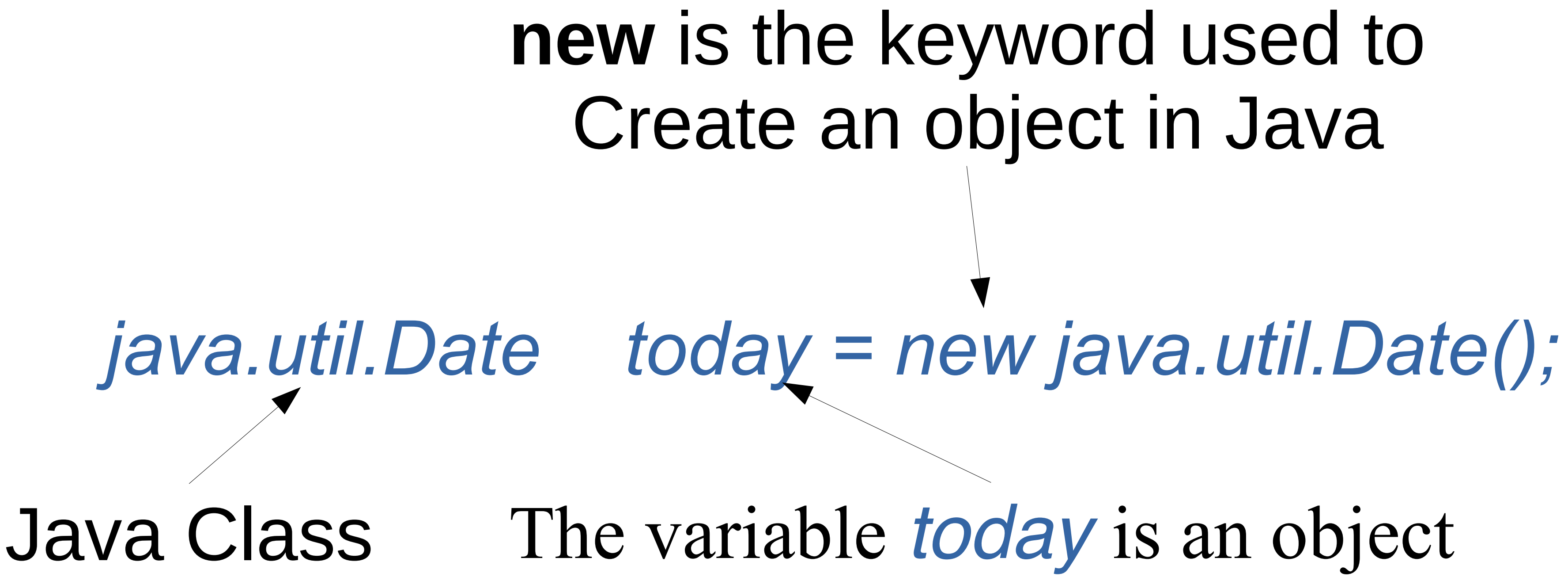
Data structures:

# Class and Object

Dr. Sophea PRUM
sopheaprum@gmail.com

# Class vs. Object

- Class is a **template** that is used to define a new type of object. It serves as a"**blueprint"** for objects of that type
- Object is an **instance** of class
- Example: we create an object *today*



**new** is the keyword used to Create an object in Java

```
java.util.Date    today = new java.util.Date();
```

Java Class

The variable *today* is an object

# Class vs. Object

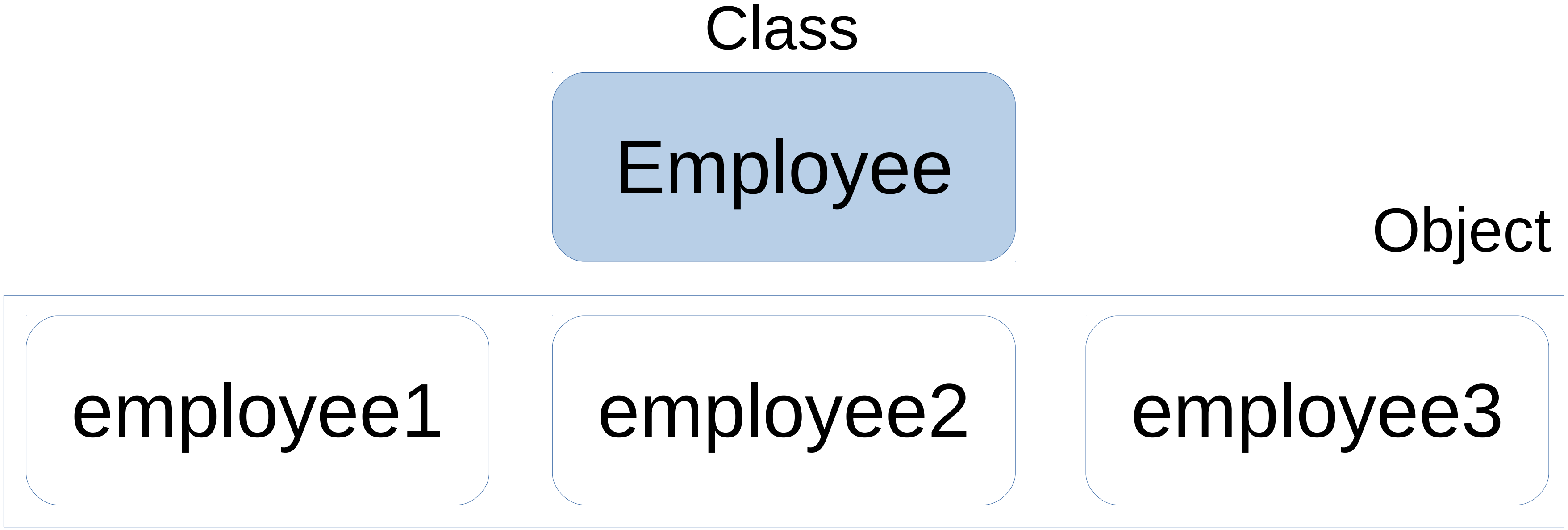
- Naming convention
  - Class name must start with an uppercase character and must be a noun e.g. *Employee, String, Color, Button, System, Thread etc.*

```
public class Employee {
    //block of code
}
```

