



EXPLORE DESIGN PERFECTION



Introduction to Java

Carlos Kavka

ESTECO SpA

esteco.com





EXPLORE DESIGN PERFECTION



Introduction to Java

Part II - classes

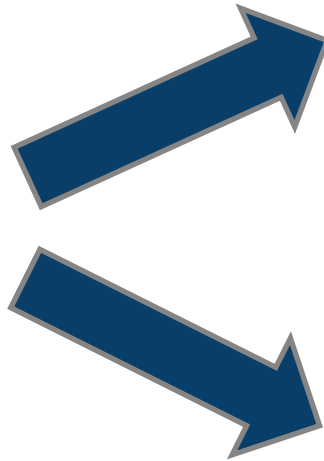
esteco.com



>> Classes

A class is a **template** for data objects

Inside a class it is possible to define



data elements
(called **instance variables**)

functions
(called **methods**)

>> Classes

Class Book with three
instance variables

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

```
Book b1 = new Book();  
Book b2 = new Book();  
Book b3;  
b3 = new Book();
```

New **instances** of the class can be
created with new

The **instance** variables can
be accessed with the dot
notation

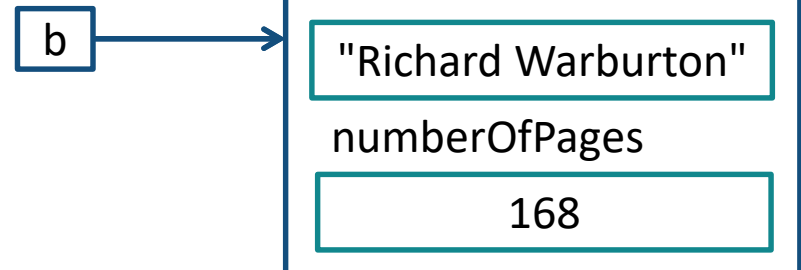
```
b1.title = "Java 8 Lambdas";
```



>> Classes

```
public class ExampleBooks {  
    public static void main(String[] args) {  
  
        Book b = new Book();  
  
        b.title = "Java 8 Lambdas";  
        b.author = "Richard Warburton";  
        b.numberOfPages = 168;  
        System.out.println(b.title + " : " + b.author +  
                             " : " + b.numberOfPages);  
    }  
}
```

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





Constructors

- ✓ The constructors allow the **creation** of instances that are properly initialized
 - ✓ A constructor is a method that:
 - has the **same name** of class to which it belongs and has no specification for the return value.
- ✓ It is possible to define **more than one** constructor for a single class



>> Constructors

```
public class Book {  
    String title;  
    String author;  
    int numberOfPages;  
  
    Book(String tit,String aut,int num) {  
        title = tit;  
        author = aut;  
        numberOfPages = num;  
    }  
}
```

Once a constructor has been defined, the **default** constructor Book() is not available any more.

```
public class ExampleBooks2 {  
    public static void main(String[] args) {  
        Book b = new Book("Java 8 Lambdas","Richard Warburton",168);  
        System.out.println(b.title + " : " + b.author + " : " + b.numberOfPages);  
    }  
}
```



>> Multiple constructors

```
public class Book {  
    String title;  
    String author;  
    int numberOfPages;  
    String ISBN;
```

```
    Book(String tit,String aut,int num) {  
        title = tit; author = aut;  
        numberOfPages = num;  
        ISBN = "unknown";  
    }
```

```
    Book(String tit,String aut,int num,String isbn) {  
        title = tit; author = aut;  
        numberOfPages = num;  
        ISBN = isbn;  
    }  
}
```

It must be possibly to
identify them through
the argument
definition

```
a = new Book("Java 8 Lambdas","Richard  
Warburton",168);
```

```
b = new Book("Java 8 Lambdas","Richard  
Warburton",168,"0-13-027363");
```


>> Methods

- ✓ A method is used to implement the **messages** that an instance (or a class) can receive.
- ✓ It is called by using the **dot** notation.
- ✓ It is implemented as a **function**, specifying arguments and type of the return value.

