R2.01 : Object-oriented development (OOD)

Coordinator : Isabelle Blasquez

My name: Cristina Onete

cristina.onete@gmail.com

Slides : https://www.onete.net/teaching.html

Interfaces

Interfaces in Java

- Inheritance highlights similarities in what two types of objects **are like**:
  - A wolf, a human, and a bear are all mammals
- Interfaces highlight similarities in how things **behave**:
  - Instruments and games are both **playable**
  - We can **buy** a wide variety of objects
- Our goal: buy things that can be bought, play things that can be played
- ... irrespective of their classes

Intro to interfaces

- A Java interface is a set of characteristics (behaviours)
  - Described as abstract methods
- Classes that behave according to an interface **implement** it
  - And must (if concrete) detail all the interface's abstract methods
- Note: **an interface is not a class**!

Example

- Take a class **Person**
  - Persons have a name and a budget
- We also include these classes: **Burger**, **Instrument**, **Piano**, **VideoGame**
- We want persons to :
  - **buy** all these things
  - **play** instruments and video games
  - **tune** **a piano**

- Interfaces group classes depending on their functionalities:
  - Things that are ***buyable :*** Burger, Instrument, Piano, VideoGame
  - Things that are ***playable*** : Instrument, Piano, VideoGame
  - Things that are ***tunable*** : Piano

Example : our three interfaces

```
interface Buyable {
    double getPrice();
}
```

```
interface Tunable {
    void tune();
}
```

```
interface Playable {
    void play(Person[] players);
}
```

keyword: interface

abstract methods

all methods by default public

Example: class implementing an interface

- A burger is buyable

Keyword: implements

Burger is a **concrete class**
It must detail the double getPrice() method in interface **Buyable**

```
public class Burger implements Buyable{
    private String type;
    private double price;

    public Burger(String type, double price){
        this.type = type;
        this.price = price;
    }

    @Override
    public double getPrice(){
        return this.price;
    }
}
```

```
interface Buyable {
    double getPrice();
}
```

Example: class implementing 2 interfaces

- A videoGame is buyable and playable

Comma between the 2 interfaces

Compulsory, as VideoGame implements Buyable

Compulsory as VideoGame implements Playable

```
public class VideoGame implements Buyable, Playable{
    private String name;
    private double price;

    public VideoGame(String name, double price){
        this.name = name;
        this.price = price;
    }
    @Override
    public double getPrice(){
        return this.price;
    }
    @ Override
    public double play(Person[] players){
        // code for this method
        // must end with return statement
    }
}
```

Example: abstract class Instrument

- Abstract class
- Can implement interfaces
- Must include all the methods in interfaces
  - Since it is abstract though, the method need not detail those methods
  - Can have both abstract and concrete methods
  - Not obliged to include any of the methods that will remain abstract from the interfaces

```
public abstract class Instrument implements Buyable,
Playable{
	private String type;
	private String brand;
	private double price;

	public Instrument(String type, String brand, double
price){
		this.type = type;
		this.brand = brand;
		this.price = price;
	}

	@Override
	public double getPrice(){
		return this.price;
	}

}
```

Example: inheritance and interfaces

- Pianos are instruments that are, in addition, tunable

extends precedes **implements**

Concrete class
Can detail/implement this method, inherited from class Instrument

Piano implements Tunable

```
public class Piano extends Instrument implements Tunable {
    private int numberOfPedals;

    // ... constructor

    @Overide
    public double play(Person[] players){
        // code of method play
    }

    @Override
    public double tune(){
        // code of method tune
    }
}
```