# Class vs. Object

- Create objects *emplyee1, employee2, employee3* using the keyword **new**

```
Employee employee1 = new Employee();
Employee employee2 = new Employee();
Employee employee2 = new Employee();
```



Class

Employee

Object

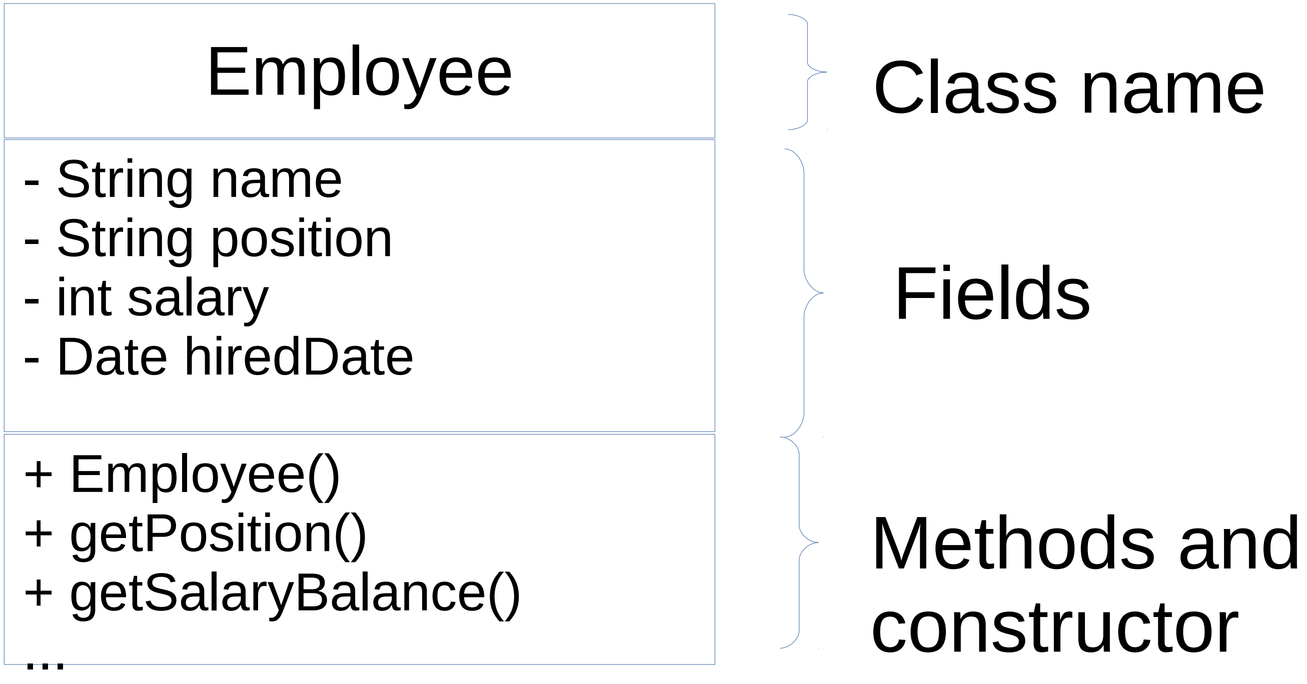
employee1

employee2

employee3

# Java class

- A Java class consists of three kinds of members: *fields*, *methods*, and *constructors*
- We can represent a Class using *Unified Modeling Language* **(UML)**



Employee

- String name
- String position
- int salary
- Date hiredDate

+ Employee()
+ getPosition()
+ getSalaryBalance()
...

Class name

Fields

Methods and constructor

# Java class - field

- *A field*
  - is a variable inside a class
  - Is used to store the data for class objects

```
public class Employee {
    String  name      ;
    String  position ;
    int      salary   ;
    Date     hiredDate;
}
```

# Java class - field

- Java Field Access Modifiers
  - determines whether the field can be accessed by its own Class or other Classes
  - four possible access modifiers for Java fields:

| 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 |

# Java class - field

- Java Field Access Modifiers

How to use modifier in java:

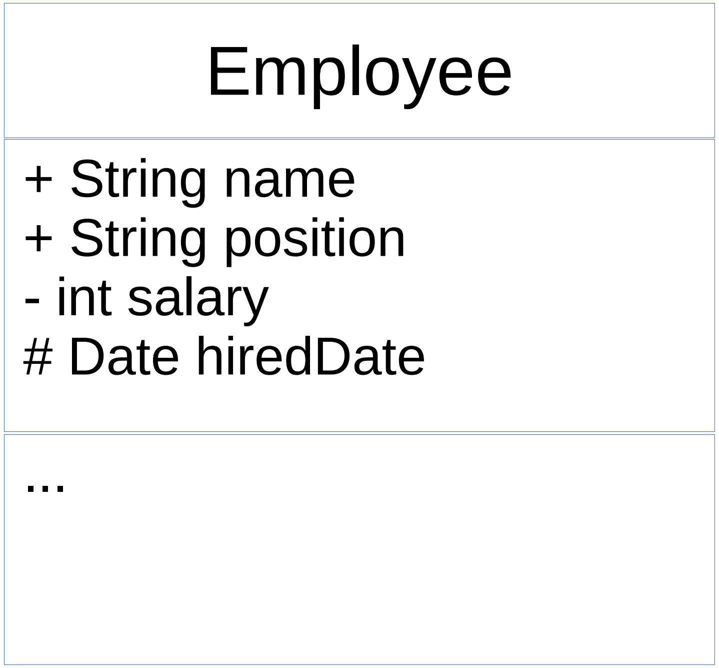
**Modifier** Type variable_name;

```
public class Employee {
    public String  name      ;
    public String  position ;
    private int      salary   ;
    Date     hiredDate;
}
```

# UML Access Modifiers

- 3 modifiers used in ULM (Unified Modeling Language): + public, - private, # protected

```
public class Employee {
    public String  name     ;
    public String  position ;
    private int     salary   ;
    protected Date    hiredDate;
}
```



Employee

+ String name
+ String position
- int salary
# Date hiredDate

...