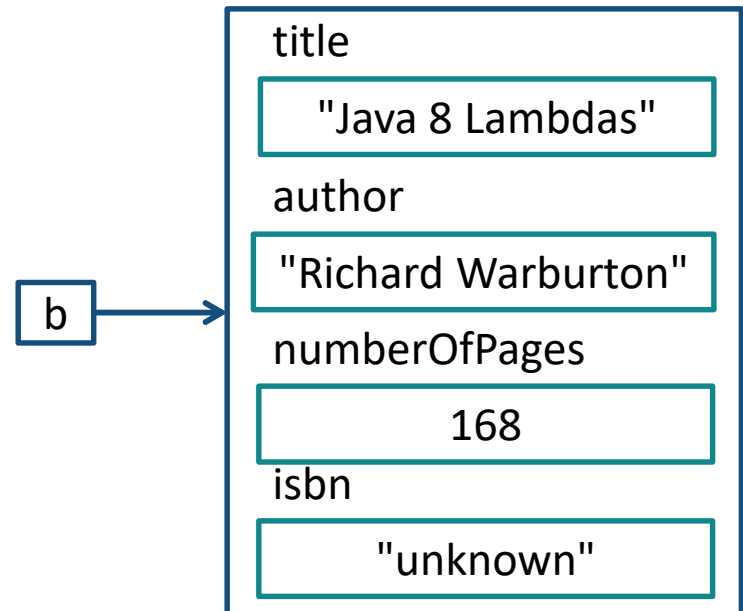


>> Methods

```
public class Book {  
    ...  
    public String getInitials() {  
        String initials = "";  
        for(int i = 0; i < author.length(); i++) {  
            char currentChar = author.charAt(i);  
            if (currentChar >= 'A' && currentChar <= 'Z')  
                initials = initials + currentChar + '.';  
        }  
        return initials;  
    }  
}
```

Initials: R.W.

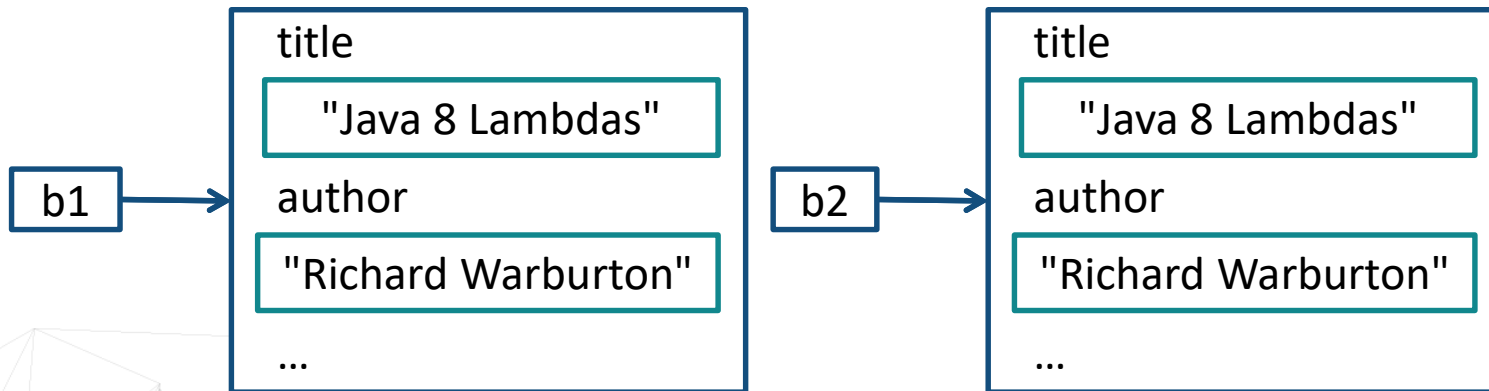
```
b = new Book("Java 8 Lambdas",  
            "Richard Warburton", 168);  
System.out.println(b.getInitials());
```



>> Equality and equivalence

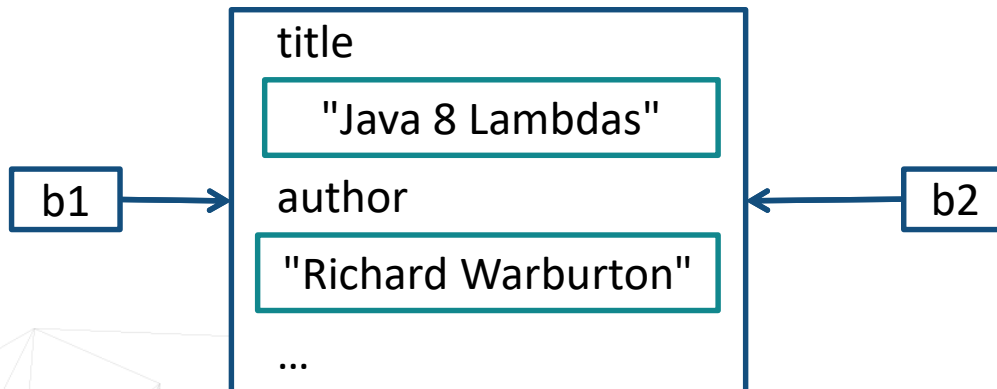
```
public class ExampleBooks6 {  
    public static void main(String[] args) {  
  
        Book b1 = new Book("Java 8 Lambdas","Richard Warburton",168);  
        Book b2 = new Book("Java 8 Lambdas","Richard Warburton",168);  
  
        if (b1 == b2)  
            System.out.println("Same");  
        else  
            System.out.println("Different");  
    }  
}
```

