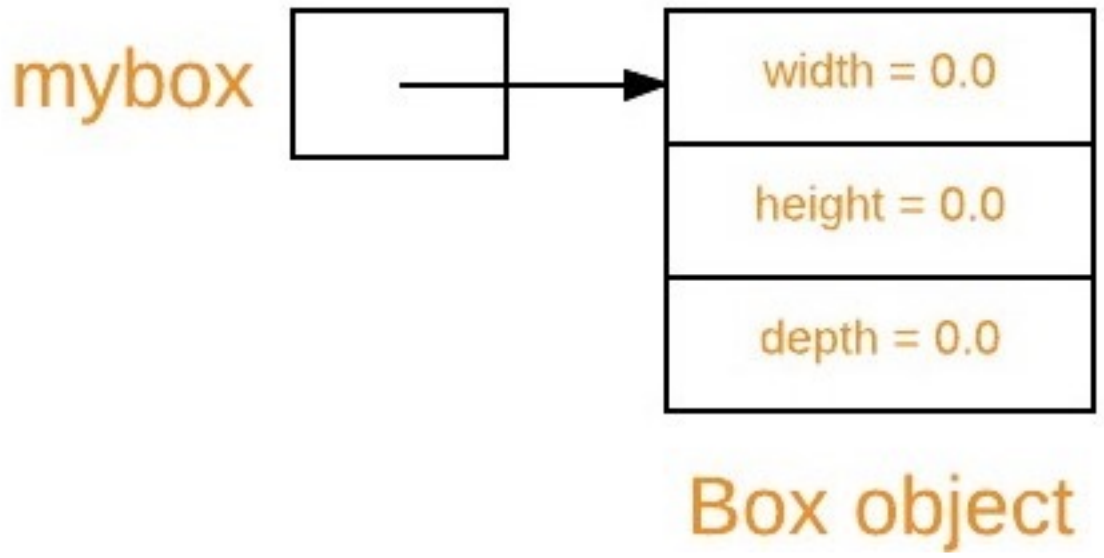
```
Syntax :
var-name = new class-name();

Example :
// instantiation via new operator and
// initialization via default constructor of class Box
mybox = new Box();
```

Example: Class Box Prototype.



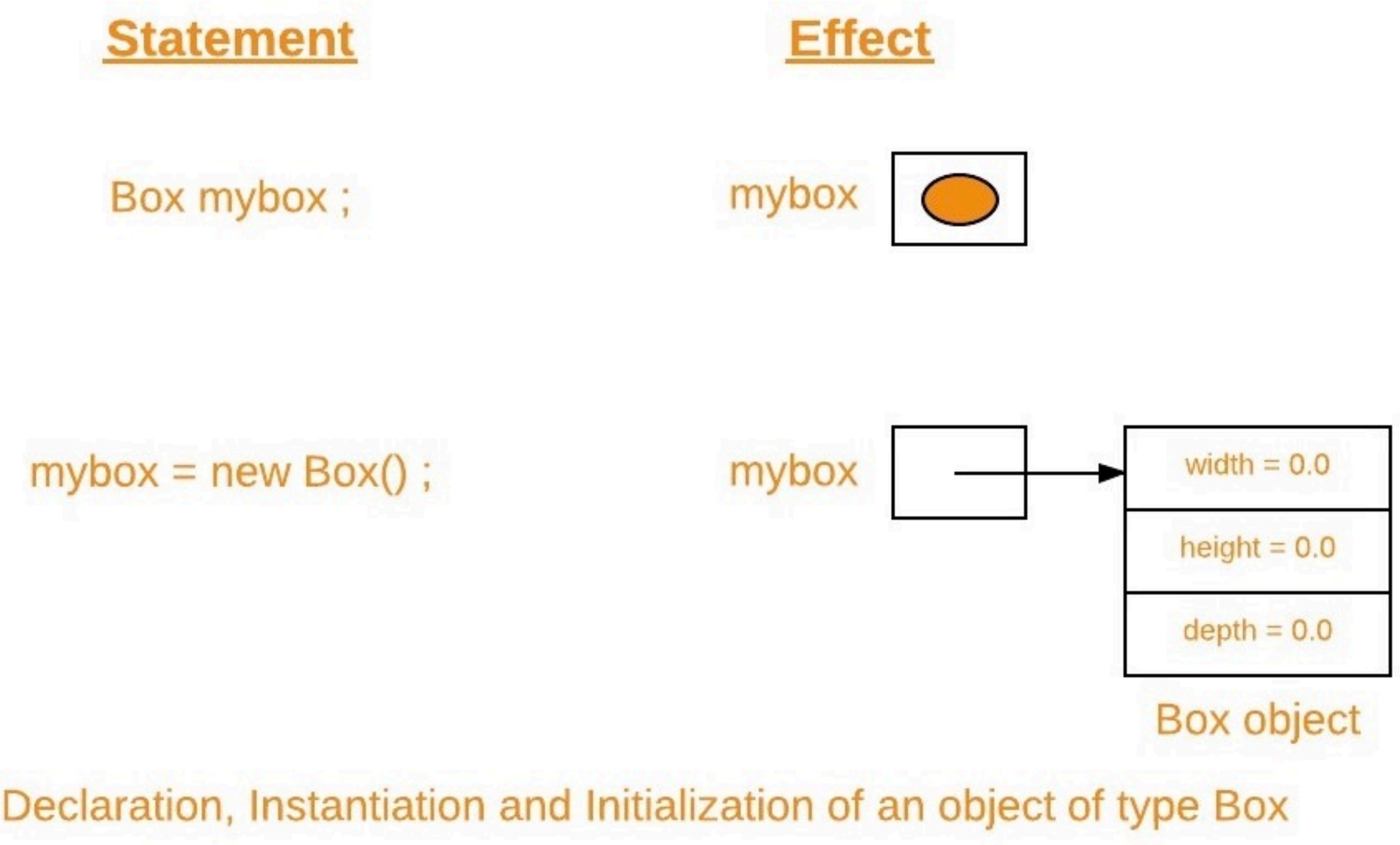
```
class Box
{
    double width;
    double height;
    double depth;
}
```

- A variable after second step, currently refer to a class object, can be illustrated as follows (*the variable name, mybox, plus a reference pointing to Box object*):



Example: Class Box Prototype.

- Hence declaration of a class variable, instantiation of a class and initialization of an object of class can be together illustrated as follows :



Example: Class Box Prototype.

- **Important points :**

1. From the previous slide, two statements can be rewritten as one statement.

```java
Box mybox = new Box();
```

2. The reference returned by the new operator does not have to be assigned to a class variable. It can also be used directly in an expression. For example:

```java
double height = new Box().height;
```

3. Since arrays are object in java, hence while instantiating arrays, we use new operator. For example:

```java
int arr[] = new int[5];
```

Example: Class Box Prototype.

- **Important points :**

  4. At this point, why we do not need to use new operator for primitives data types. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "**normal**" variables.

  5. The phrase "**instantiating a class**" means the same thing as "**creating an object.**" When we create an object, we are creating an "**instance**" of a class, therefore "**instantiating**" a class.

Assigning object reference variables

- When you assign one object reference variable to another object reference variable, we are not creating a copy of the object, we are only **making a copy of the reference.**

- Let us understand this with an example.