# Java class - field

- Static fields
  - A *static field* belongs to the class
  - The value of the *static field* is the same for all the objects of that class

```
public class Employee {
    String  name     ;
    String  position ;
    int     salary   ;
    Date    hiredDate;
    static String companyName;
}
```

# Java class - field

- Static fields
  - CompayName is a static field. This filed can be accessed directly from the class and from its objects

```
Employee.companyName = “Geek Dev”

System.out.println(Employee.companyName);

Employee emp1 = new Employee();
System.out.println(emp1.companyName);
```

# Java class - field

- Non-static fields
  - are located in the instances of the class
  - each instance of the class can have its own values for these fields
  - non-static fields can be accessed only by the object

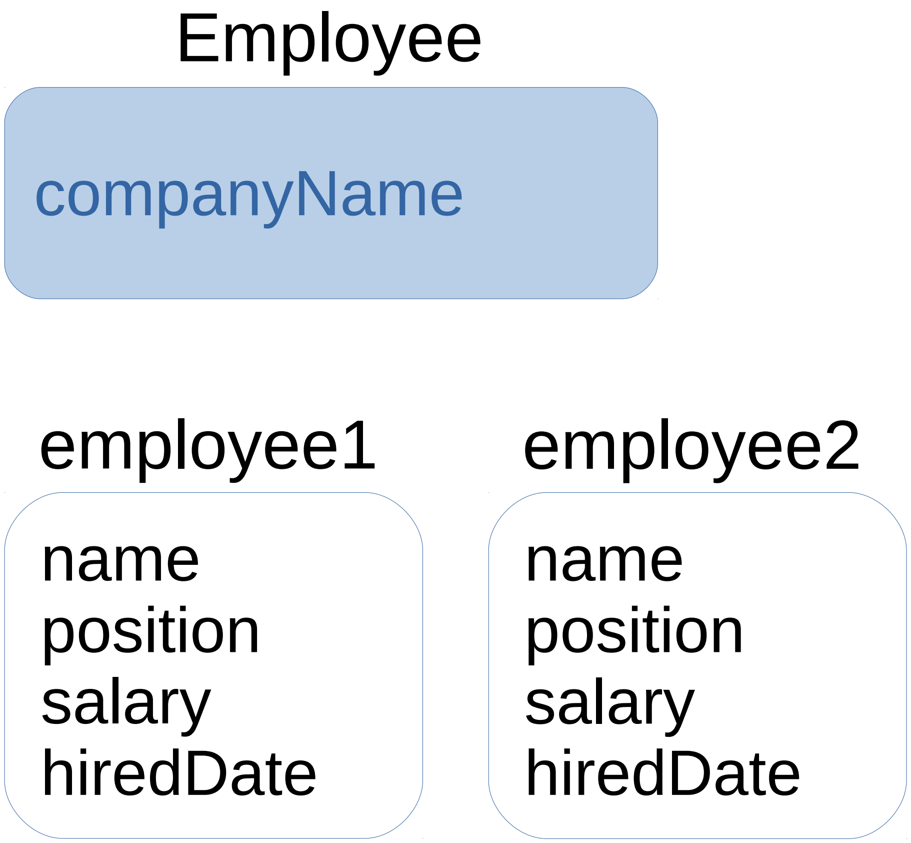
```
Employee employee1 = new Employee();
System.out.println(employee1.name);

System.out.println(Employee.name);
```

==> Error

# Java class - field

- Static vs. Non-static fields

```
public class Employee {
    String  name     ;
    String  position ;
    int     salary   ;
    Date    hiredDate;
    static String companyName;
}
```



# Java class - field

- Final field
  - A *final* field cannot have its value changed, once assigned.
  - The value of the *final* filed can be assigned only in constructor or when declaring the variable
  - The final field belongs to objects. ==> different object can hold different value

```
public class Employee {
    ...
    final float impactFactor=2.5;
}
```

# Java class - field

- Static final field
  - Is used to create a constants
  - The value of this field belong to Class
  - All the objects of this class have the same value

```
public class Employee {
    ...
    static final float impactFactor=2.5;
}
```

# Java class - field

- Final vs. static final

```
public class Employee {
    ...
    final float impactFactor=2.5;
}
```

Employees may have different impactFactor value

```
public class Employee {
    ...
    static final float impactFactor=2.5;
}
```

All the employees have the same impactFactor value

# Java class - method

- A method contains a serie of well **disigned statements** that performs some operations on some data
- Naming convention
  - A method name start with lowercase character
  - The second word start with uppercase character
  - Chose the names that have some meaning

# Java class - method

```
public class Employee {
    ...
    public void viewEmployee(String someMessage){
        System.out.println(“Here is some message: ”+someMessage);
        System.out.println(“Employee name: ”+this.name);
        System.out.println(“Employee position: ”+this.position);
        ...
    }
}
```

This method **viewEmployee** have one parameter as input called **someMessage** and does not return any value (**void**)

- Access Modifiers ==> Field access modifier

# Java class - method

– How to call the method

```
//Create an object
Employee employee1 = new Employee();
//Call the method someMessage
employee1.viewEmployee(“Viewing employees”);
```

# Java class - method

- How to call the method from another method

```
public void callSum() {
    int theSum = add(1, 3);
    System.out.print(theSum);
}

public int add(int value1, int value2) {
    return value1 + value2;
}
```

# Java class - constructor

- Constructors are special methods that are called when an object is instantiated
  - Generally used to initiate the value(s) of field(s)
  - Must have the same name as its Class
  - Do not return any value
  - Java generats a default constructor in every class
  - The default constructor does not take any parameter

# Java class - constructor

- Example of class Employee

```
public class Employee {
    String  name     ;
    String  position ;
    int     salary   ;
    Date    hiredDate;

    public void toString(){
        System.out.println(“Employee name: ”+this.name);
    }
}
```

# Java class - constructor

```
//Create an object
Employee employee1 = new Employee();
```

Create an object *employee1* using the default constructor

==> No initiate value of the fields (name, position, salary, hiredDate)

However, the values of these fields can be initialized by using **setter**

# Java class - constructor

- Example

```
//Create an object
Employee employee1 = new Employee();

//Initialize the value of field name and salary
using setters

employee1.setName(“Titi”);
employee1.setSalary(4000);
...
```