\$ java ExampleBooks6
Different



>> Equality and equivalence

```
public class ExampleBooks6a {  
    public static void main(String[] args) {  
  
        Book b1 = new Book("Java 8 Lambdas","Richard Warburton",168);  
        Book b2 = b1;  
  
        if (b1 == b2)  
            System.out.println("Same");  
        else  
            System.out.println("Different");  
    }  
}
```



```
$ java ExampleBooks6a  
Same
```





Static instance variables

- ✓ Class variables are fields that **belong to the class** and do not exist in each instance.
- ✓ There is always **only one copy** of this data field, independently of the number of the instances that were created.



>> Static instance variables

```
public class Book {  
    ...  
    static String location;  
    ...  
    public void setLocation(String name) {  
        location = name;  
    }  
    public String getLocation() {  
        return location;  
    }  
}
```

Location of book b1: Kampar
Location of book b2: Kampar

```
Book b1,b2;  
b1 = new Book("Java 8 Lambdas","Richard Warburton",168);  
b2 = new Book("Java in a nutshell","David Flanagan",353);  
b1.setLocation("Kampar");  
System.out.println("Location of book b1: " + b1.getLocation());  
System.out.println("Location of book b2: " + b2.getLocation());
```





Static methods

- ✓ With the **same idea** of the static data members, it is possible to define class methods or static methods
- ✓ These methods **do not work** directly with instances but with the class
- ✓ Can access **only** static instance variables



>> Static methods

The method `getLocation()` is a good candidate to be defined as a **static** method

```
public class Book {  
    ...  
    static String location;  
    ...  
    public static String getLocation() {  
        return "Books are located in" + location;  
    }  
}
```

Book are located in: Kampar
Books are located in: Kampar

```
Book b1,b2;  
b1 = new Book("Java 8 Lambdas","Richard Warburton",168);  
b2 = new Book("Java in a nutshell","David Flanagan",353);  
b1.setLocation("Kampar");  
System.out.println(b2.getLocation());  
System.out.println(Book.getLocation());
```





Instance variables initialization

- ✓ All instance variables are **guaranteed** to have an initial value.
- ✓ The **value is 0** for basic types and null for references
- ✓ Instance variables can be **also** initialized by calling instance methods

>> Instance variables initialization

```
public class Values {  
    int x = 2;  
    int y;  
    float f = inverse(x);  
    String s;  
    Book b;  
    Values(String str) { s = str; }  
    public float inverse(int value) { return 1.0F / value; }  
    public void dump() { System.out.println("'" + x + "'," + y + "'," + f + "'," + s + "'," + b); }  
}
```

```
$ java InitialValues  
2, 0, 0.5, hello, null
```

```
public class InitialValues {  
    public static void main(String[] args) {  
        Values v = new Values("hello");  
        v.dump()  
    }  
}
```



>> initialization block

a block of code inside the body of a class, but outside any methods or constructors, used for **initialization**

```
public class Values2 {  
    private int x;  
    private static String s;  
    {  
        x = 123;  
    }  
    static {  
        s = "abc"  
    }  
}
```

instance initialization
block: executed once per
instance

static initialization
block: executed
once per class

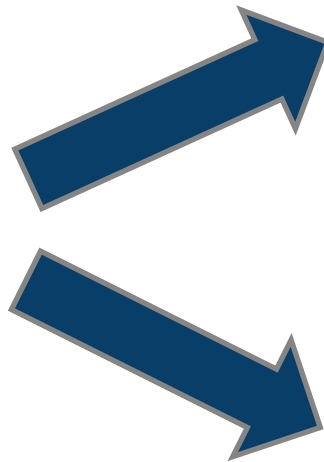
why should we use them?