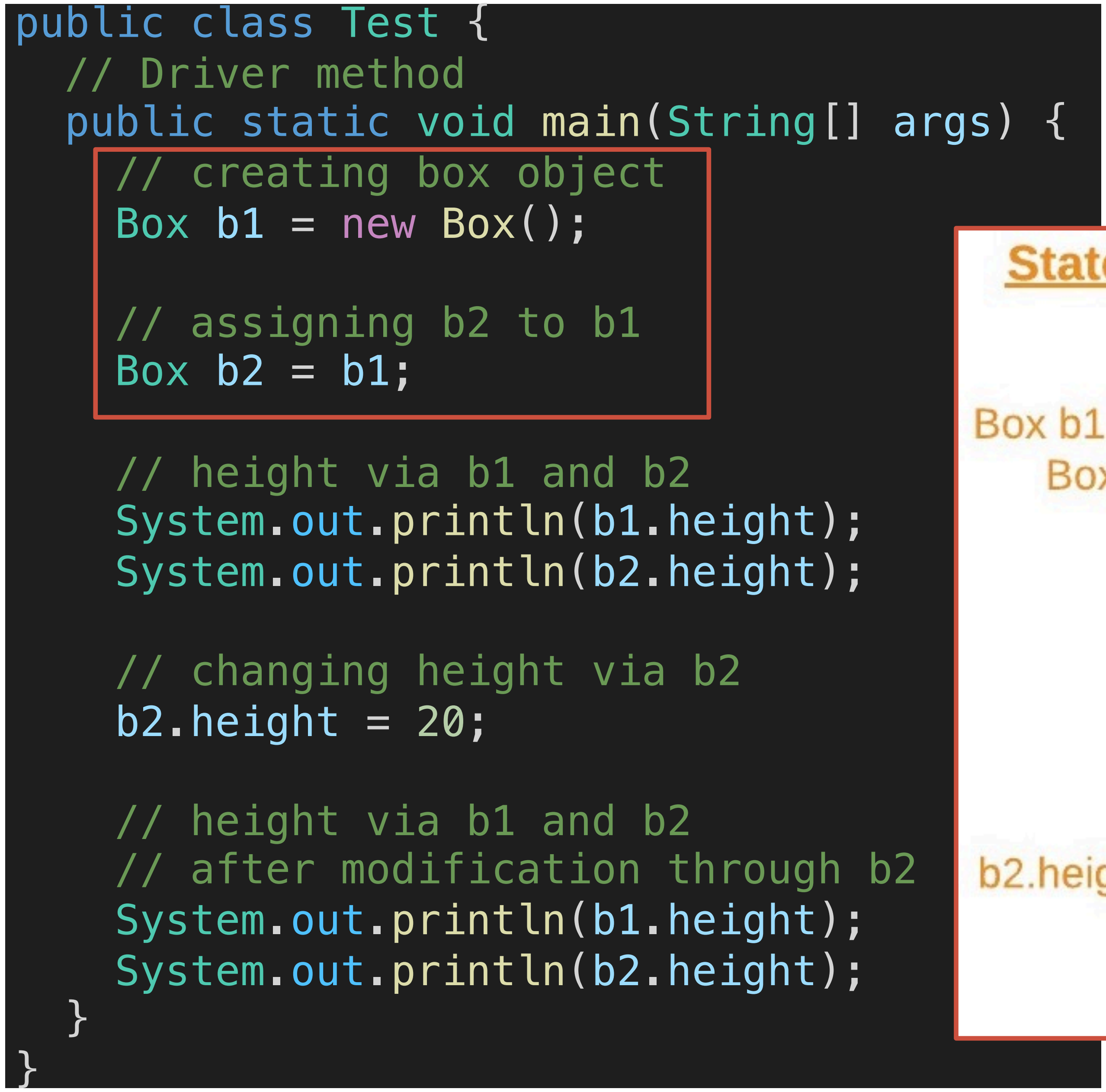
```
// Box class
class Box
{
    double width;
    double height;
    double depth;
}
```

```
Output :

0.0
0.0
20.0
20.0
```

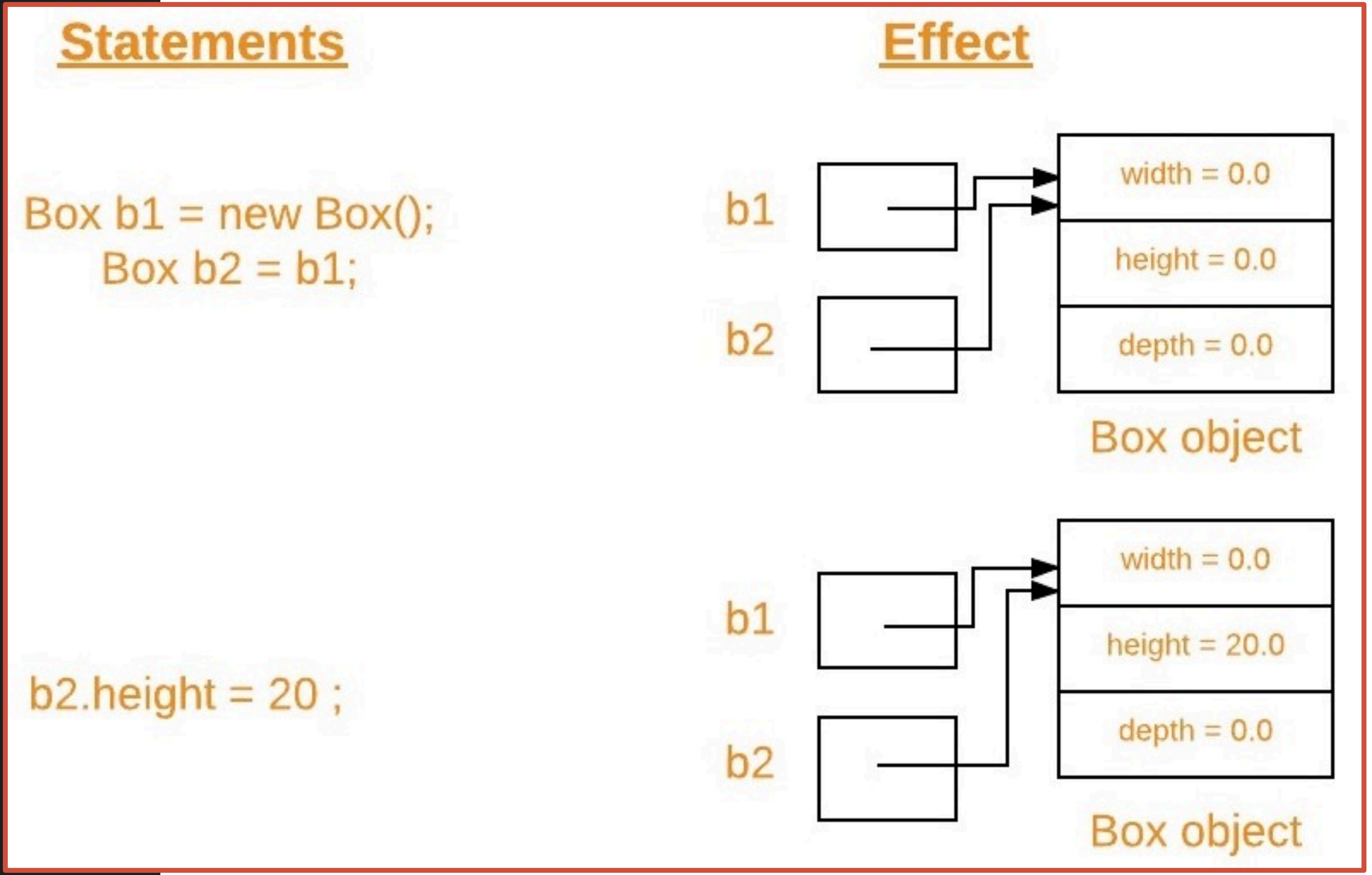
```
public class Test {
  // Driver method
  public static void main(String[] args) {
    // creating box object
    Box b1 = new Box();

    // assigning b2 to b1
    Box b2 = b1;

    // height via b1 and b2
    System.out.println(b1.height);
    System.out.println(b2.height);

    // changing height via b2
    b2.height = 20;

    // height via b1 and b2
    // after modification through b2
    System.out.println(b1.height);
    System.out.println(b2.height);
  }
}
```

Assigning object reference variables

- First let us understand what the following fragment does in above program.



```
public class Test {
  // Driver method
  public static void main(String[] args) {
    // creating box object
    Box b1 = new Box();

    // assigning b2 to b1
    Box b2 = b1;

    // height via b1 and b2
    System.out.println(b1.height);
    System.out.println(b2.height);

    // changing height via b2
    b2.height = 20;

    // height via b1 and b2
    // after modification through b2
    System.out.println(b1.height);
    System.out.println(b2.height);
  }
}
```



Assigning object reference variables

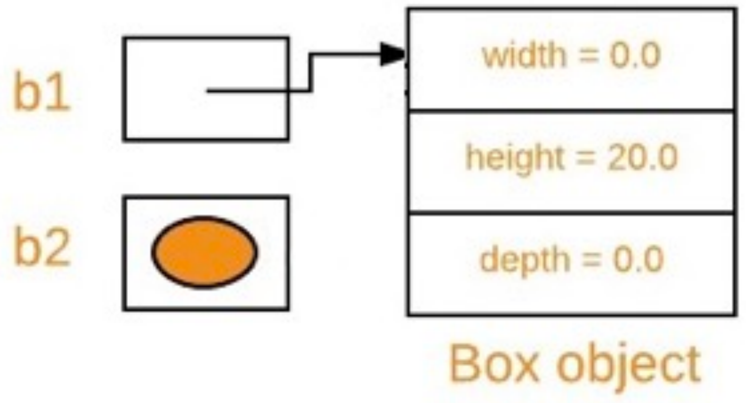
Note : Although b1 and b2 both refer to the same object, they are not linked in any other way.

For example: a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.

For example :

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to *null*, but **b2** still points to the original object.