# Java class - constructor

- Create our own constructors

```
public class Employee {
    String  name     ;
    String  position ;
    int     salary   ;
    Date    hiredDate;
    Public Employee(String name){
        this.name = name;
    }
}
```

**Note:** the keyword **this** is used to invoke current class field, method or constructor

**this.name ==>**
*name* is the field of the current class Employee and **NOT** the parameter of the constructor

# Java class - constructor

- Create an object employee1 using our own constructor

```
//Create an object
Employee employee1 = new Employee(“Titi”);
```

==> The value of the field *name* of the object *employee1* is initialized in the constructor

# Java class - constructor

- Constructor Overloading
  - A class can have multiple constructors , as long as the parameters they take are not the same.
  - This is called Constructor Overloading

# Java class - constructor

- Constructor Overloading

```
public class Employee {
    ...
    Public Employee(String name){
        this.name = name;
    }
    Public Employee(String name, String position, int salary,
                    Date hiredDate){
        this.name = name;
        this.position = position;
        this.salary = salary;
        This.hiredDate = hiredDate;
    }
}
```

# Java class - constructor

- Constructor Overloading

```
Employee employee1 = new Employee(“Titi”);

Date today = new Date();

Employee employee2 = new Employee(“Titi”, “IT
manager”, “8000”, today);
```

# Accessors: getter and setter

- Setter: special method used to initialize the value of each field. It does not return anything

```
public void setName(String name) {
    this.name = name;
}
public void setPosition(String position) {
    this.position = position;
}
public void setSalary(int salary) {
    this.salary = salary;
}
public void setHiredDate(Date hiredDate) {
    this.hiredDate = hiredDate;
}
```

# Accessors: getter and setter

- Getter: special method used to get the value of field(s)

```
public String getName() {
     return name;
}
public String getPosition() {
     return position;
}
public int getSalary() {
     return salary;
}
public Date getHiredDate() {
     return hiredDate;
}
```

# Accessors: getter and setter

- Why we must use getter and setter
  - Getter and setter are accessors which are used to access to fields of an object
  - Public getter and setter allow to access to private fields of an object from outside of the class
  - However, it is **highly recommended** to used getter and setter even if the field is public
  - Create a public field is **NOT recommended**
  - **Do not directly exposing fields of a class**
  - Using getter and setter allow you to have fully control when accessing to each field

# JVM: memory management

- JVM mainly uses two spaces of memory: Stack and Heap
  - Stack memory
    - is used for execution of a thread
    - whenever a method is invoked, a new block is created in the stack memory to hold local primitive values and reference to other objects
  - Heap memory
    - is used store the object
    - Object is referenced by the variable(s) in stack memory

# JVM: memory management

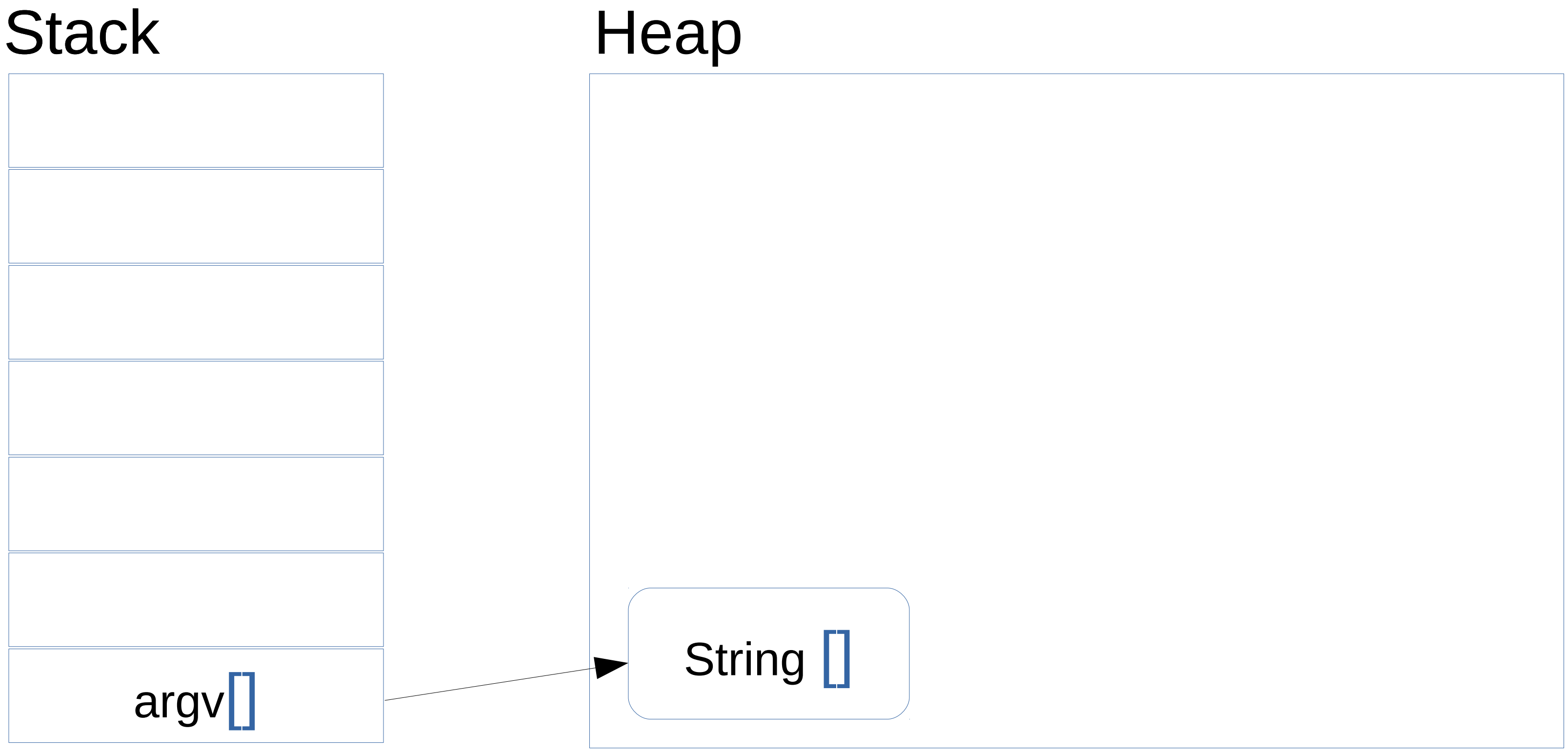
```
static void doSomeThing(Rectangle r){
    r1.height = r1.height * 2; //7
}
public static void main(String[] args) { //1
    int i = 10; //2
    String s = "Hello world!"; //3
    Rectangle r1 = new Rectangle(10,10); //4
    Rectangle r2 = r1; //5

    doSomeThing(r1); //6
}
```

