COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Week 3-4: OOD3 Other methods, mutable vs immutable, final variables

Giulia Alberini, Fall 2020

OBJECTS – QUICK REVIEW

Book.java

```
public class Book {
    public String title;
    public String author;
}
```


TestBook.java

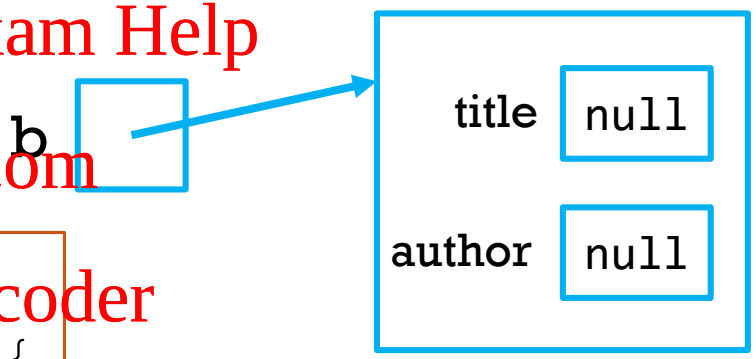
```
public class TestBook {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Matilda";
        b.author = "Roald Dahl";
        System.out.println(b);
    }
}
```

OBJECTS – QUICK REVIEW

```
public class Book {
    public String title;
    public String author;
}
```



```
public class TestBook {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Matilda";
        b.author = "Roald Dahl";
        System.out.println(b);
    }
}
```

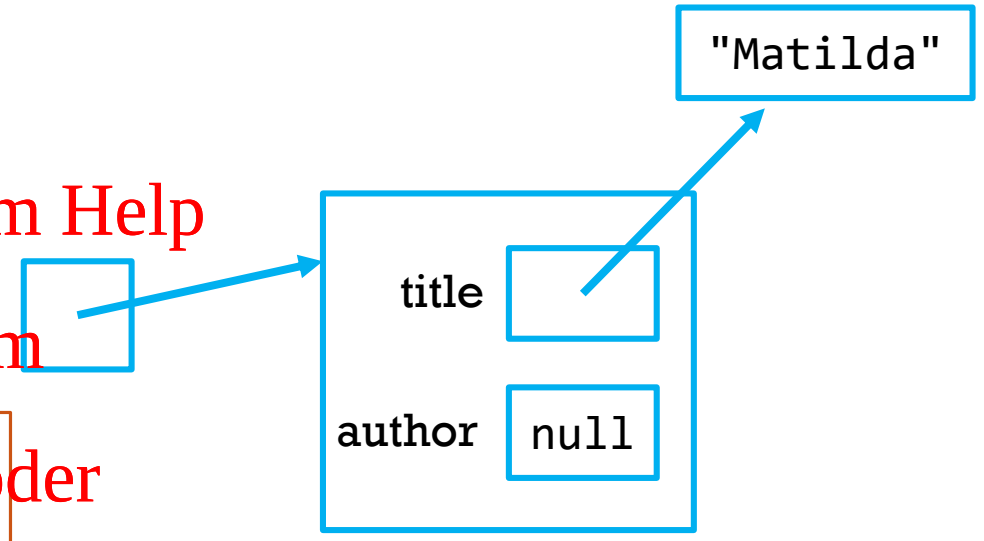


OBJECTS – QUICK REVIEW

```
public class Book {
    public String title;
    public String author;
}
```



```
public class TestBook {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Matilda";
        b.author = "Roald Dahl";
        System.out.println(b);
    }
}
```