Classes and interfaces

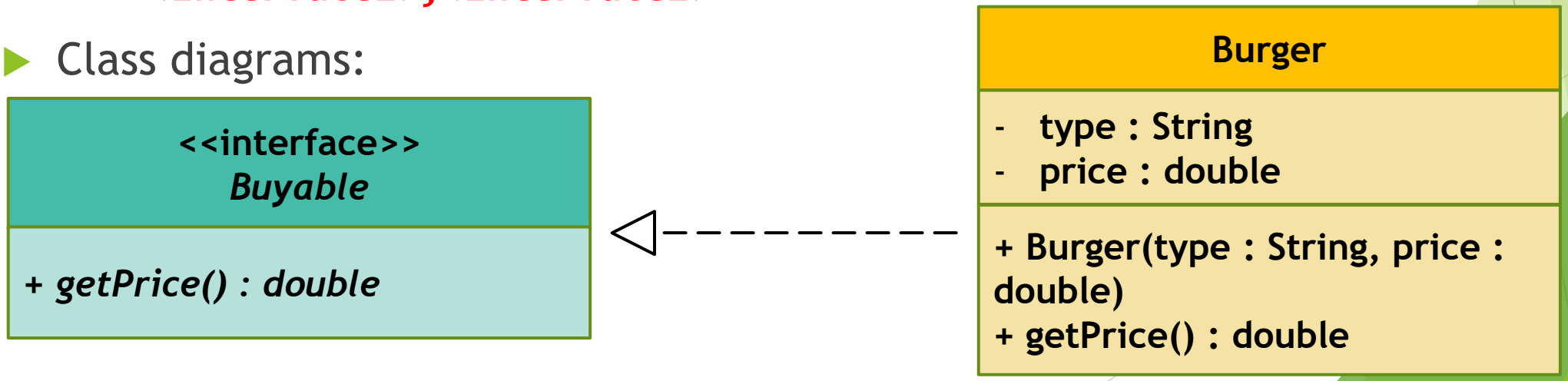
- A concrete class implementing an interface details **all its methods**
- An interface can be implemented by multiple classes
  - A class can also implement multiple interfaces

    ```
    class <ClassName> implements <Interface1>, <Interface2>
    ```

  - A class can inherit from at most one class & implement multiple interfaces

    ```
    class <ClassName> extends <Superclass> implements
    <Interface1>,<Interface2>
    ```

- Class diagrams:

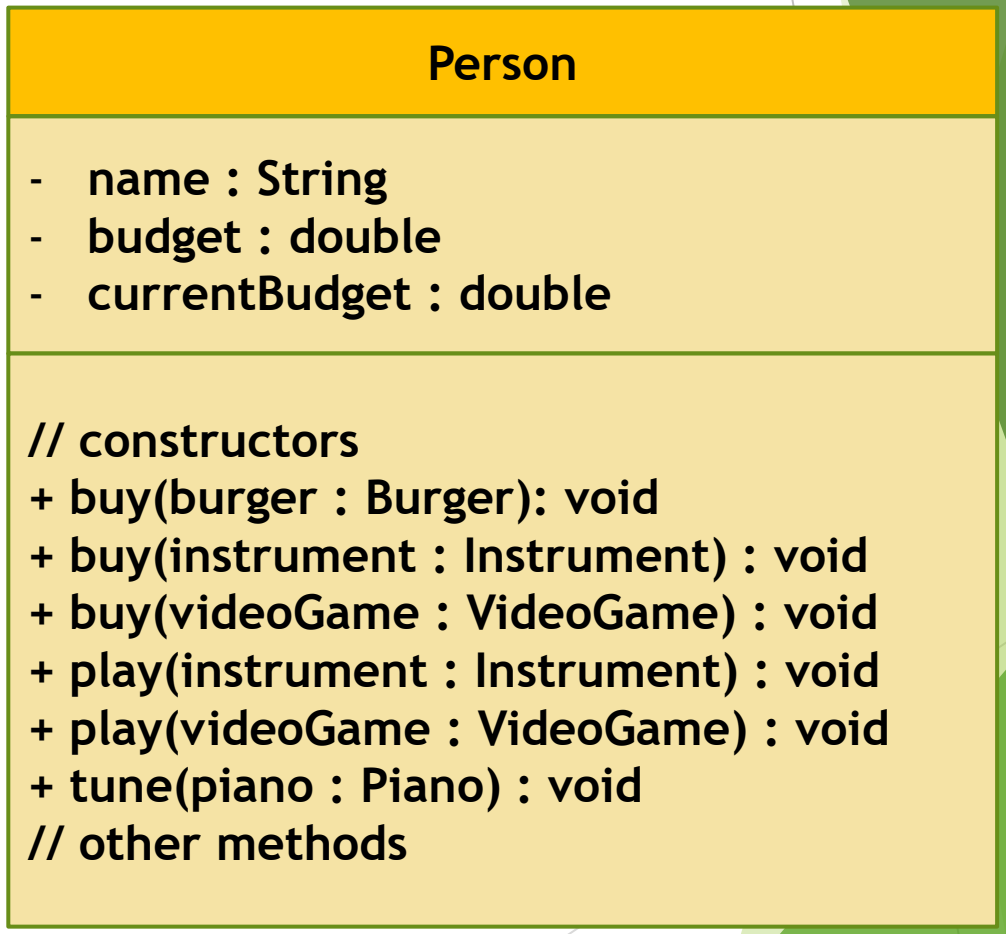


Why interfaces are useful

- Polymorphism can be used with interfaces
- A person must be able to buy: burgers, instruments, pianos, video games

- Without polymorphism: multiple methods:
  - `void buy(Burger burger)`
  - `void buy(Instrument instrument)`
  - `void buy(VideoGame videoGame)`

- More methods needed for playing (an instrument, a video game) or tuning (a piano)