>> The "this" keyword

The keyword **this**, when used inside a method, refers to the receiver object

It has **two main uses**:



to **return a reference** to the receiver object from a method

to **call constructors** from other constructors.

>> The "this" keyword

The class Book has two constructors

```
Book(String tit,String aut,int num) {  
    title = tit; author = aut; numberOfPages = num;  
    ISBN = "unknown";  
}  
Book(String tit,String aut,int num,String isbn) {  
    title = tit; author = aut; numberOfPages = num;  
    ISBN = isbn;  
}
```

```
Book(String tit,String aut,int num,String isbn) {  
    this(tit,aut,num);  
    ISBN = isbn;  
}
```

It is better to define the second constructor in terms of the first one

>> The "this" keyword

The method setLocation
in the previous Book class
could have been defined
as

```
public class Book {  
    ...  
    static String location;  
    ...  
    public Book setLocation(String name) {  
        location = name;  
        return this;  
    }  
}
```

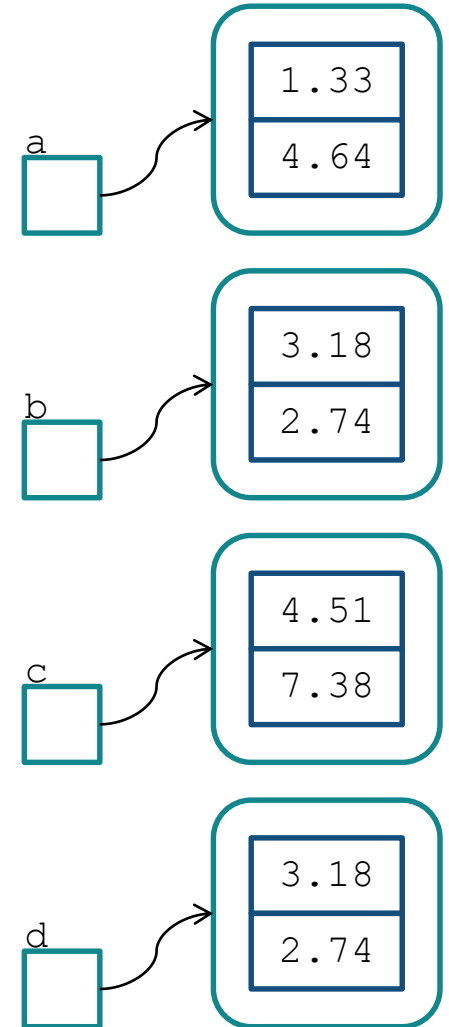
Operations can be
performed now in
"cascade" mode

```
Book b1,b2;  
b1 = new Book("Java 8 Lambdas","Richard Warburton",168);  
  
System.out.println("Initials: " + b1.setLocation("Kampar").getInitials());
```

>> A complete example

```
public class TestComplex {  
  
    public static void main(String[] args) {  
        Complex a = new Complex(1.33,4.64);  
        Complex b = new Complex(3.18,2.74);  
        Complex c = a.add(b);  
  
        System.out.println("c=a+b=" + c.getReal() +  
                           " " + c.getImaginary());  
  
        Complex d = c.sub(a);  
        System.out.println("d=c-a=" + d.getReal() +  
                           " " + d.getImaginary());  
    }  
}
```

```
$ java TestComplex  
c=a+b= 4.51 7.38 d=c-a= 3.18 2.74
```



>> A complete example

```
public class Complex {  
  
    double real;    // real part  
    double im;      // imaginary part  
  
    Complex(double real, double im) {  
        this.real = real;  
        this.im = im;  
    }  
  
    public double getReal() {  
        return real;  
    }  
  
    public double getImaginary() {  
        return im;  
    }  
}
```

```
a = new Complex(1.33, 4.64);
```

```
double realPart = a.getReal();
```

```
double imPart = a.getImaginary();
```