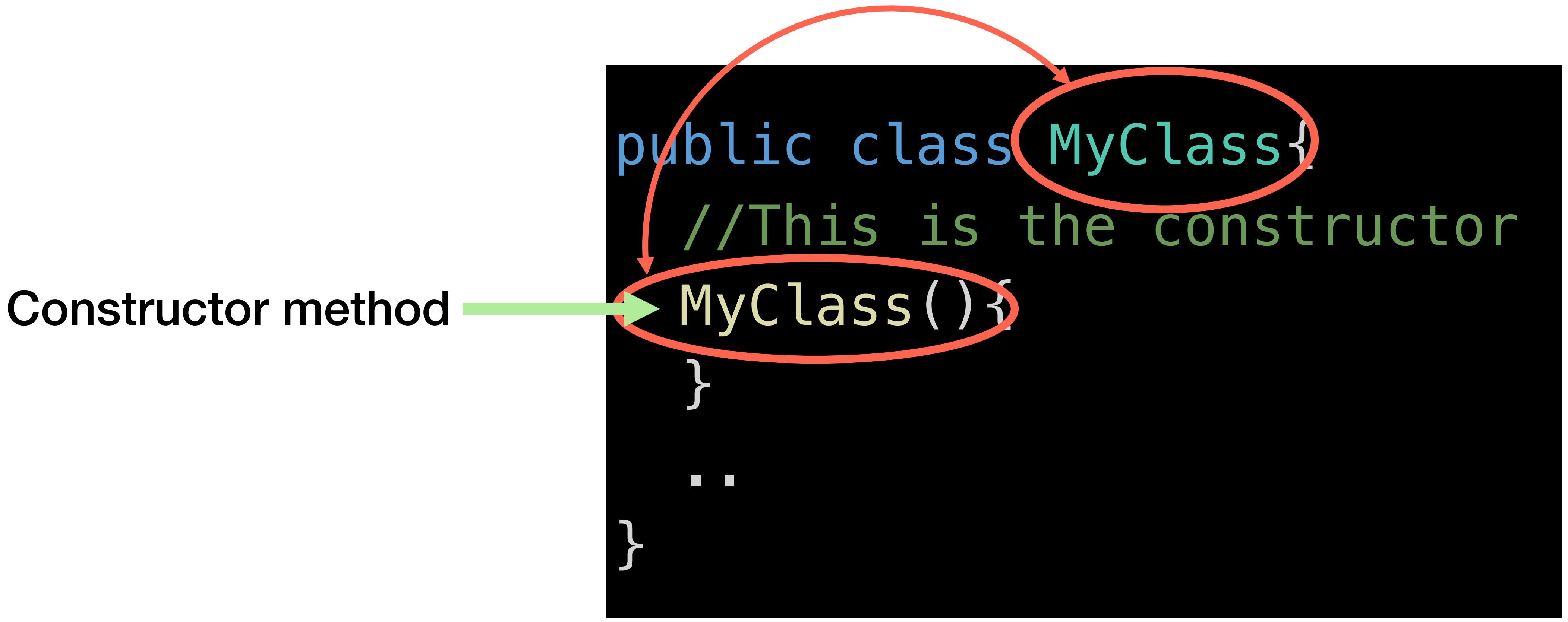
Constructor Methods

Java Constructors

- A constructor in Java is a **special method** that is used to **initialize objects**.
- A constructor similar to an **instance method** in java but it's not a method as it doesn't have a return type.
- In short constructor and method are different.
- People often refer constructor as special type of method in Java.

Constructor Methods

- Constructor has **same name as the class** and looks like this in a java code.



```
public class MyClass{
  //This is the constructor
  MyClass(){
  }
  ..
}
```

Note: that the constructor name matches with the class name and it doesn't have a return type.

How does a constructor work

- To understand the working of constructor, lets take an example. lets say we have a class MyClass. When we create the object of MyClass like this:

```
MyClass obj = new MyClass();
```

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

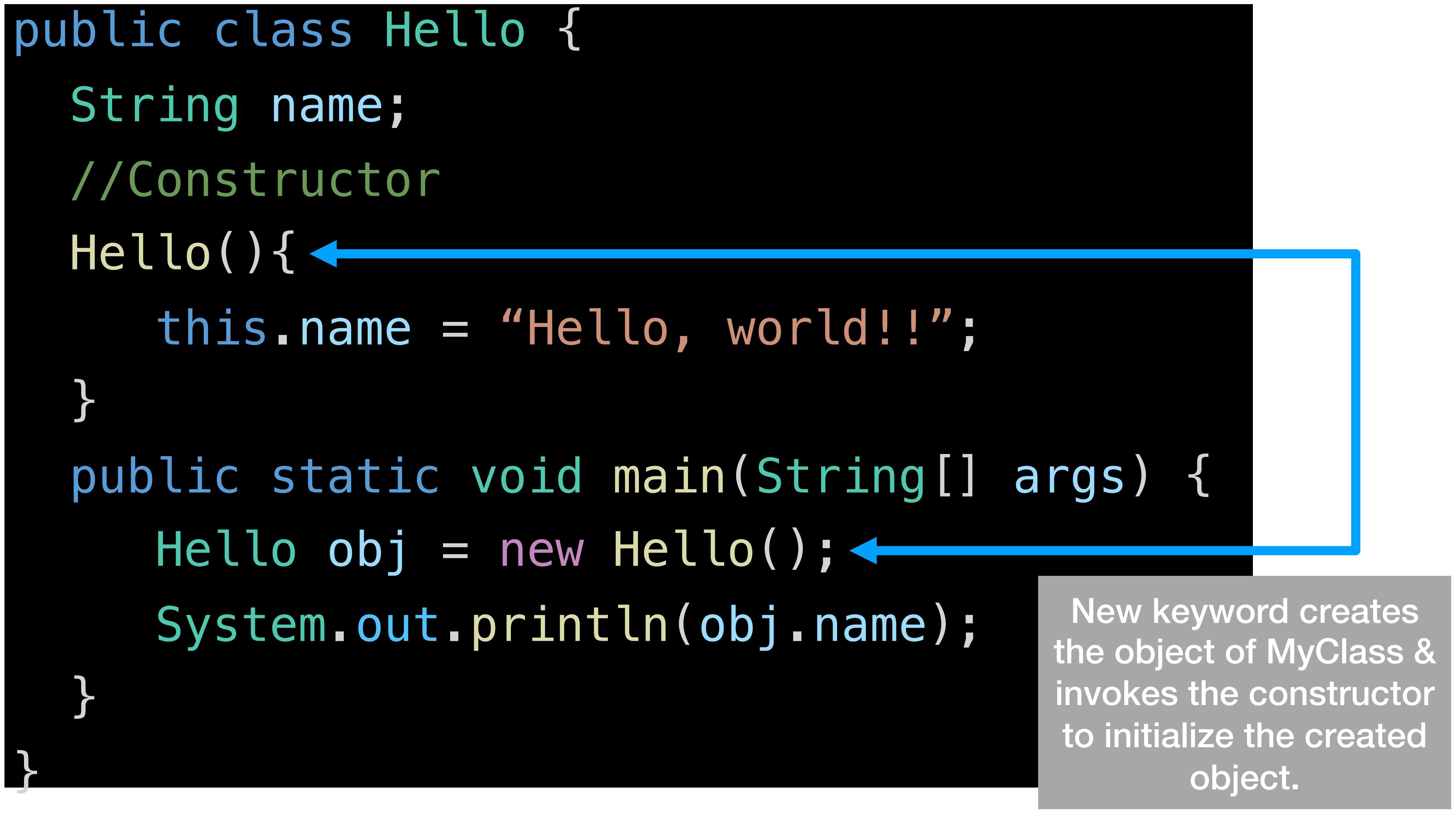
A simple constructor program in java

- Here we have created an object obj of class Hello and then we displayed the instance variable name of the object.

```
public class Hello {
  String name;
  //Constructor
  Hello(){
     this.name = “Hello, world!!”;
  }
  public static void main(String[] args) {
     Hello obj = new Hello();
     System.out.println(obj.name);
  }
}
```

```
Output:

Hello, world!!
```

A simple constructor program in java



```java
public class Hello {
  String name;
  //Constructor
  Hello(){
     this.name = “Hello, world!!”;
  }
  public static void main(String[] args) {
     Hello obj = new Hello();
     System.out.println(obj.name);
  }
}
```

New keyword creates the object of MyClass & invokes the constructor to initialize the created object.

