

# Java

## Lecture 4 –

### Objects in memory, Constructors, Passing & Returning values



IT Learning &  
Outsourcing Center

[www.pragmatic.bg](http://www.pragmatic.bg)

Lector: Peter Manolov

Skype: nsghost1

E-mail: [p.manolov@gmail.com](mailto:p.manolov@gmail.com)

Facebook: <https://www.facebook.com/pmanolov>



# Objects in memory

- There are two types of memory in Java – static and dynamic (heap)
- Primitives are stored into the static memory
- Objects are reference data types and are stored into the heap
- The reference to the object is kept in the static memory
- The phrase "*instantiating a class*" means the same thing as "*creating an object*." When you create an object, you are creating an "*instance*" of a class, therefore "*instantiating*" a class.



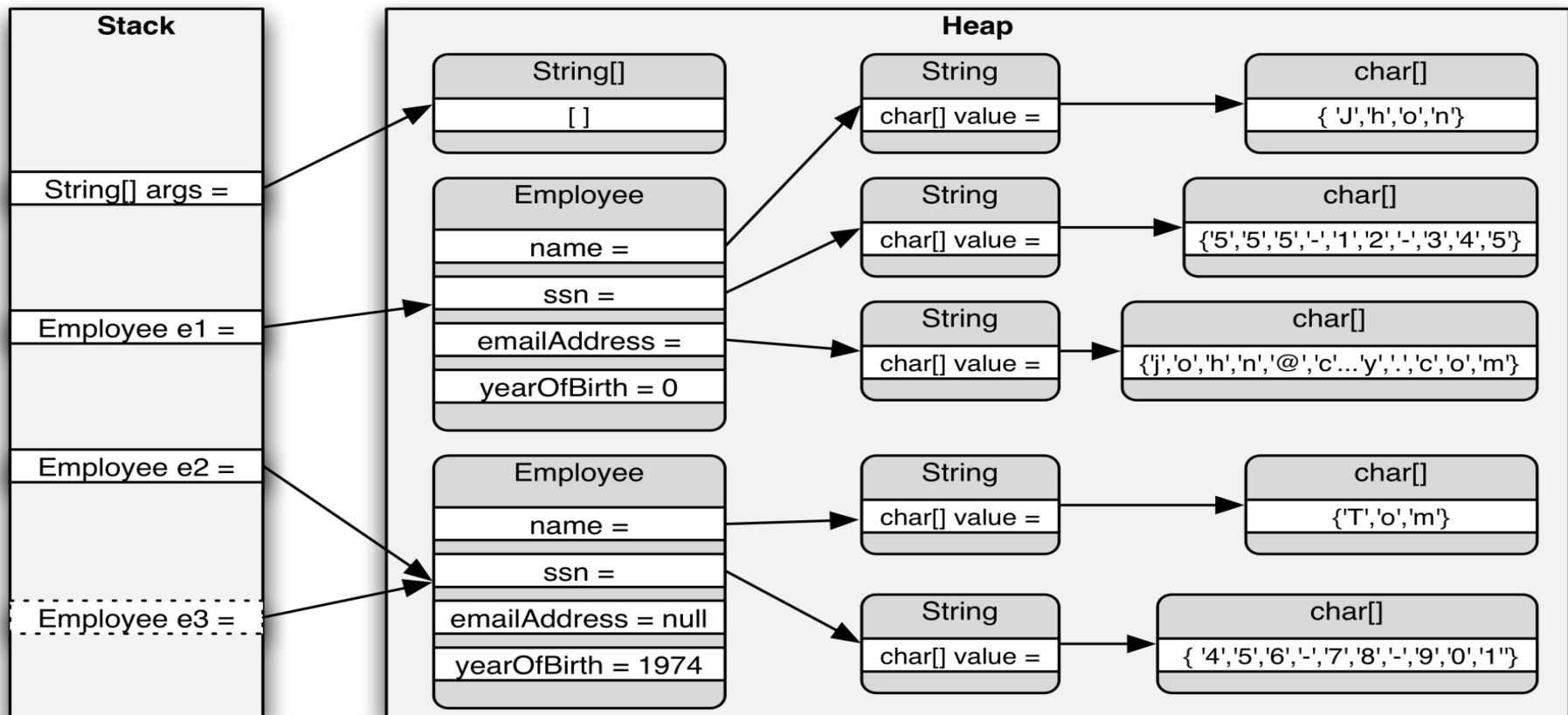
# Objects in memory

- Objects are created via constructors - operator **new** allocates memory in the heap
- The Garbage collector destroys the unused objects – clears the heap
- The destruction of objects is not a programmer task – the garbage collector does it for you



# Objects in memory

## ■ Picture of memory



# Objects - Nothing created in memory



- `int height = new Rectangle().height;`
- This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.
- The rectangle class can be seen under **Rectangle.java** in code examples



# References

- Objects are referent types
- Primitives are not referent types
- Dealing with objects is always dealing with its reference
- Declaration of an object creates a reference but it points to nothing – i.e. null



# References

- Using = with objects deals only with the reference

```
Person mitko;  
Person joro = new Person(18, "Georgi");  
mitko = joro;  
mitko.age = 21;  
System.out.println(joro.age);
```

What is going to be  
the console output?

```
Person joro = new Person(18, "Georgi");  
Person mitko = new Person(20, "Dimitar");  
mitko = joro;
```

What happens with  
{20, Dimitar}?



# Constructor

- Constructor is responsible for creating an object
- You can recognize a constructor because its declaration uses the same name as the class and it has no return type – it should always return the newly created object
- To constructors we can pass parameters
- Constructors should have a body
- Constructors are always named to the class name





# Constructor

## ■ Constructors examples

```
Person () {  
  
}
```

Default  
constructor