# JVM: memory management

public static void main(String[] args) //1

The parameter *args* is an array of String, where String is class type

==> JVM create an object in Heap and a variable *argv* in stack.
The variable *argv* contains the reference to the object created in Heap memory

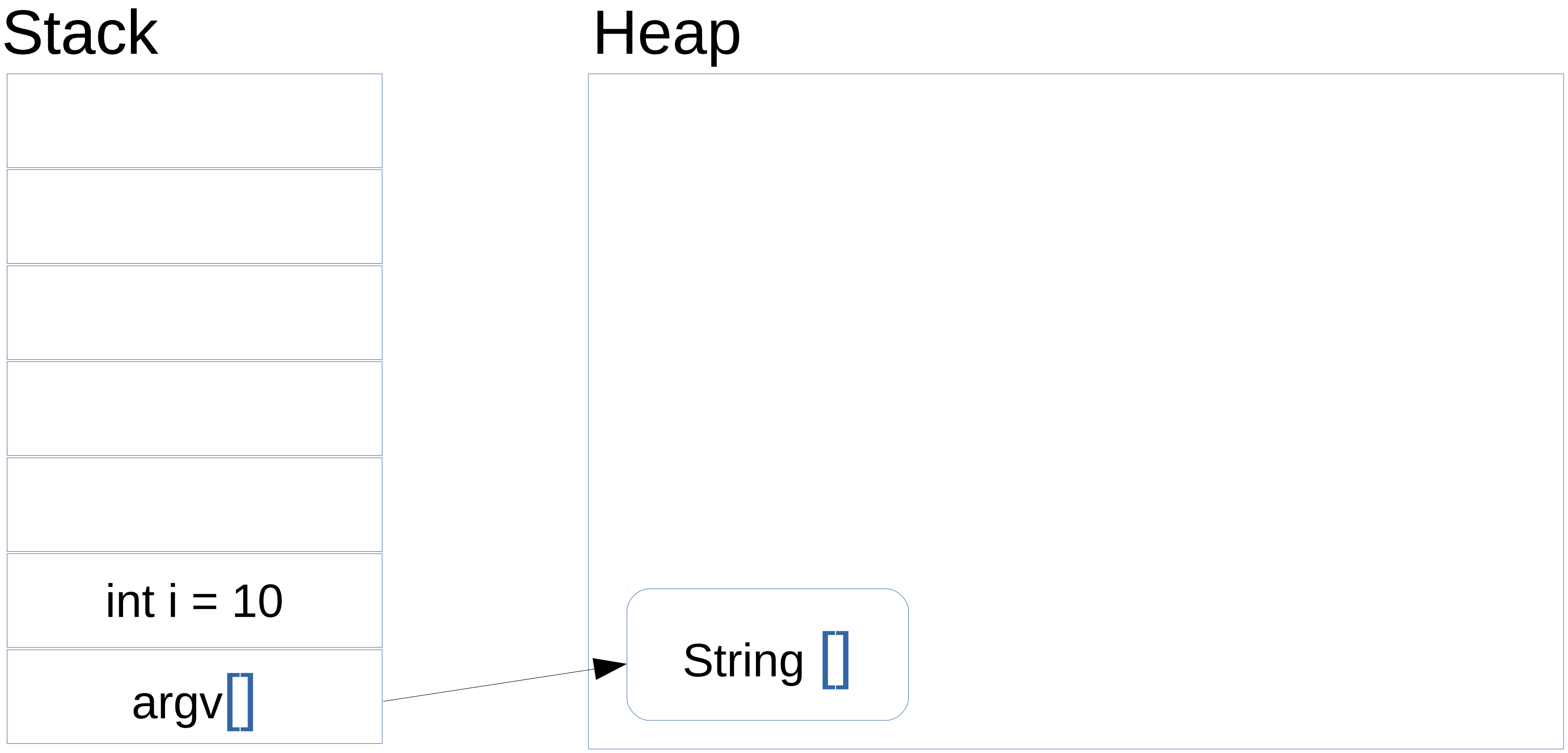


# JVM: memory management

int i = 10; //2

The variable *i* is a primitive type

==> JVM create a variable in stack to store is value directly

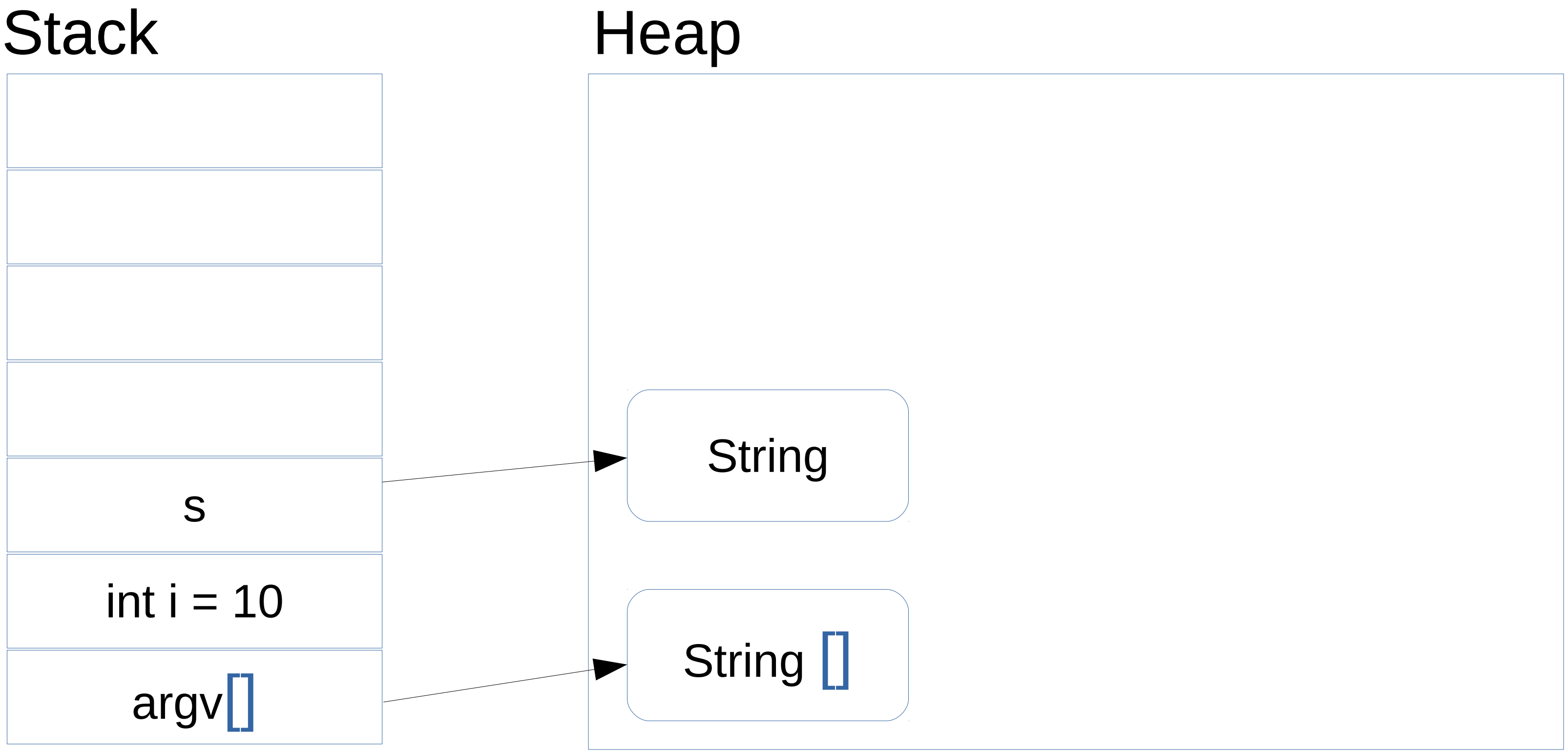