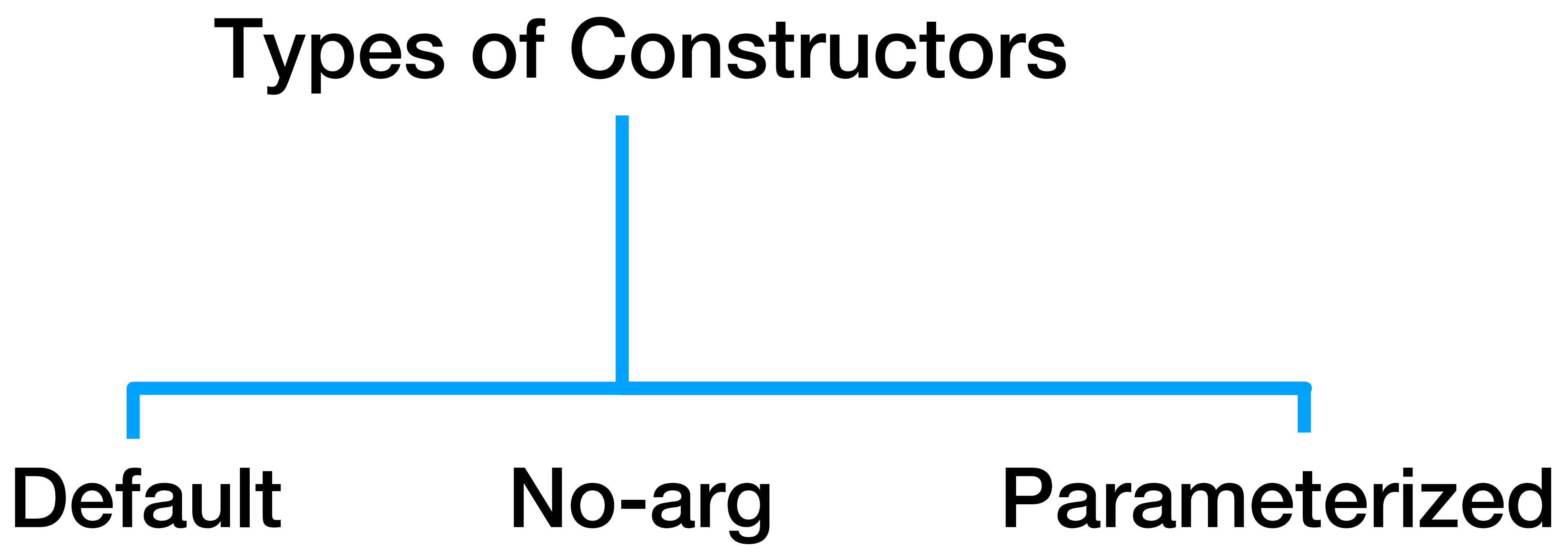
Do It by Yourself

- The class **MyClass** in package **camt.day2.constructor**
  - Complete the code depend on each instruction.

```java
public class MyClass {
  // TODO: Create a class attribute name "int x;"

  public MyClass() {
    // TODO: Set the initial value for the class attribute x with 5

  }

  public static void main(String [] args) {
    // TODO: Create an object of class MyClass (This will call the constructor)

    // TODO: Print the value of x

  }
}
```

Types of Constructors

- There are three types of constructors: **Default**, **No-arg** constructor and **Parameterized**.



Default constructor

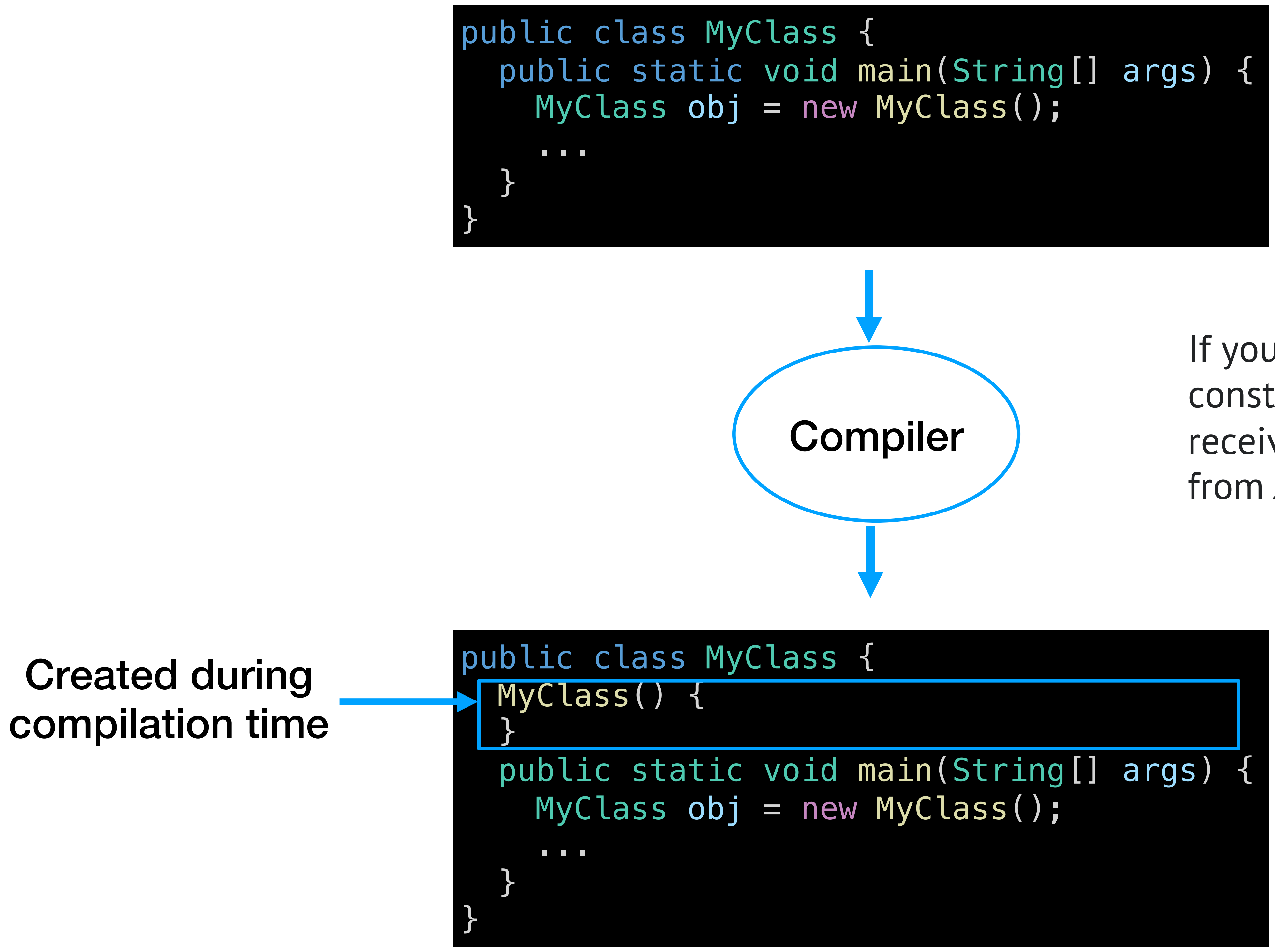
- If you do not implement any constructor in your class, Java compiler inserts a **default constructor** into your code.

- This constructor is known as **default constructor**.

- You would not find it in your source code (*the java file*) as it would be inserted into the code during compilation and exists in .class file.

- This process is shown in the diagram in next slide:

Default constructor



```
public class MyClass {
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    ...
  }
}
```

```
public class MyClass {
  MyClass() {
  }
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    ...
  }
}
```

If you implement any constructor then you no longer receive a default constructor from Java compiler.

no-arg constructor

- Constructor with no arguments is known as **no-arg constructor**.
- The signature is same as default constructor, **however body can have any code** unlike default constructor where the body of the constructor is empty.

```java
class Demo
{
     public Demo()
     {
         System.out.println("This is a no argument constructor");
     }
     public static void main(String args[]) {
       new Demo();
     }
}
```

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

Parameterized constructor

- Constructor with arguments (or you can say parameters) is known as **Parameterized constructor**.

- In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2.

```java
package camt.day2.constructor;

public class Employee {

  int empId;
  String empName;

  // parameterized constructor with two parameters
  Employee(int id, String name) {
    this.empId = id;
    this.empName = name;
  }

  void info() {
    System.out.println("Id: " + empId + " Name: " + empName);
  }

  public static void main(String args[]) {
    Employee obj1 = new Employee(10245, "Chaitanya");
    Employee obj2 = new Employee(92232, "Negan");
    obj1.info();
    obj2.info();
  }
}
```

Example2: parameterized constructor

- In this example, we have two constructors, a default constructor and a parameterized constructor.
- When we do not pass any parameter while creating the object using **new** keyword then **default constructor is invoked**, however when you pass a parameter then parameterized constructor that matches with the passed **parameters list gets invoked**.

```java
class Example2 {
  private int var;

  // default constructor
  public Example2() {
    this.var = 10;
  }

  // parameterized constructor
  public Example2(int num) {
    this.var = num;
  }

  public int getValue() {
    return var;
  }

  public static void main(String args[]) {
    Example2 obj = new Example2();
    Example2 obj2 = new Example2(100);
    System.out.println("var is: " + obj.getValue());
    System.out.println("var is: " + obj2.getValue());
  }
}
```

```
Output:

var is: 10
var is: 100
```

Parameterized constructor

- What if you implement only parameterized constructor in class.

```java
class Example3 {
  private int var;

  public Example3(int num) {
    var = num;
  }

  public int getValue() {
    return var;
  }

  public static void main(String args[]) {
    Example3 myobj = new Example3();  // Error
    System.out.println("value of var is: " + myobj.getValue());
  }
}
```

- Output: It will throw a compilation error.