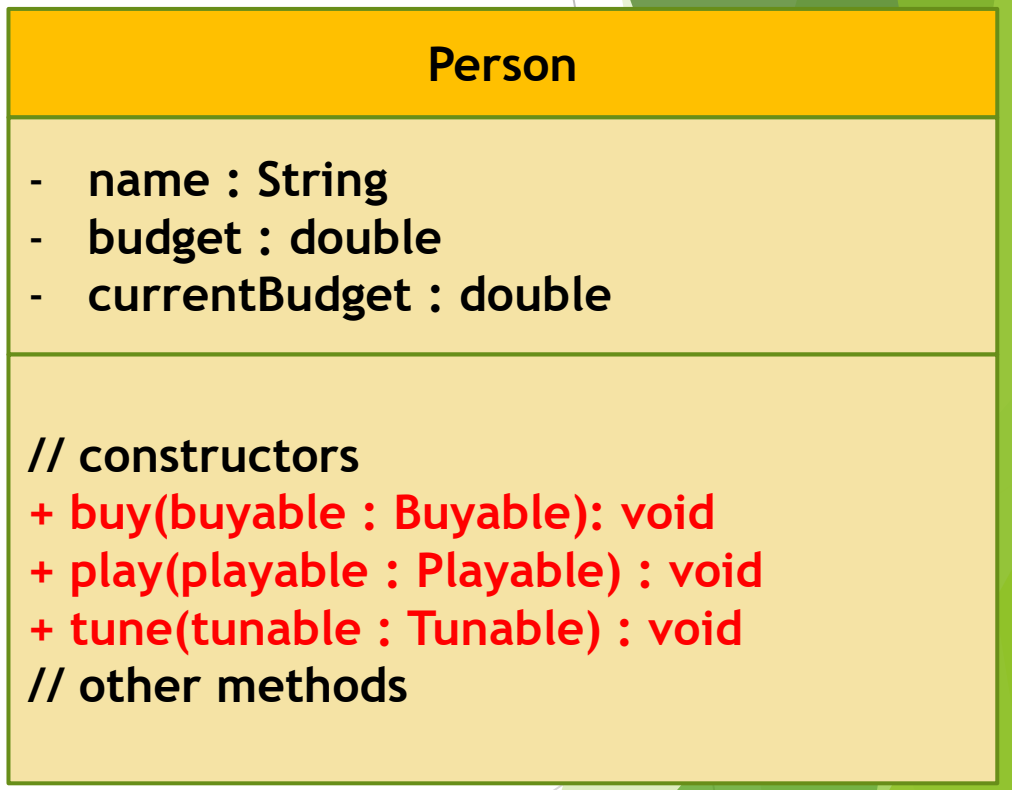


Using interfaces

- A person must be able to buy: burgers, instruments, pianos, video games

- Let's use our interfaces: Buyable, Tunable, Playable
  - **Single method** `void buy(Buyable)` includes all buyables: a burger, an instrument, a video game
  - We can be sure that the right classes **contain the right methods** (`tune` for tunables, `getPrice` for buyables...)