# JVM: memory management

String s = "Hello world!"; //3

The variable *s* is a Class type

==> JVM create an object of type String in Heap and a variable *s* in stack. The variable *s* hold the reference to the object created in heap memory.

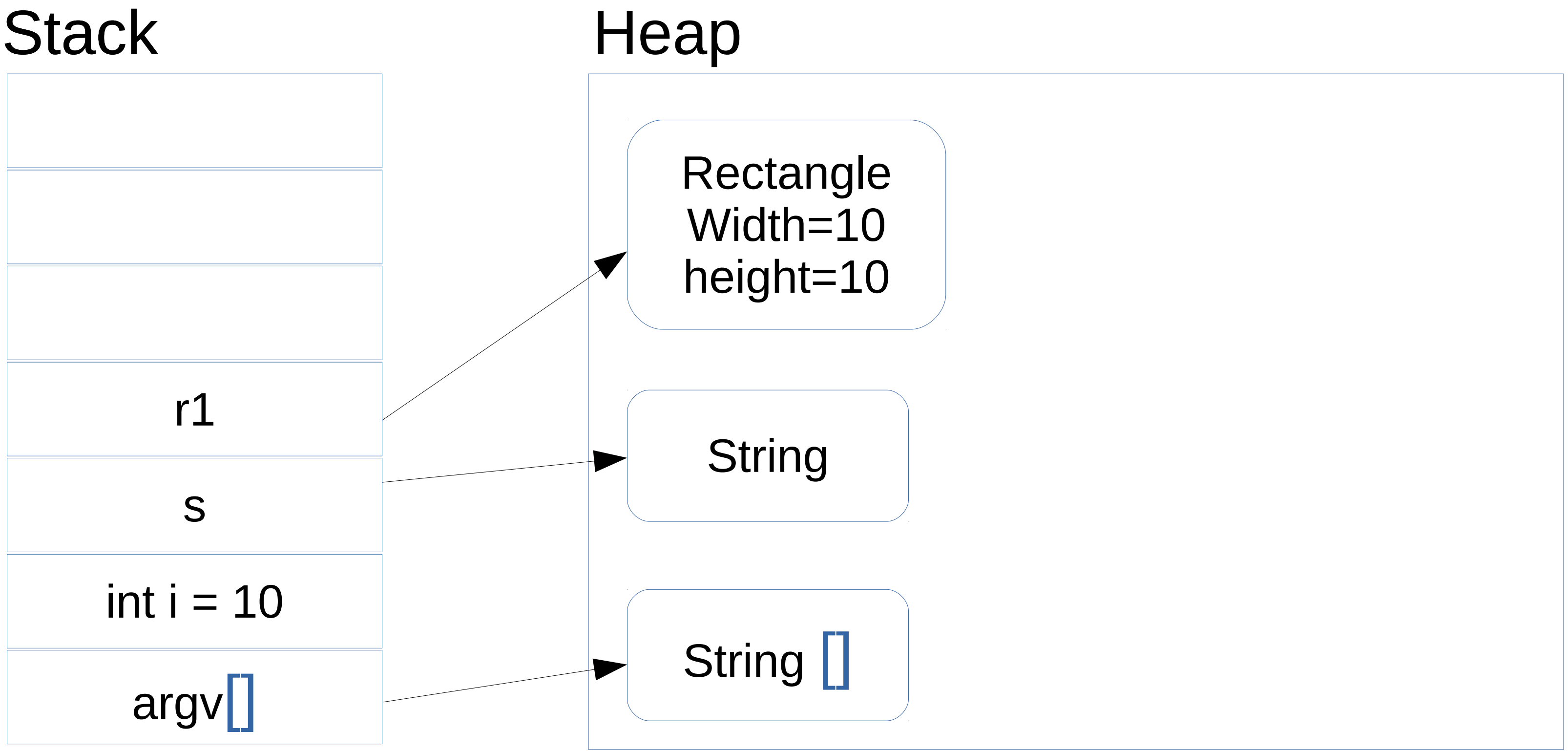


# JVM: memory management

Rectangle r1 = new Rectangle(10,10); //4

The variable *r1* is a Class type

==> JVM create an object of type Rectangle in Heap and a variable *r1* in stack. The variable *r1* hold the reference to the object created in heap memory.