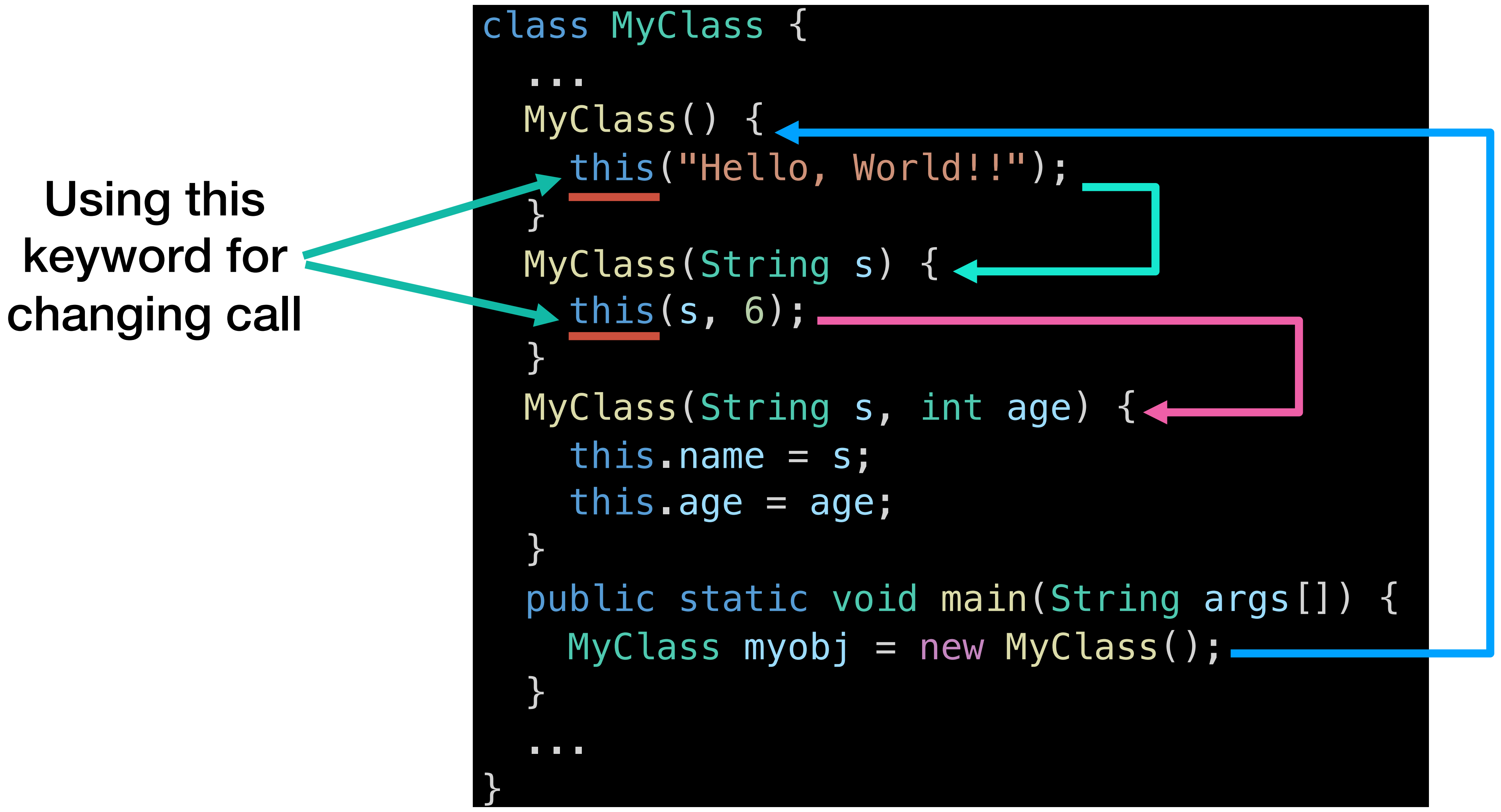
- The reason is, the statement Example3 myobj = new Example3() is invoking a default constructor which we don’t have in our program.

Parameterized constructor

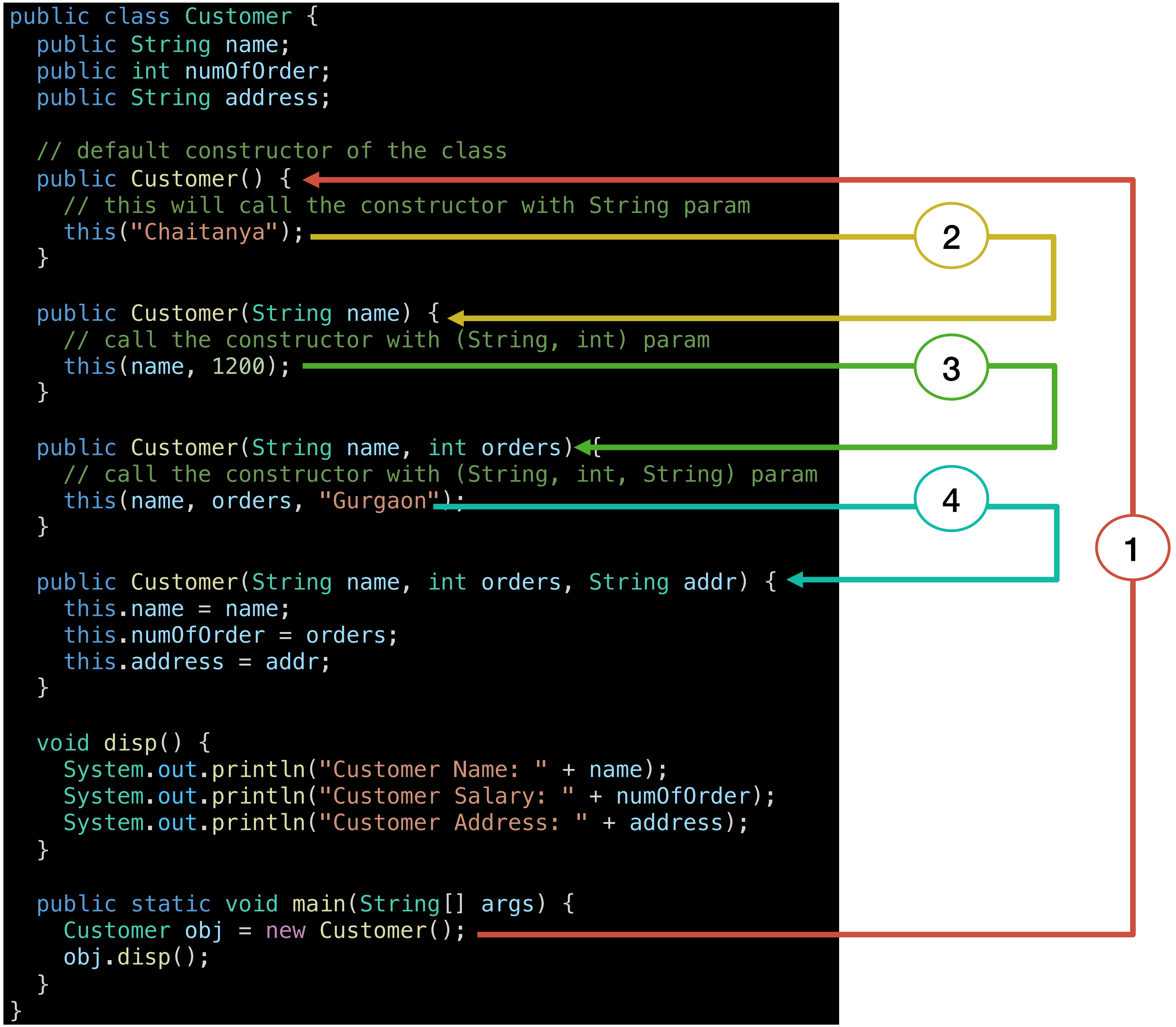
- when you don’t implement any constructor in your class, compiler inserts the default constructor into your code.
- However when you implement any constructor, then you don’t receive the default constructor by compiler into your code.
- If we remove the parameterized constructor from the above code then the program would run fine, because then compiler would insert the default constructor into your code.