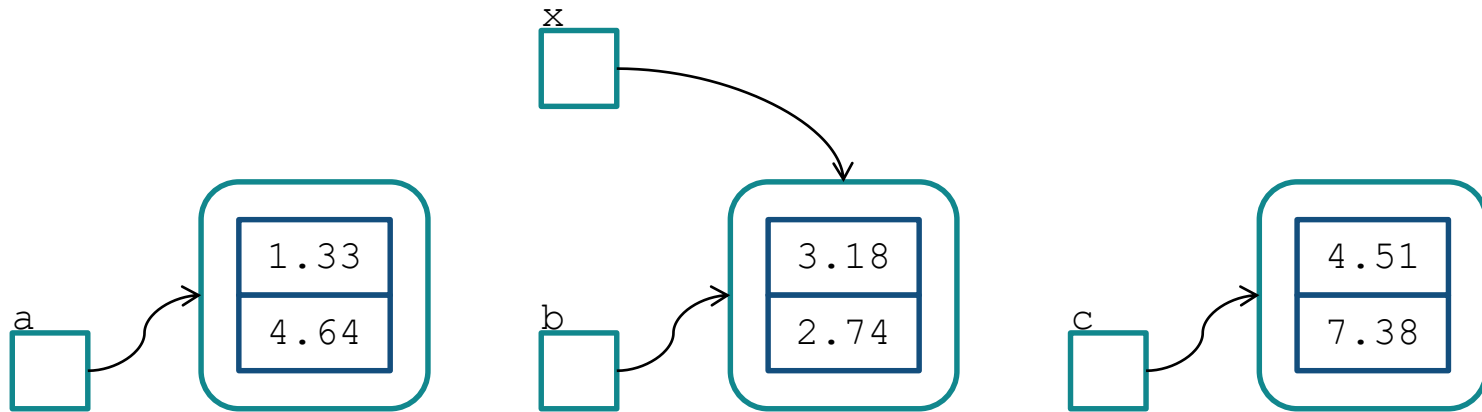


>> A complete example

```
public Complex add(Complex x) {  
    return new Complex(real + x.real, im + x.im);  
}
```

```
public Complex sub(Complex x) {  
    return new Complex(real - x.real, im - x.im);  
}
```

```
Complex c = a.add(b);
```

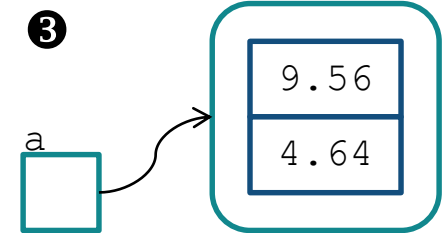
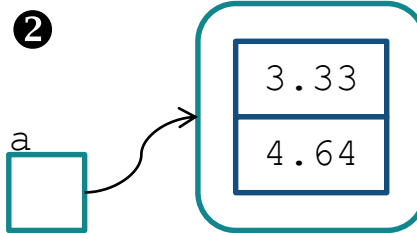
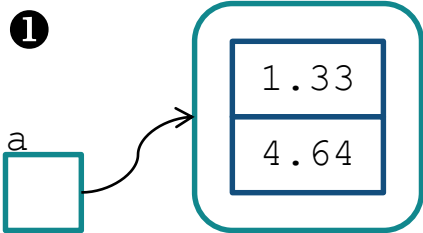


>> A complete example

```
public Complex addReal(double x) {  
    real += x;  
    return this;  
}  
Complex(Complex x) {  
    this(x.real,x.im);  
}
```

The method `addReal` increments just the real part of the receptor of the message with the value passed as argument

- ❶ `Complex a = new Complex(1.33,4.64);`
- ❷ `a.addReal(2.0);`
- ❸ `a.addReal(3.0).addReal(3.23);`



The new constructor can be used for assignment operations

```
Complex b = new Complex(a);
```

>> Type wrappers

- ✓ Primitive types are used for **performance** reasons, however many situations require an object
- ✓ Type wrappers are classes that **encapsulate primitive types** within an object
- ✓ There exist one type wrapper class **for each primitive type**



>> Boxing and Unboxing operations

boxing and unboxing operations are provided to **encapsulate/extract** the values to/from an object.

```
Integer iObject = new Integer(21);  
int i = iObject.intValue();
```

```
Integer iObject = 21;  
int i = iObject;
```

However, auto-boxing and auto-unboxing operations are provided to make **easier** to work with wrapped objects:

However... these operations add **overhead**, to be used only when required.

>> Type wrappers

There are many options to create and handle objects of wrapper classes

```
Integer iObject = new Integer(21);  
int i = iObject.intValue();  
  
Double dObject = new Double("121.1");  
double d = dObject.doubleValue();  
int x = dObject.intValue();
```

```
String str = "123.45";  
float f = Float.parseFloat(str);  
  
int i = Integer.MAX_VALUE;  
  
Character c = Character.valueOf('a');  
Boolean b = Character.isLowerCase(c);
```

and nice support from the classes



Thank you for your attention!



EXPLORE DESIGN PERFECTION