```
Person (int ageParam, String nameParam) {  
    age = ageParam;  
    name = nameParam;  
}
```

Constructor with  
parameters for age and  
name



# Car, Person Example

- We will start writing example with Car and Person (the classes from the previous lesson)
- 1. First start with adding the fields `price` and `isSportCar` to the class Car
- 2. Write constructor in class Car:
  - `Car(String modelParam, boolean isSportCarParam, String colorParam)`
  - it sets the parameter to the fields and set default values to `currentSpeed` and `gear`



# Keyword this

- **this** always refers to the current object
- Using **this** in constructors is good practice

```
public class Person {  
    int age;  
    String name;  
  
    Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
}
```

- In the case above, using **this** is obligatory. If **this** is not used, the scope of age and name is restricted only for the constructor i.e. when referencing them,



# Car, Person Example

- In constructor
- `Car(String model, boolean isSportCar, String color)`
- 3. Use *this* and change the parameters' names as shown above



# More about constructors

- Default constructor - a constructor without parameters
- Default constructor is always available if no other constructors are defined
- Each class can have more than one constructor
- If a constructor with parameters is defined, the default constructor is not available
- The constructors can be invoked in the body of another constructor



# More about constructors

```
public class Person {  
    int age;  
    String name;  
    double height;  
  
    Person() {}  
  
    Person(int age) {  
        this();  
        this.age = age;  
    }  
  
    Person(int age, String name) {  
        this(age);  
        this.name = name;  
    }  
  
    Person(int age, String name, double height) {  
        this(age, name);  
        this.height = height;  
    }  
}
```

This constructor uses  
the default one

This constructor  
uses another  
constructor which  
uses the default  
one



# Car, Person Example

- 4. Write constructor in class Car:
- `Car(String model, boolean isSportCar, String color, double price, double maxSpeed)`
- it calls the other constructor and then set the other parameters to the fields. It also checks if the car is sport before setting its maxSpeed to more than 200



# Car, Person Example

- In class Person add 2 constructors:

- 5. Default constructor - it sets age to 0 and weight to 4.0
- Change class Person to contain array of Friends instead of one friend
- 6. `Person(String name, long personalNumber, boolean isMale)`
- it calls the default constructor first, then set the values and initialize the friends array with new array with 3 elements





# Car, Person Example

- 7. Create class Demo with main method and test the constructors of class Car and Person



# Methods - declaration

- return type (boolean, int, String, <any other class>)
- Method name (starts with lowerCase, use CamelCase convention)
- Brackets (mandatory)
- List with parameters in the brackets (not mandatory)
- Body – starts with { and ends with }
- *Parameters* refers to the list of variables in a method declaration.
- *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.
- The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.



# Method - parameter names

- When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.
- The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.



# Returning a Value from a Method (part 1)

- A method returns to the code that invoked it when it
  - completes all the statements in the method,
  - reaches a return statement, or
  - throws an exception (covered later)
- whichever occurs first.



# Returning a Value from a Method (part 2)

- You declare a method's return type in its method declaration. Within the body of the method, you use the **return** statement to return the value.
- Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:
  - **return;**



# Returning a Value from a Method (part 3)

- If you try to return a value from a method that is declared **void**, you will get a compiler error.
- Any method that is not declared void must contain a return statement with a corresponding return value, like this:
  - **return returnValue;**
  - The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.



# Example of method with returned type int

Return type

Method name

Parameters

```
int sum(int a, int b) {  
    int sum = a + b;  
    return sum;  
}
```

body

Return int value



# Returning a Value from a Method (part 4)

## ■ Returning a primitive type example:

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

## ■ Returning reference type example:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                             Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```





# Passing Primitive Data Type Arguments (part 1)

- Primitive arguments, such as an *int* or a *double*, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:



# Passing Primitive Data Type Arguments (part 2)

```
public class PassPrimitiveByValue {  
  
    public static void main(String[] args) {  
  
        int x = 3;  
  
        // invoke passMethod() with  
        // x as argument  
        passMethod(x);  
  
        // print x to see if its  
        // value has changed  
        System.out.println("After invoking passMethod, x = " + x);  
  
    }  
  
    // change parameter in passMethod()  
    public static void passMethod(int p) {  
        p = 10;  
    }  
}
```

After invoking  
passMethod, still  
x = 3



# Passing Reference Data Type Arguments (part 1)

- Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.
- Here is an example:



# Passing Reference Data Type Arguments (part 2)

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {  
    // code to move origin of circle to x+deltaX, y+deltaY  
    circle.setX(circle.getX() + deltaX);  
    circle.setY(circle.getY() + deltaY);  
  
    // code to assign a new reference to circle  
    circle = new Circle(0, 0);  
}
```

```
moveCircle(myCircle, 23, 56)
```

- Let the method be invoked with these arguments:
- Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the



# Car, Person Example

- 8. Create method in class Car
  - `boolean isMoreExpensive(Car car)`
- 9. Test it in class Demo



# Car, Person Example

- 10. Create method in class Car
  - `double calculateCarPriceForScrap(double metalPrice)`
  - 
  - The price = metalPrice \* coef
  - The coefficient starts from 0.2 and depends of the car's color and if it's sport:
    - If the color is black or white, 0.05 is added to the coefficient
    - If the car is sport, 0.05 is added to the coefficient
- 11. Test it in class Demo



# Car, Person Example

- To the class Person add fields:
- 11. money – money of the Person
- 12. car – reference to his own car



# Car, Person Example

- To the class Person add method:

- 13. `void buyCar (Car car)`
- *the person buy the car if has enough money*

- To the class Car add method:

- 14. `void changeOwner (Person newOwner)`





# Car, Person Example

- To the class Person add method:

- `15. double sellCarForScrap(double metalPrice)`
- the method returns the money of the person after the car is sold for scrap



# Summary

- Objects and referent types
- What is a reference
- Constructors
- Default constructor and how to use constructors
- How to call constructor in constructor
- Methods with returned types not void
- How to use *return* keyword