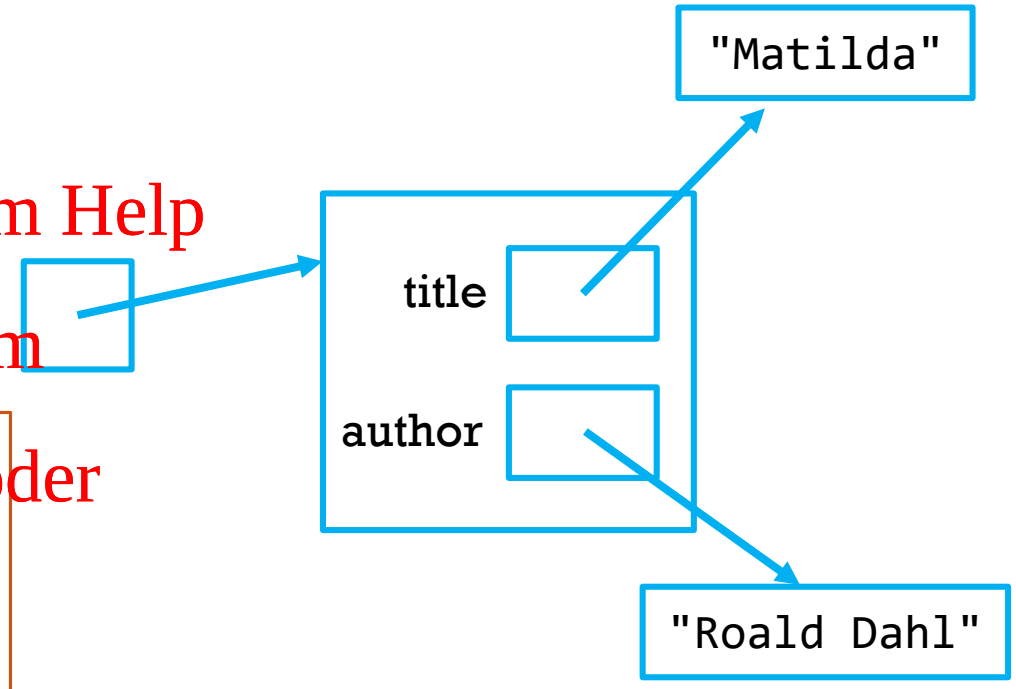


OBJECTS – QUICK REVIEW

```
public class Book {
    public String title;
    public String author;
}
```



```
public class TestBook {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Matilda";
        b.author = "Roald Dahl";
        System.out.println(b);
    }
}
```



OBJECTS – QUICK REVIEW

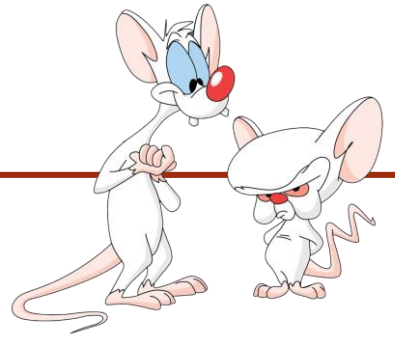
```
public class Book {
    public String title;
    public String author;
}
```



```
public class TestBook {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Matilda";
        b.author = "Roald Dahl";
        System.out.println(b);
    }
}
```

A reference

WHAT ARE WE GOING TO DO IN THIS VIDEO?



OOD3

- Other methods
- Mutable vs Immutable
- `final` variables

OTHER METHODS

STEP 3

```
public class ClassName {
   // some data declared here
   <modifier> <type> <variable_name>;

   public ClassName() {
      //constructor
   }

   // declare other methods
}
```


Data

Method to create an object

Other methods

File name: **ClassName**.java

OTHER METHODS

- You write all the other methods as before, except:
  - If the method is static, it has access to the class variables.

  - If the method is not static, it has access to both the class and the instance variables

RECOMMENDED EXERCISES

In the `Patient` class:

- **add a method `printName` that prints out the name of the patient.**
- **add a method `addTemp` that takes as input a `double` and appends it at the end of the array of temperature of the patient. To do this, you need to create a new array whose length is *greater*, copy all the values over, append the double, and update the value of the corresponding field.**

GET/SET METHODS

GETTERS AND SETTERS

- In general, all fields should be declared as `private`. We can then declare `public` method to regulate how they can be accessed.

- Those methods that give access to the value of a field are formally called "accessors", but commonly referred to as ***getters.***

- Those methods that allows you to modify the value of a field are formally called "mutators", but commonly referred to as ***setters.***

GET (ACCESSOR) METHOD

Most getters have a very similar format:


```
public <type> getField() {
    return this.field;
}
```

SET (MUTATOR) METHOD

Most setters have a very similar format:


```
public void setField(<type> value) {
    this.field = value;
}
```

SET, GET, AND REFERENCES

Careful when writing get/set methods:

- If you are returning a reference type, be sure you are actually giving access to the thing you intend.
- If you are setting a value, be sure that you are using the thing you intend.

TRY IT!

- **Modify the `Patient` class to include get and set methods. How should you do it?**

ENCAPSULATION

- Process of wrapping data and the code acting on that data in one unit. The idea is to better control the data.