- Same for `void play(Playable)`, `void tune(Tunable)`

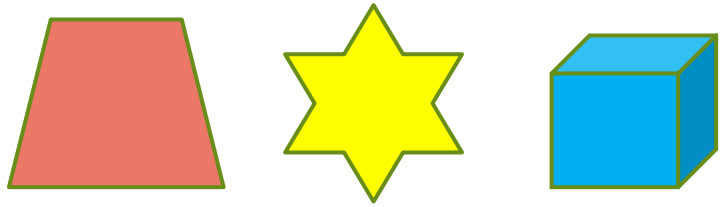


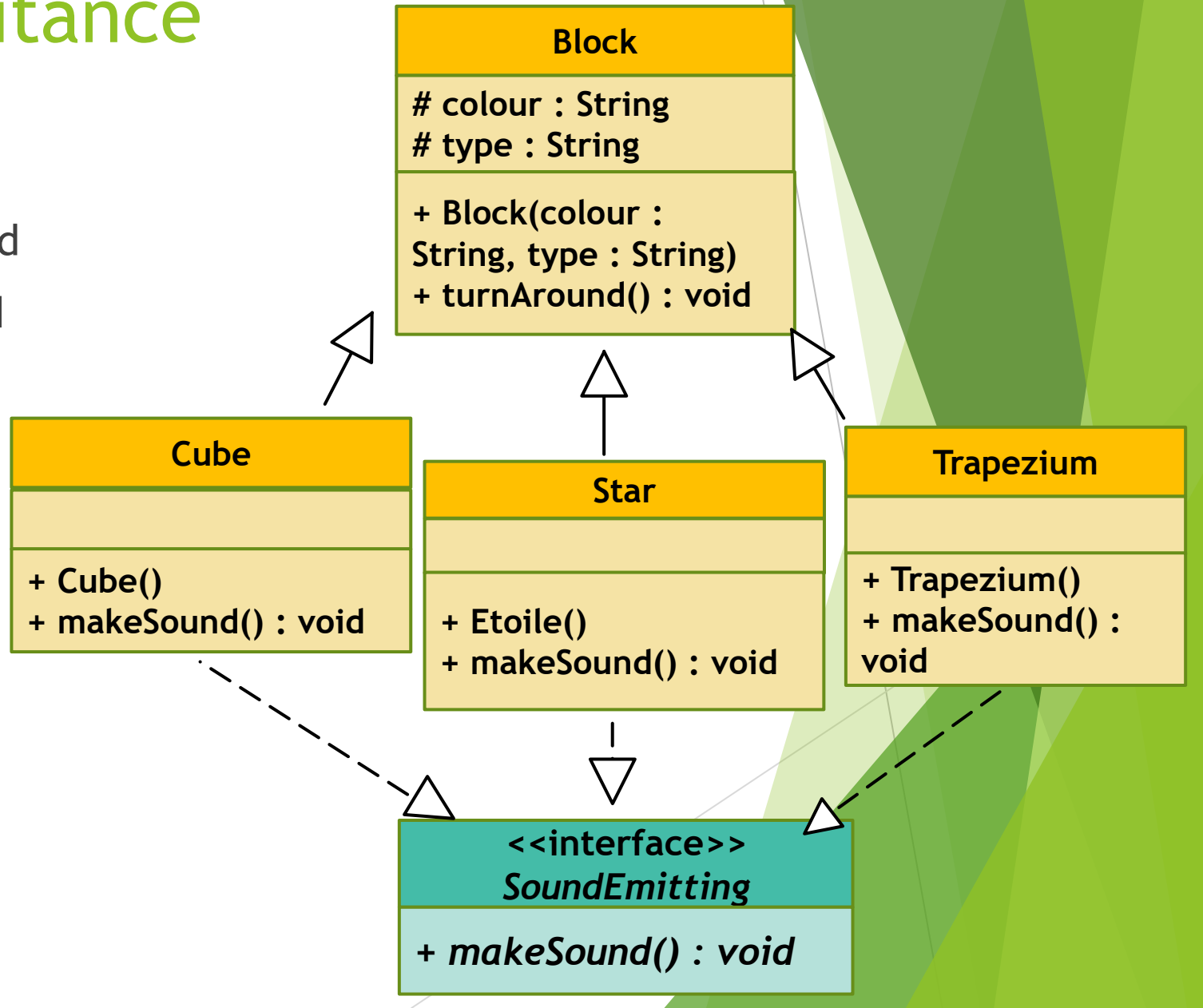
Interface vs. inheritance

Inheritance	Interface
• = "is a type of"	• = "works like a"
• Can inherit from **at most 1 superclass**	• Can implement **many interfaces**
• Keyword **`extends`**	• Keyword **`implements`**
• May contain concrete variables/methods	• Does not contain variables • Only abstract methods
• A subclass **may** redefine superclass methods (use of mention **`@Override`**)	• Concrete class **must** detail all the method of interfaces it implements

Interfaces and inheritance

- Recall our block example:
  - Each block makes different sound
  - Each block can be turned around





Exceptions

What is an exception?

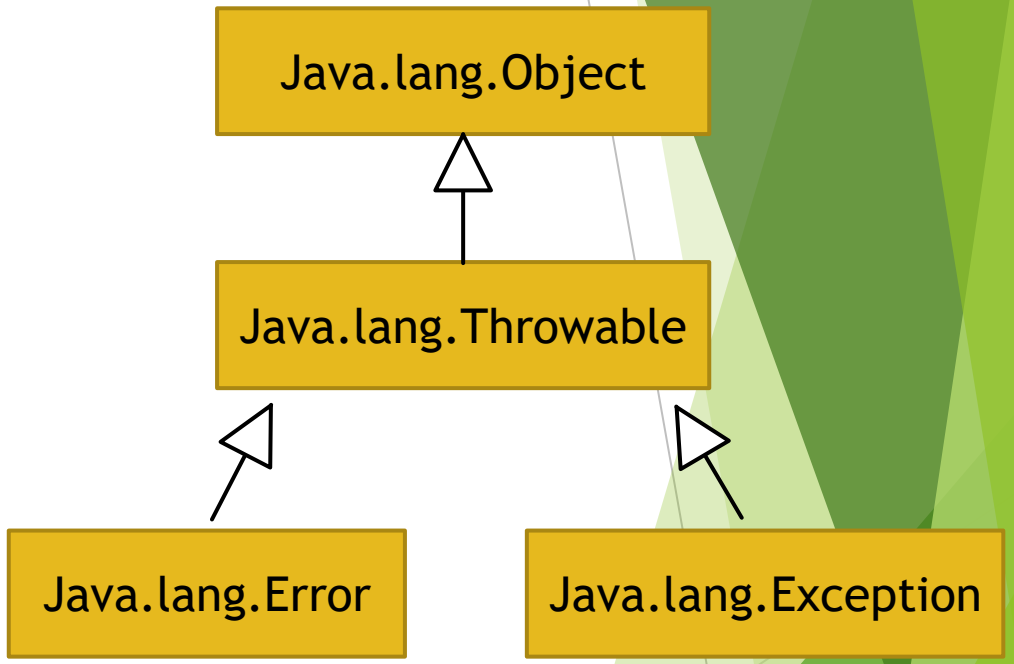
- An exception is an object (instance of a class modelling it)
- An exception is instantiated whenever some event interrupts the normal flow of an algorithm
- An exception can appear when a program halts, or upon some aberrant flow