Constructor Chaining

- When A constructor calls another constructor of same class then this is called **constructor chaining**.



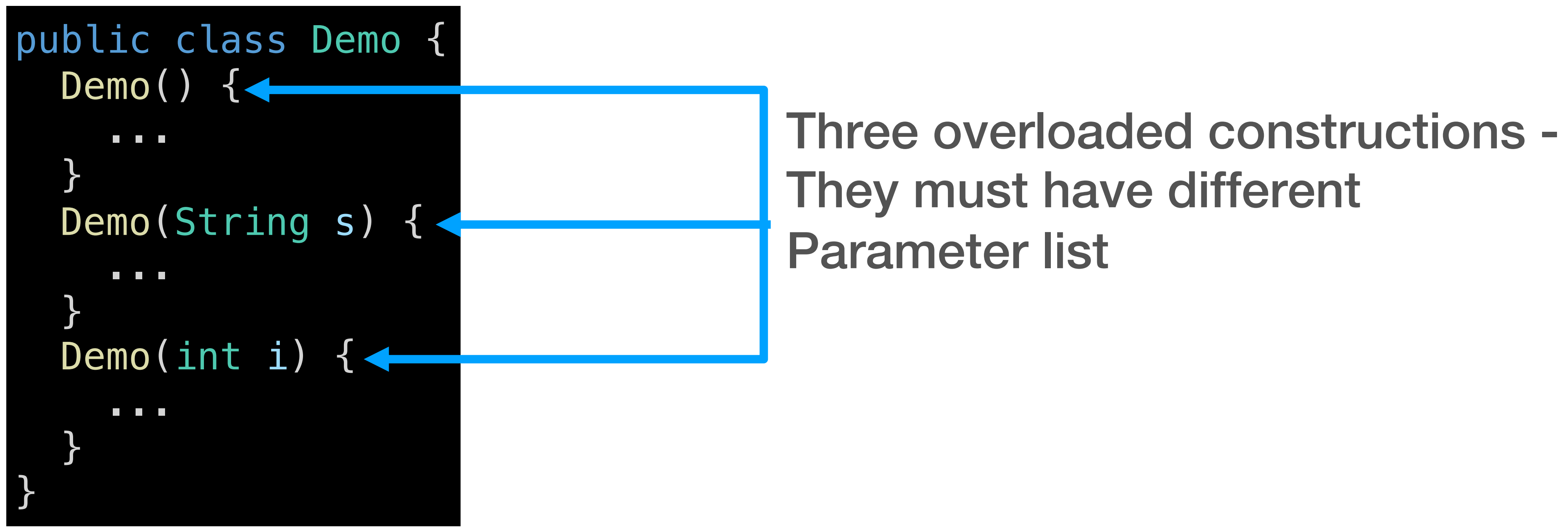
```
class MyClass {
  ...
  MyClass() {
    this("Hello, World!!");
  }
  MyClass(String s) {
    this(s, 6);
  }
  MyClass(String s, int age) {
    this.name = s;
    this.age = age;
  }
  public static void main(String args[]) {
    MyClass myobj = new MyClass();
  }
  ...
}
```



```
public class Customer {
  public String name;
  public int numOfOrder;
  public String address;

  // default constructor of the class
  public Customer() {
    // this will call the constructor with String param
    this("Chaitanya");
  }

  public Customer(String name) {
    // call the constructor with (String, int) param
    this(name, 1200);
  }

  public Customer(String name, int orders) {
    // call the constructor with (String, int, String) param
    this(name, orders, "Gurgaon");
  }

  public Customer(String name, int orders, String addr) {
    this.name = name;
    this.numOfOrder = orders;
    this.address = addr;
  }

  void disp() {
    System.out.println("Customer Name: " + name);
    System.out.println("Customer Salary: " + numOfOrder);
    System.out.println("Customer Address: " + address);
  }

  public static void main(String[] args) {
    Customer obj = new Customer();
    obj.disp();
  }
}
```

Constructor Overloading

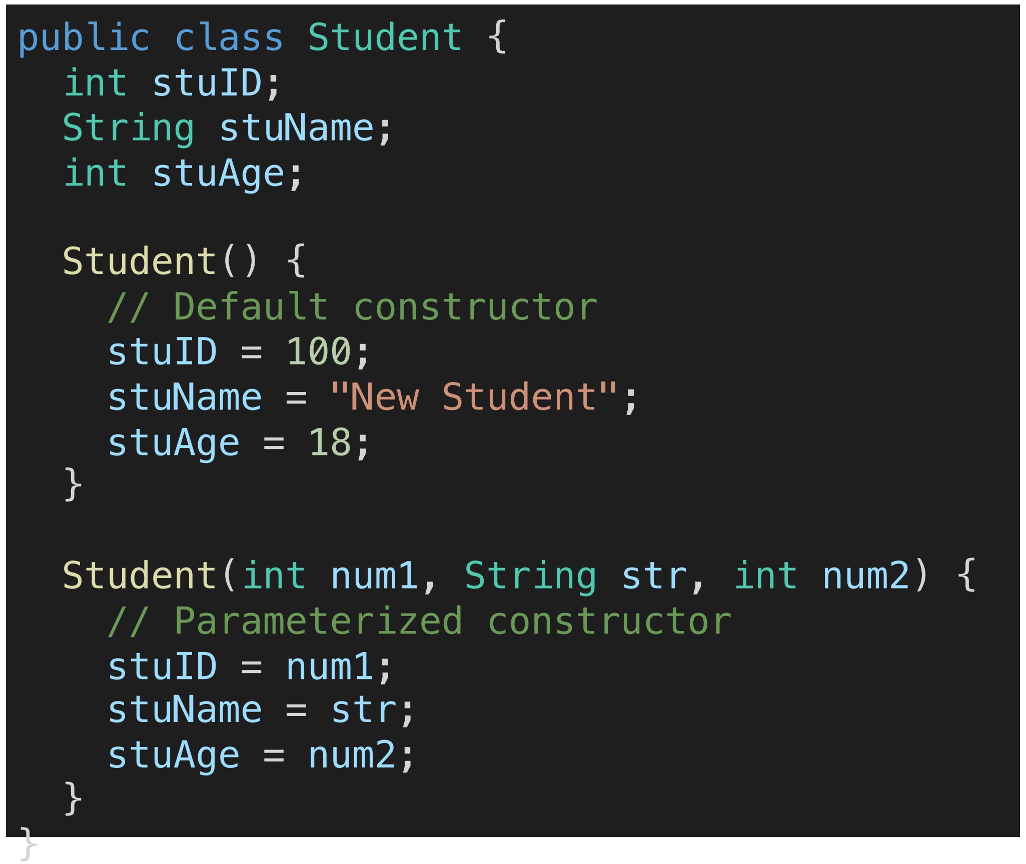
- Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.