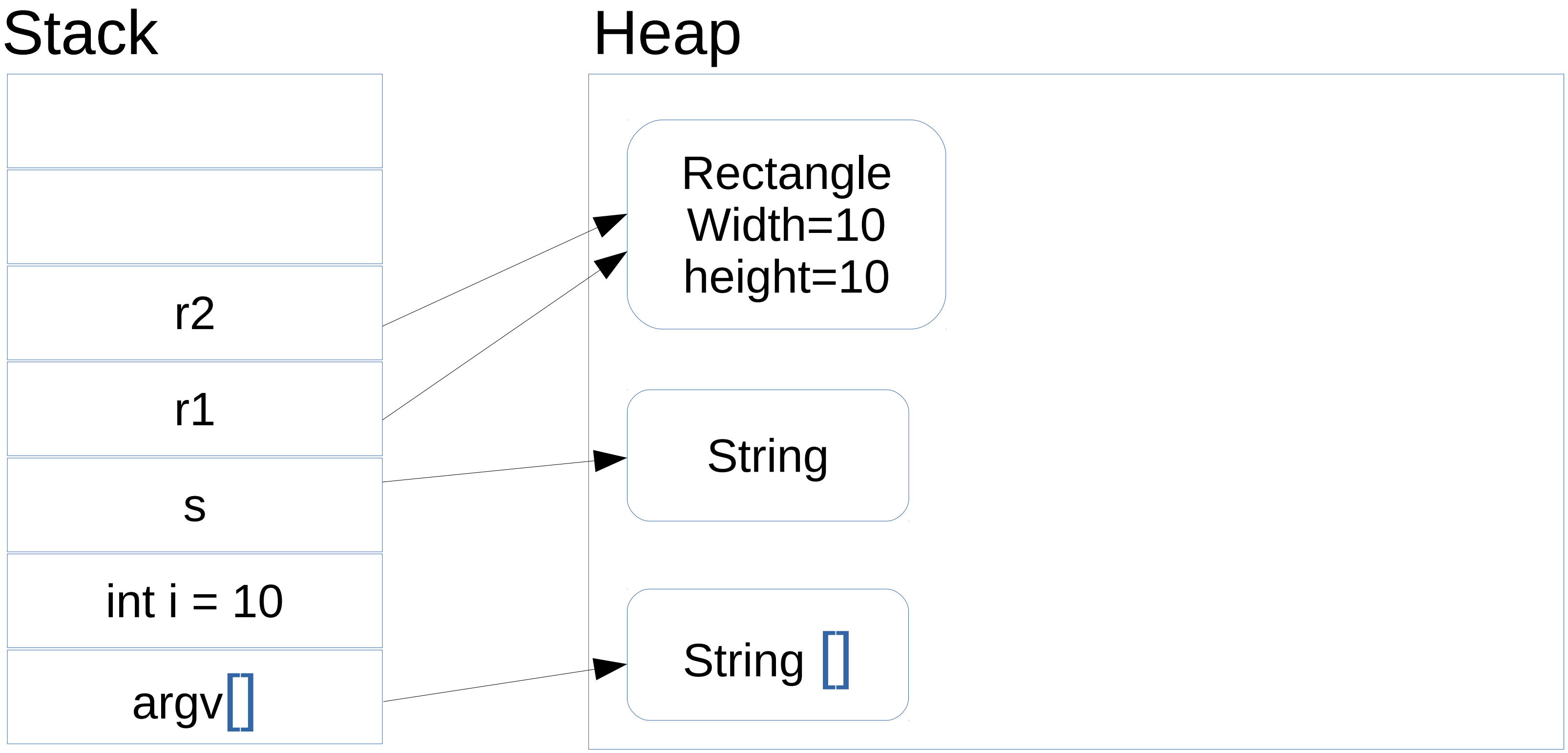


# JVM: memory management

Rectangle r2 = r1; //5

The variable *r2* is a Class type. The value of *r2* is assigned to the same value of *r1* ==> ***r2* hold the same reference as *r1***

==> JVM **WILL NOT create** duplicate object in Heap. However, its create another variable *r2* in Stack holding the reference to the object created in Heap