Some errors can be anticipated and the program, modified to account for them

Others are harder to spot and can lead to premature halts

Different kinds of exceptions

- **Error**:
  - Terminal exception: halts the program entirely
  - Errors signal the existence of a serious flaw and we must let the program halt if they occur
  - VirtualMachineError, OutOfMemory, ...
- **Exception:**
  - Less serious than an error, but can still create some problems
  - Two kinds: exceptions at compilation, exceptions at runtime
  - IOException, SQLException, NullPointerException



Exceptions are thrown...

- If a technical failure interrupts the flow of the program
  - For instance if an index falls out of bounds (in an array)

    IndexOutOfBoundsException
  - Or if the Java virtual machine encounters some error:

    OutOfMemoryError
- If an application failure interrupts the flow of the program
  - For instance if the program tries to access an inexistent file:

    FileNotFoundException

Throwing exceptions

- Keyword: throw
- Raise exception in if statement:

```
if (age > 99){
    throw new RuntimeException("My students
can't be that old!");
}
```

- We can also do it in a "try-catch" block:

```
try{
    // code we ideally want to run
}
catch (FileNotFoundException e) {
    // in case the file doesn't exist, Java will raise an exception
}
```

What's the difference ?

Checked and unchecked exceptions

- Exceptions can be checked or unchecked
- Checked exceptions:
  - Exceptions which Java knows can occur in particular circumstances
    - For instance: when trying to open a file that doesn't exist, Java is able to throw a FileNotFoundException
  - The possibility of such an exception is anticipated by the compiler, which instructs the user to provide for it (typically try-catch)
- Unchecked exceptions:
  - Exceptions raised at runtime, can be caused by some errors or an abnormal program flow
  - Example: `NullPointerException`

Intermezzo: null

Null references

- NullPointerException:
  - Raised when Java stumbles on an object that does not exist
  - For instance the undefined (but existing) n-th element of an array

- How can we prevent this?
  - Check if the object exists: `if(object == null){...}`
  - a `try-catch` block around the code involving the exception

- Careful: null is not an object!
  - We do not use `object.equals(null)` or `null.equals(object)`
  - The latter even yields a `NullPointerException`!

End of Intermezzo

Checked exceptions

- Can be checked within a method directly (in a try-catch block)

```
public FileReader read(String filePath) {
    try{
        return (new FileReader(filePath));
    } catch (FileNotFoundException e){
        //treatment of exception
    }
}
```

- Or we can check it when we use the method (at call-time)

```
public FileReader read(String filePath) throws
FileNotFoundException{
    // ... some code
    return (new FileReader(filePath));
}
```

Try-catch blocks

- Try-catch blocks consist of :
  - An original `try` block, in which we write the code we would like to run
  - A first `catch` block indicating what to do when throwing a first exception
  - We can have multiple `catch` blocks
    - In that case, we are treating exceptions in reverse hierarchy
    - Subclasses before superclasses: `NullPointerException` before `RuntimeException`
- Optionally, we can have a `finally` block:
  - Always executed, even if an exception was thrown previously
  - Allows us to close open processes (for instance if a file is open)

Handy instructions

- System.err.println(String): a method that allows us to print (in red) a text that is meant to be printed when an exception is thrown
- printStackTrace(): method that can be run for any instance of a class that implements the interface Throwable; when called, this method tracks the cause of the error or exception that was thrown