- What to do?
  - Make all the fields `private`
  - Provide getters and setters as needed.

- Note: through the methods we can do data validation, while we have little control over the data stored in a `public` field.

toString()

PRIVATE AND INFORMATION DISPLAY

- When your fields are all private, it might be difficult to display the content of an Object outside of the class.
- If we try to use `println` on the Object, we just see a reference.
- References are not very useful when trying to debug a code.

`toString()` AND `println()`

- Each time we use `println()` with objects, it calls the method `toString()` on the object and displays the result.
- By default, `toString()` on an Object returns a `String` representing the address at which the contents of the Object can be found.
- This is not the case for all the Objects, though. If the method `toString()` is included inside the class, then when we use `println()` on variables of such type, not the default `toString()`, but the one from the class is used.

THE `toString()` METHOD

You can write a `toString()` method in any class.

It must have the following header:


```
public String toString(){
   // returns a value of type String
}
```


If you do that, then when you call `print()`/`println()` on an instance of that class, this method is called automatically!

TRY IT!

In the `Patient` **class, add a** `toString()` **method that returns a** `String` **in the following format:**


```
Name: [name]
Age: [age]
Reported Temperatures: [list of temperatures]
```

Now, try to use `print` **on an object of type** `Patient`**.**

MUTABLE vs IMMUTABLE

IMMUTABLE REFERENCE TYPES

- If when you create a class, you make all the fields private, and you do not write any mutator methods, then you have effectively created an Immutable Type!

- If the only way to assign values to fields is through the constructor, then the values of the Object cannot be changed after it has been created.

MUTABLE

Mutator methods change the content of the object.


```
Cat myCat = new Cat("Sprizt");
myCat.setName("Small Cat");
```


The content of object reference by `myCat` has changed by changing the name of the cat.

MUTABLE

If we add a second `Cat` variable referencing to the same cat


```
Cat myCat = new Cat("Spirit");
Cat aCat = myCat;
myCat.setName("Small Cat");
```


Then, also the content of `aCat` is changed after the last instruction.

IMMUTABLE


```
String s = "cats";
s.charAt(0) = 'r'; // compile-time error!
```


There is no method in `String` that allows us to set the value of a character.

IMMUTABLE


```
String s = "cats";
```



```
String t = s;
```



```
t = "dogs";
```

The value of `s` does not change!

MUTABLE VS IMMUTABLE

- Mutable objects: more flexible in what they allow users to do with the object and more efficient because you don't create a new object each time you want to modify the content. But, they can be error-prone.

- Immutable: easier to keep track of what changes when the contents of the object change. They can save you from long hours of debugging.

- Immutable reference types behave like primitive data type.

CONSTRUCTORS AND MUTABLE REFERENCE TYPES

Remember the Patient class:

```
public class Patient{
    private String name;
    private int age;
    private double[] temps;

    public Patient(String n, int a, double[] t) {
        this.name = n;
        this.age = a;
        this.temps = t;
    }
}
```


Code like this can cause issues!

EXAMPLE

Patient.java

```
public class Patient{
    private String name;
    private int age;
    private double[] temps;

    public Patient(String n, int a, double[] t){
        this.name = n;
        this.age = a;
        this.temps = t;
    }
}
```

TestPatient.java

```
public class TestPatient{

    public static void main(String[] args) {
        double[] tempsForP= {37.8, 38.6, 40.0, 37.4, 36.5};
        Patient p = new Patient("John", 42, tempsForP);
        tempsForP[0]= 0.0;
    }

}
```


The last statement in `TestPatient.java` will change the value of the field `temps` even though it is a private field!

GETTERS AND MUTABLE REFERENCE TYPES

Let's add a get method for the field `temps`:

```
public class Patient{
    private String name;
    private int age;
    private double[] temps;

    public Patient(String n, int a, double[] t) {
        ...
    }
    public double[] getTemps() {
        return this.temps;
    }
}
```


Also code like this can cause issues!

EXAMPLE

Patient.java

```
public class Patient{
    private String name;
    private int age;
    private double[] temps;

    public Patient(String n, int a, double[] t){
        ...
    }
    public double[] getTemps() {
        return this.temps;
    }
}
```

TestPatient.java

```
public class TestPatient{

    public static void main(String[] args) {
        double[] tempsForP= {37.8, 38.6, 40.0, 37.4, 36.5};
        Patient p = new Patient("John", 42, tempsForP);
        double[] x = p.getTemps();

        x[0] = 0.0;
    }

}
```


Similarly, the last statement in `TestPatient.java` will change the values of the field `temps` even though it is a private field!