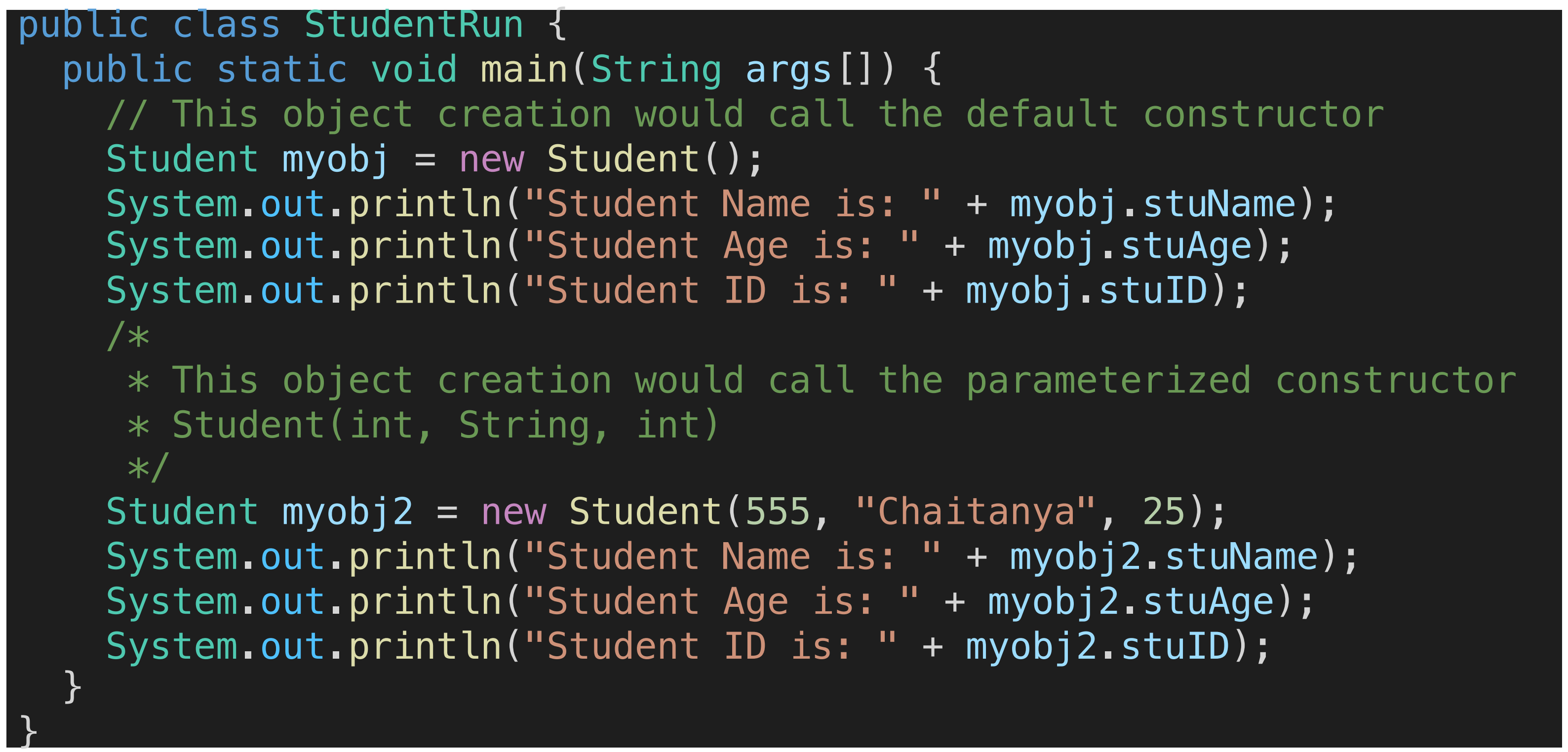
```
public class Demo {
  Demo() {
    ...
  }
  Demo(String s) {
    ...
  }
  Demo(int i) {
    ...
  }
}
```

Constructor Overloading Example

- Here we are creating two objects of class **StudentData**. One is with **default constructor** and another one using **parameterized constructor**.
- Both the constructors have different initialization code, similarly you can create any number of constructors with different-2 initialization codes for different-2 purposes.



```java
public class Student {
  int stuID;
  String stuName;
  int stuAge;

  Student() {
    // Default constructor
    stuID = 100;
    stuName = "New Student";
    stuAge = 18;
  }

  Student(int num1, String str, int num2) {
    // Parameterized constructor
    stuID = num1;
    stuName = str;
    stuAge = num2;
  }
}
```



```java
public class StudentRun {
  public static void main(String args[]) {
    // This object creation would call the default constructor
    Student myobj = new Student();
    System.out.println("Student Name is: " + myobj.stuName);
    System.out.println("Student Age is: " + myobj.stuAge);
    System.out.println("Student ID is: " + myobj.stuID);
    /*
     * This object creation would call the parameterized constructor
     * Student(int, String, int)
     */
    Student myobj2 = new Student(555, "Chaitanya", 25);
    System.out.println("Student Name is: " + myobj2.stuName);
    System.out.println("Student Age is: " + myobj2.stuAge);
    System.out.println("Student ID is: " + myobj2.stuID);
  }
}
```

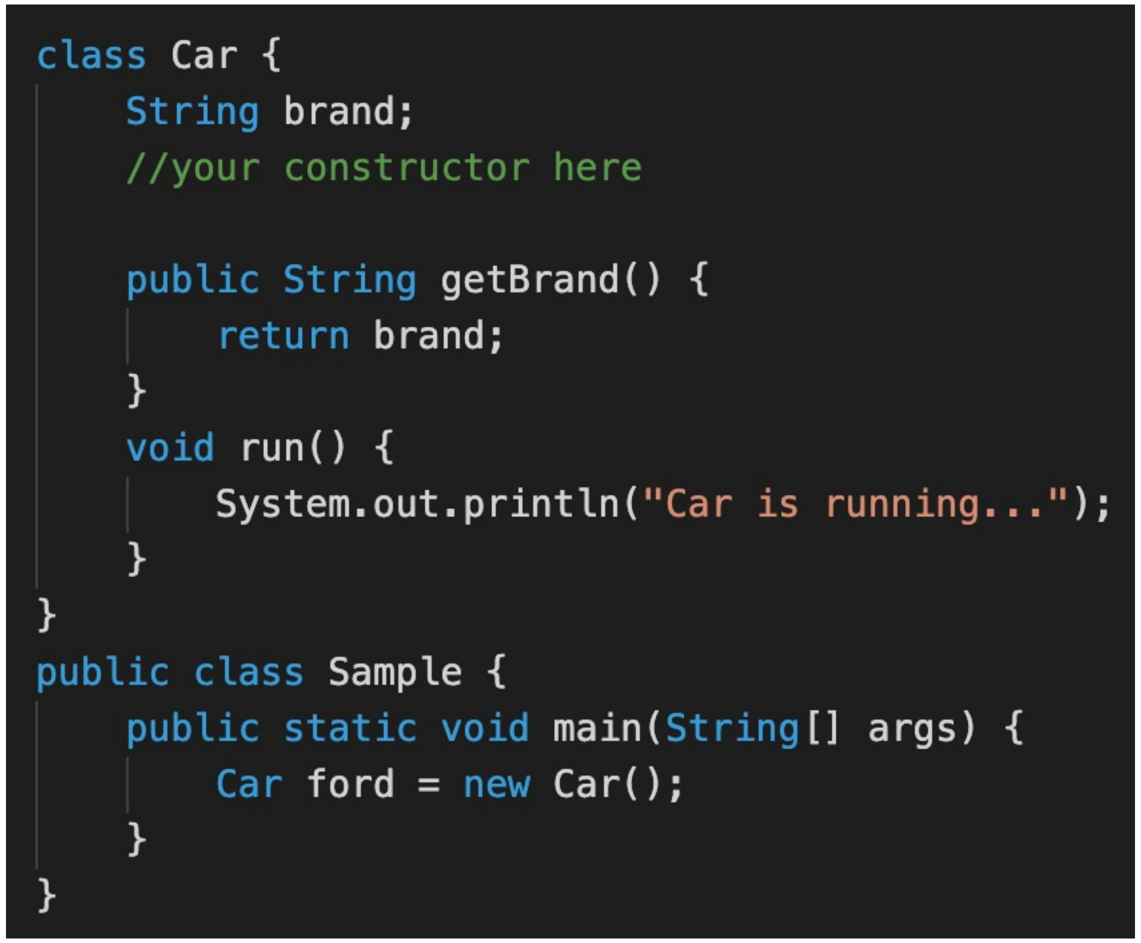
```java
public class OverloadingExample2 {
  int rollNum;
  OverloadingExample2() {
    rollNum = 100;
  }
  OverloadingExample2(int rnum) {
    this();
    /*
     * this() is used for calling the default constructor from parameterized
     * constructor. It should always be the first statement inside constructor body.
     */
    rollNum = rollNum + rnum;
  }
  public static void main(String args[]) {
    OverloadingExample2 obj = new OverloadingExample2(12);
    System.out.println(obj.rollNum);
  }
}
```

Lab 03

EXERCISE-1:

Write a constructor in the Car class given below that initializes the brand class field with the string “Ford”.

Call the getBrand() method in the main method of the Sample class and store the value of the brand in a variable, and print the value.