Example: index out of bounds

Check out this code

MonException.java

```
1
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 3;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
12         System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
13
14     }
15
16     public int augmenter (int index) {
17         System.out.println("La valeur initiale de l'index est " + index);
18         index = index + 1;
19         System.out.println("L'index a ete augmente a " + index);
20         return index;
21     }
22
23     public int doubler (int valeur) {
24         System.out.println("Nous allons doubler la valeur " +valeur);
25         valeur = valeur * valeur;
26         System.out.println("Valeur doublee " + valeur);
27         return valeur;
28     }
29 }
30
```

Lors d'une execution normale

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
La deuxieme composante du tableau est 100
```

Let's add an arror

ErrorProgram/src/MonException.java

```
1
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 1;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
12         System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
13
14     }
15
```

changed size of array

Problems Javadoc Declaration Console

<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_

```
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
Exception in thread "main" L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
java.lang.ArrayIndexOutOfBoundsException: 1
        at MonException.main(MonException.java:11)
```

Throw exception

```
1
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 1;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         try {
12             monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13             System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14         }
15         catch(IndexOutOfBoundsException e) {
16             System.out.println("Taille du tableau depassee !");
17         }
18     }
19
20     public int augmenter (int index) {
21         System.out.println("La valeur initiale de l'index est " + index);
22         index = index + 1:
```

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Jav
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
```

try-catch block

Java looks out for an IndexOutOfBounds exception

exception is thrown

Tracking down the source of the error

```
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 1;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         try {
12             monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13             System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14         }
15         catch(IndexOutOfBoundsException e) {
16             System.err.println("Taille du tableau depassee !");
17             e.printStackTrace();
18         }
19         finally {
20             System.out.println("Dans le bloc finally");
21         }
22     }
```

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_141\bin\javaw.exe (Jan 17, 2018, 1:43:26 PM)
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
java.lang.ArrayIndexOutOfBoundsException: 1
        at MonException.main(MonException.java:12)
Dans le bloc finally
```

System.out.println replaced by System.err.println

tracks down error

Tracking down the source of the error

```
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 1;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         try {
12             monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13             System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14         }
15         catch(IndexOutOfBoundsException e) {
16             System.err.println("Taille du tableau depassee !");
17             e.printStackTrace();
18         }
19         finally {
20             System.out.println("Dans le bloc finally");
21         }
22     }
```

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_141\bin\javaw.exe (Jan 17, 2018, 1:43:26 PM)
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
java.lang.ArrayIndexOutOfBoundsException: 1
        at MonException.main(MonException.java:12)
Dans le bloc finally
```

A finally block

Making our own exceptions

Catching new errors

- Remember this example?

```
if (age > 99){
    throw new RuntimeException("My students
can't be that old!");
}
```

- Here, we are throwing an exception which is not normally speaking an exception: an age larger than 99

- In this example we throw a generic RuntimeException...
  - Could we be more specific and create our own exception ?

Create a new exception

- We can, in fact, create new exceptions in Java, which will inherit from superclasses of exceptions:
  - Checked exceptions inherit from class Exception
  - Unchecked exceptions inherit from class RuntimeException

- The new exception will be a new class, with attributes and methods

GP1: Do not create a new Java exception if you can use existing ones!

Example: StudentTooOld

```
public class StudentTooOldException extends RuntimeException{
    // customized constructor using the superclass constructor
    public StudentTooOld(String m){
        super(m); // we will use the constructor of RuntimeException
    }
}
```

```
public class MainClass{
    public static void main(String[] args) {
        Student anneLeclerc = new Student();
        if (anneLeclerc.getAge() > 99){
            throw new StudentTooOldException("This student is too old!");
        }
    }
 }
```

Other good practices

- Crucial to catch the exceptions potentially touching our code

GP2: Use the best approximation you can have of your exception (subclass rather than Exception/RuntimeException directly)

- Try-catch-finally blocks have special structures

GP3: Do not use catch blocks as regular else blocks in the code!

GP4 : Never raise an exception using a `return` statement

Any questions