GUIDELINE

- Don't write a constructor initializing a mutable reference type without making a copy!

- Don't add a get/set method to access/mutate a mutable reference type without making a copy!

EXAMPLE

Patient.java

```
public class Patient{
    private String name;
    private int age;
    private double[] temps;

    public Patient(String n, int a, double[] t){
        ...
    }
    public double[] getTemps() {
        // create a copy
        int n = this.temps.length;
        double[] c = new double[n];
        for(int i=0; i<n; i++) {
            c[i] = this.temps[i];
        }
        return c;
    }
}
```

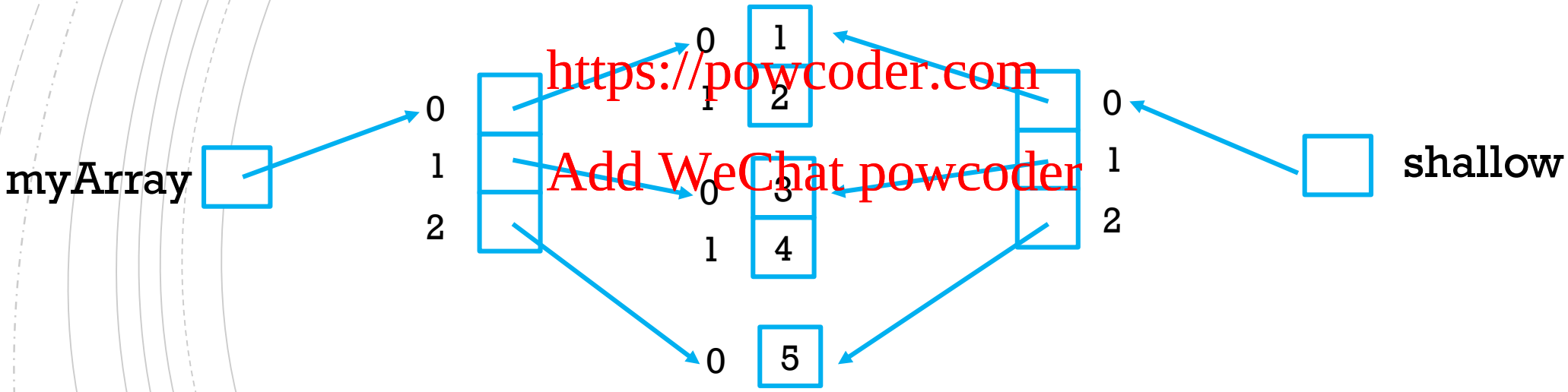

TestPatient.java

```
public class TestPatient{

  public static void main(String[] args) {
     double[] tempsForP= {37.8, 38.6, 40.0, 37.4, 36.5};
     Patient p = new Patient("John", 42, tempsForP);

     double[] x = p.getTemps();
     x[0] = 0.0;
  }

}
```

Now it doesn't matter because the get method returns a copy of the array.

SHALLOW VS DEEP COPY

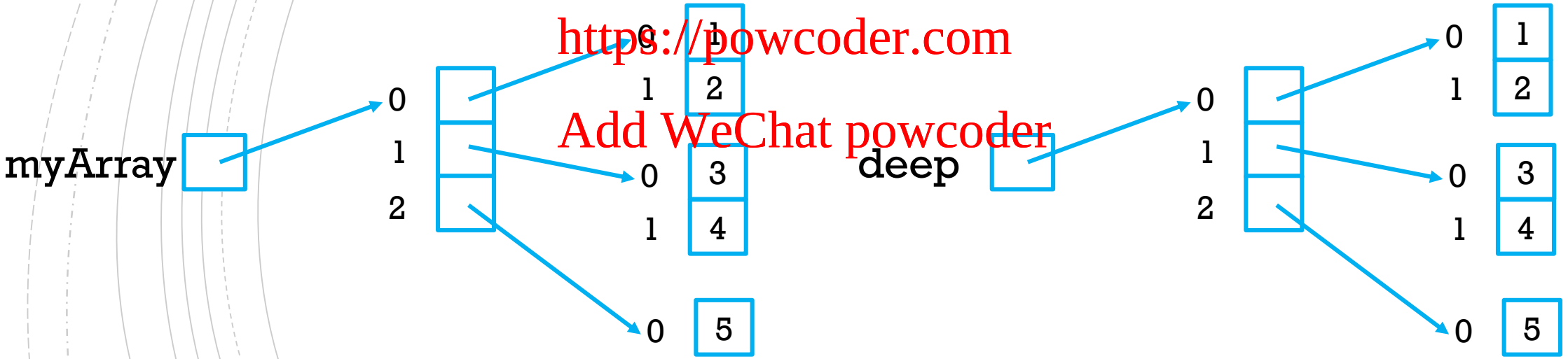
Consider a 2-dimensional array. A shallow copy is obtained by creating a new array and copying to it the references stored in the original array one by one.




```
shallow[0][0] = 5; # affects also myArray
shallow[0] = new int[1]; # does not affect myArray
shallow = new int[5][1]; # does not affect myArray
```