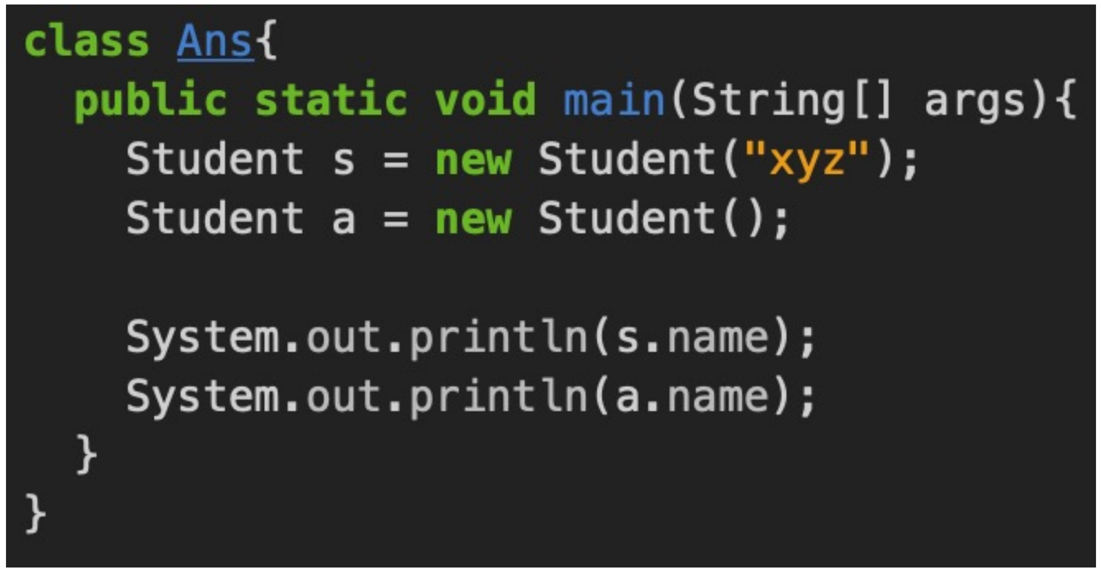


```java
class Car {
    String brand;
    //your constructor here

    public String getBrand() {
        return brand;
    }
    void run() {
        System.out.println("Car is running...");
    }
}
public class Sample {
    public static void main(String[] args) {
        Car ford = new Car();
    }
}
```

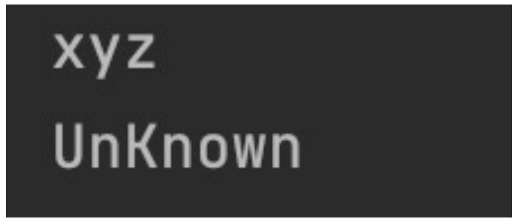
Lab 03

EXERCISE-2:

Write a program to print the names of students by creating a Student class. If no name is passed while creating an object of Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating object of Student class.



```java
class Ans{
  public static void main(String[] args){
    Student s = new Student("xyz");
    Student a = new Student();

    System.out.println(s.name);
    System.out.println(a.name);
  }
}
```

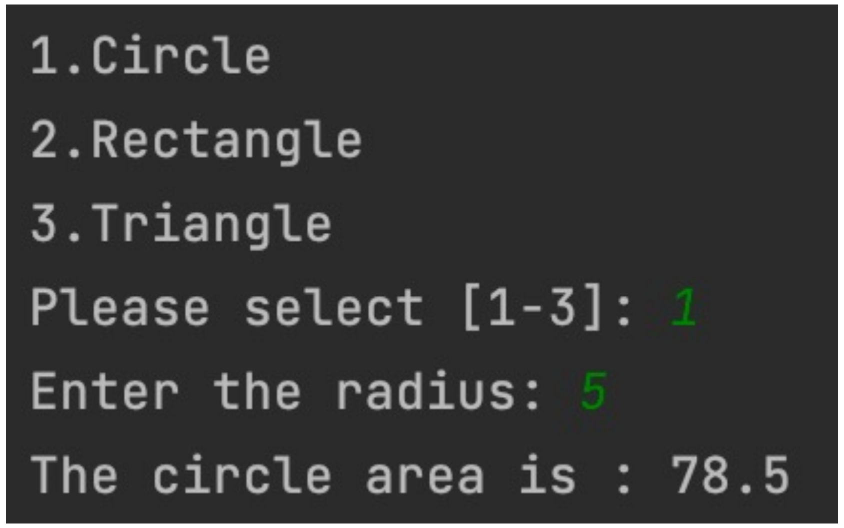


```
xyz
UnKnown
```

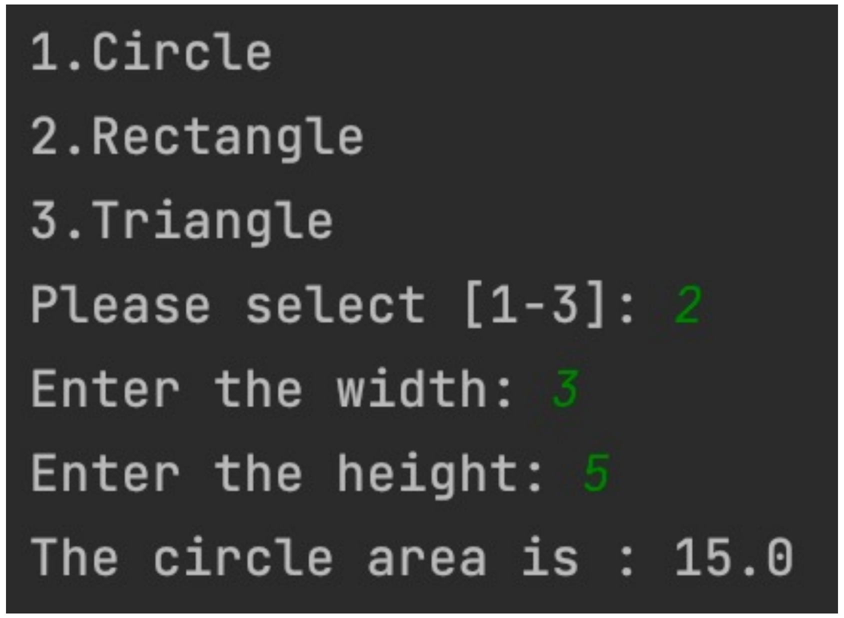
Lab 03

EXERCISE-3:

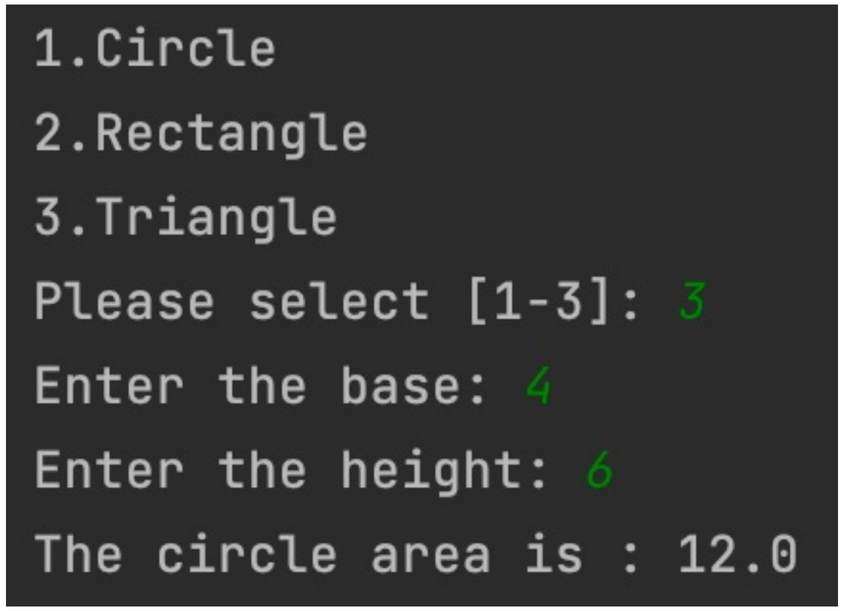
From the previous lab2, modify the code to use a constructor to initialize the attribute used in each class. (You have to determine what is the class's attribute).



```
1.Circle
2.Rectangle
3.Triangle
Please select [1-3]: 1
Enter the radius: 5
The circle area is : 78.5
```



```
1.Circle
2.Rectangle
3.Triangle
Please select [1-3]: 2
Enter the width: 3
Enter the height: 5
The circle area is : 15.0
```



```
1.Circle
2.Rectangle
3.Triangle
Please select [1-3]: 3
Enter the base: 4
Enter the height: 6
The circle area is : 12.0
```

Reference

- 6 OOP Concepts in Java with examples [2020] · Raygun Blog https://raygun.com/blog/oop-concepts-java/ Accessed: 2020-07-04
- Encapsulation in Java – GeeksforGeeks https://www.geeksforgeeks.org/encapsulation-in-java/ Accessed: 2020-07-04
- ava Encapsulation and Getters and Settershttps://www.w3schools.com/java/java_encapsulation.asp Accessed: 2020-07-05
- How to explain object-oriented programming concepts to a 6-year-oldhttps://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/ Accessed: 2020-07-05
- Constructors in Java - A complete study!!https://beginnersbook.com/2013/03/constructors-in-java/ Accessed: 2020-07-05
- Visibility of Variables and Methods - Learning Java, 4th Edition [Book]https://www.oreilly.com/library/view/learning-java-4th/9781449372477/ch06s04.html Accessed: 2020-07-10
- new operator in Java - GeeksforGeekshttps://www.geeksforgeeks.org/new-operator-java/?ref=lbp Accessed: 2020-07-11