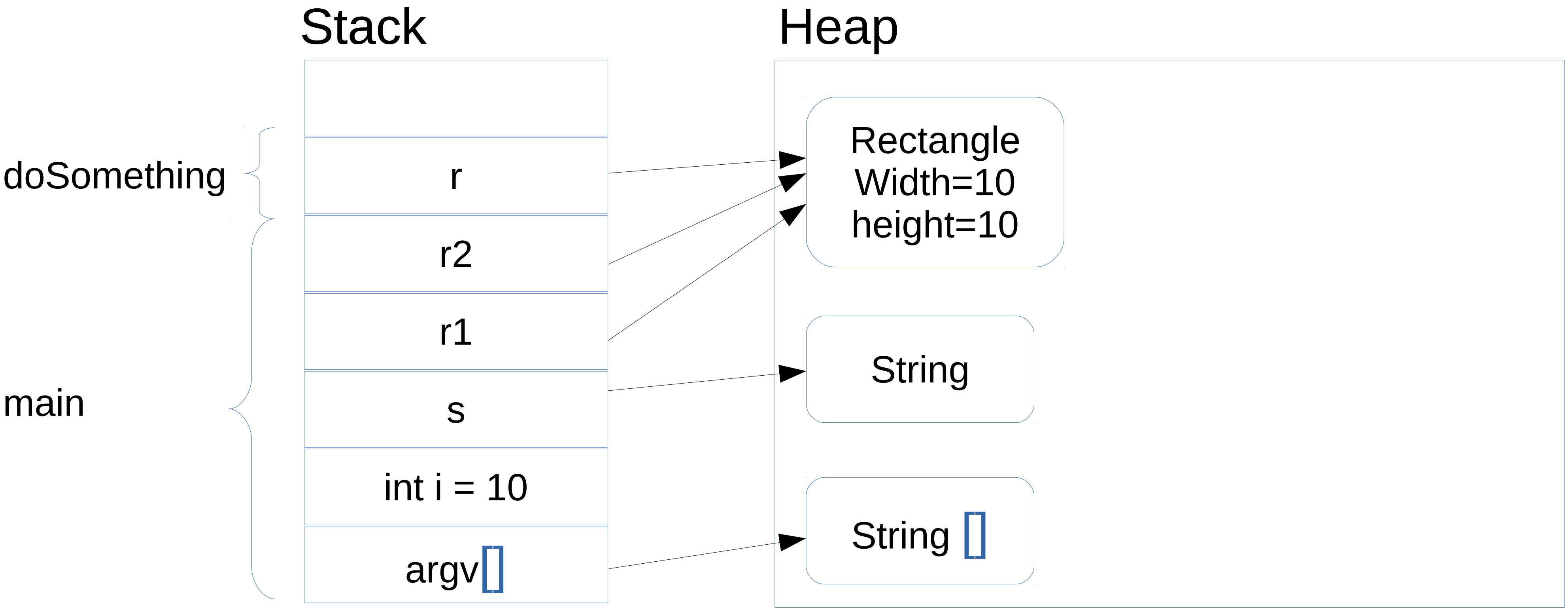


# JVM: memory management

doSomeThing(r1); //6

When calling the method doSomeThing, a block in the top of the stack is created to be used by this method to store parameter(s) and local variable(s)

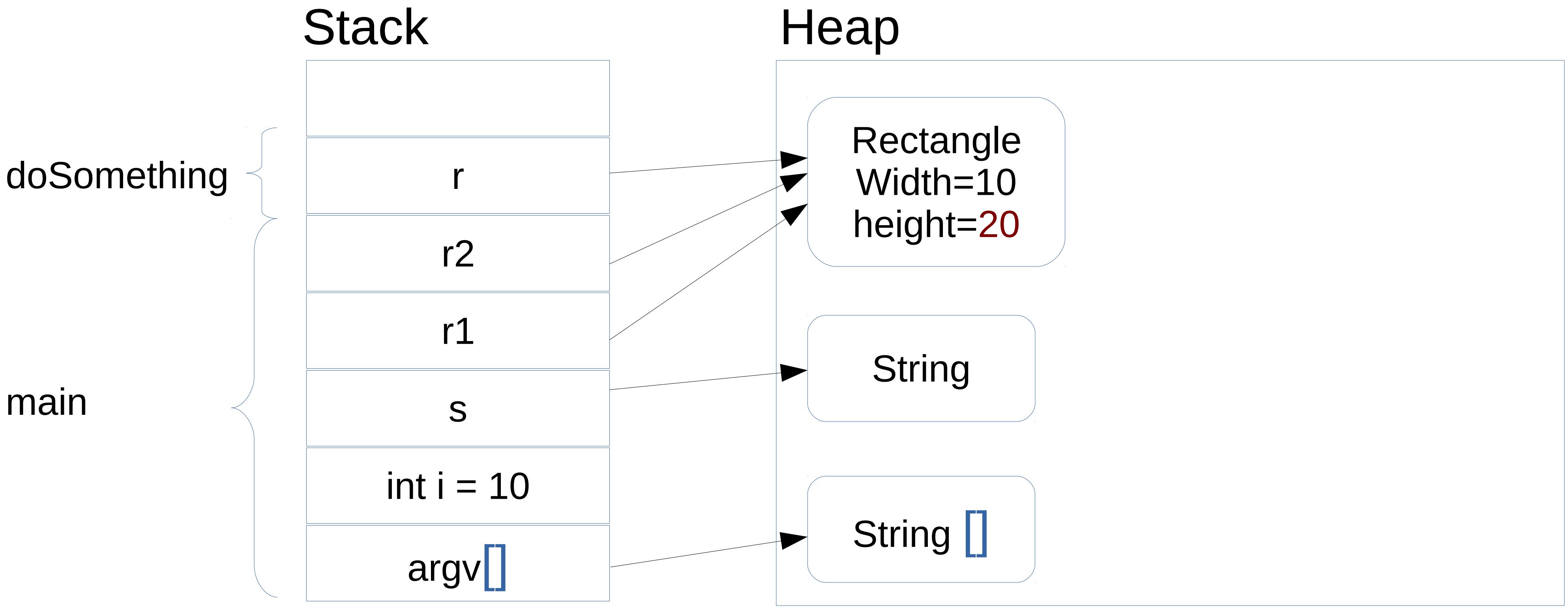
Since Java is **pass by value**, a new reference to Object is created for the method doSomeThing



# JVM: memory management

```
static void doSomeThing(Rectangle r){
     r.height = r.height * 2; //7
}
```

Method doSomeThing modified the value of the object



# JVM: memory management

```
static void doSomeThing(Rectangle r){
    r1.height = r1.height * 2; //7
}
public static void main(String[] args) { //1
    int i = 10; //2
    String s = "Hello world!"; //3
    Rectangle r1 = new Rectangle(10,10); //4
    Rectangle r2 = r1; //5
    System.out.println(r1.height);
    System.out.println(r2.height);

    doSomeThing(r1); //6
    System.out.println(r1.height);
    System.out.println(r2.height);
}
```

```
Output:
r1.height=10
r2.height=10
r1.height=20
r2.height=20
```

# Resumed

- You have learned
  - Class and object
  - Field
  - Method
  - Constructor
  - JVM memory management

# References

- http://tutorials.jenkov.com/java/fields.html
- http://tutorials.jenkov.com/java/methods.html
- http://tutorials.jenkov.com/java/constructors.html
- http://www.journaldev.com/4098/java-heap-space-vs-stack-memory
- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- More Exercises
  - http://www3.ntu.edu.sg/home/ehchua/programming/java/j3f_oopexercises.html