SHALLOW VS DEEP COPY

Consider a 2-dimensional array. A deep copy is obtained by creating a new array and copying to it a (deep) copy of the objects in the original array one by one.



Nothing we can do through `deep` can affect `myArray` (and vice versa)

FINAL

FINAL VARIABLES

If a variable is declared to be `final`, its value can ***never*** be changed after is has been initialized.


```
final int x = 5;
x = 10; // compile-time error!
```



```
final Cat myCat = new Cat("Small cat");
myCat = new Cat("Spritz"); // compile-time error!
```

MUTABLE REFERENCE TYPES AND FINAL

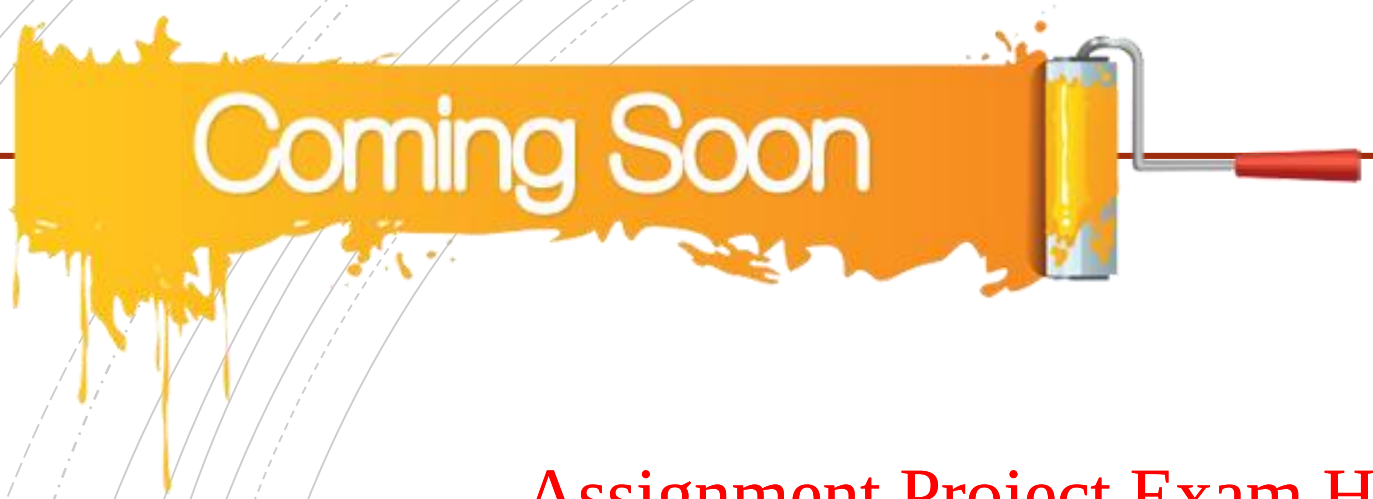
```
final Cat myCat = new Cat("Small cat");
myCat = new Cat("Spritz"); // compile-time error!
```


However, you can still change the object that `myCat` points at, without changing `myCat`'s value.

```
myCat.setName("Spritz"); // no problem!
```

FINAL FIELDS

- Final fields must be initialized!
  (Otherwise **compile-time error**)
  - If the class has a final instance variable (i.e., a final non-static field), you must initialize it in *every* constructor!
  - If the class has a final class variable (i.e. a final static field), you should initialize it in place (on the same line of the declaration) or in a Static Initializer Block.



In the next video:

- UML Diagrams
- Inheritance