2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

Java Arrays, Collections, Packages & Exceptions

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- Packages
- Arrays & ArrayList
- Exceptions

Packages

Arrays & ArrayList

Exceptions

Packages

PACKAGES



- Code files can be organised into packages
- This provides a better structure
- If no packages are used, all classes reside in the default package

PACKAGES

```
package java_demo;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<Integer>();

        // ...
    }
}
```

- Every class starts with a *package declaration*
  - This is the first line of a class file
  - It declares the package this class should live in
- The compiler will throw an error if the class is not in the right location per the declaration

PACKAGES

```
package java_demo;

import java.util.ArrayList;
import java.util.List;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<Integer>();

        // ...
    }
}
```

- You can use a class from another package by importing it
  - Your IDE helps with this!
- Whether the class is built-in or self-written makes no difference

PACKAGES

```
public class Application {

    public int number1 = 1;
    private int number2 = 2;
    protected int number3 = 3;
    int number4 = 4; // No modifier -> package private!

    public static void main(String[] args) {
        // ...
    }
}
```

- Using packages enables a fourth access modifier
  - Public, visible everywhere
  - Private, visible in the defining class
  - Protected, visible in the defining class and subclasses
  - Package private, visible everywhere in the package of the defining class

Packages

Arrays & ArrayList

Exceptions

Arrays & ArrayList

ARRAYS

```
public static void main(String[] args) {
    int number1 = 2;
    int number2 = -51;
    int number3 = 88;

    int[] numbers = new int[3];
    numbers[0] = 2;
    numbers[1] = -51;
    numbers[2] = 88;
}
```

- An array is **an object** in which you can store multiple variables of the same type
- When creating an array, you specify the length, after creation the length of the array is fixed and cannot be changed anymore
- A position of an array is called an *index*, the positions are called *indices*
- You work with an array by passing an index in operations

ARRAYS

```
public static void main(String[] args) {
    int[] numbers = new int[3];
    numbers[0] = 2;
    numbers[1] = -51;
    numbers[2] = 88;

    numbers[1] = 23;
    int number = numbers[2];
}
```

- A position in an array behaves like a normal variable
- You can change the value, assign it to another variable, ...

ARRAYS

```
public static void main(String[] args) {
    int[] numbers = new int[3];
    numbers[0] = 2;
    numbers[1] = -51;
    numbers[2] = 88;

    System.out.println(numbers[-1]); // Crash!!
    System.out.println(numbers[4]);  // Crash!!
}
```

- You can reach the first position of the array using index 0
- The last position of the array has as index "length array - 1"
- Querying a position outside the array results in an *ArrayIndexOutOfBoundsException*

ARRAYS

```
public static void main(String[] args) {
    int[] numbers = new int[3];
    numbers[0] = 2;
    numbers[1] = -51;
    numbers[2] = 88;

    for (int i = 0; i < numbers.length; i++) {
        System.out.println(numbers[i]);
    }

    // Shorter syntax for the same for-loop
    for (int j : numbers) {
        System.out.println(j);
    }
}
```

- The length of an array can be retrieved via the "length" field
- To loop through an array, a loop is often used
- Watch out for *ArrayIndexOutOfBoundsException*

ARRAYS

```
public static void main(String[] args) {
    int[] getallen = new int[2];
    getallen[0] = 2;
    getallen[1] = -51;

    verhoogMet3(getallen);

    for (int i : getallen) {
        System.out.println(i);
    }
}

public static void verhoogMet3(int[] getallen) {
    for (int i = 0; i < getallen.length; i++) {
        getallen[i] = getallen[i] + 3;
    }
}
```

- An array is an object
- Thus, when used as a parameter, a reference is copied into the method

EXERCISE @CLASS

- exercise1



LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(1);
    numbers.add(27);

    for (int number : numbers) {
        System.out.println(number);
    }
}
```

- A list is a collection of data of the same type with a fixed order and a dynamic size
- List is the interface
- ArrayList is the most used class implementing this interface

LIST AND ARRAYLIST

```
import java.util.ArrayList;
import java.util.List;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<Integer>();

        // ...
    }
}
```

- List and ArrayList are built into Java, to use them in your program you need to import them

LIST AND ARRAYLIST

```
import java.util.ArrayList;
import java.util.List;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<Integer>();

        // ...
    }
}
```

- List and ArrayList make use of *generics*
- This is an advanced technique to ensure that the same classes and interfaces can work with multiple types
- When creating a List, you need to specify the type of the content between "<>"

LIST AND ARRAYLIST

```
import java.util.ArrayList;
import java.util.List;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();

        // ...
    }
}
```

- The content type must only be indicated in the variable type
- This information may be repeated when calling the constructor, but this is optional

LIST AND ARRAYLIST

```
import java.util.ArrayList;
import java.util.List;

public class Application {

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();

        // ...
    }
}
```

- Instead of "int", "Integer" is used as type
- Each primitive type has a class as a counterpart (these are called *wrapper classes*)
- The content of an Integer class corresponds to a primitive int
- Wrapper classes are used to ensure compatibility between primitive types and objects where necessary

LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    System.out.println(numbers.get(0));
    System.out.println(numbers.size());
}
```

- Adding data can be done via the method ".add()"
- Data retrieval can be done via the method ".get()"
  - The index runs between 0 and "length list - 1"
- Size retrieval can be done via the method ".size()"
- The ArrayList is automatically rescaled in memory as it grows

LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    System.out.println(numbers.get(1)); // Crash
}
```

- Retrieving data from an invalid index position results in IndexOutOfBoundsException

LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(37);
    numbers.add(52);
    numbers.add(-18);

    for (int i = 0; i < numbers.size(); i++) {
        System.out.println(numbers.get(i));
    }

    for (int number : numbers) {
        System.out.println(number);
    }
}
```

- You can walk through a list using a loop
- Easiest option is often to use the short syntax for a for loop (*for-each*)

LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(37);
    numbers.add(52);
    numbers.add(-18);

    System.out.println(numbers.indexOf(-18));
    System.out.println(numbers.contains(52));
    System.out.println(numbers.remove(1));
    System.out.println(numbers.contains(52));
}
```

- ".remove()" removes an element via index or value
- ".contains()" checks whether a value is present
- ".indexOf()" returns the first index of a value

EXERCISE @CLASS

- exercise4



Packages

Arrays & ArrayList

Exceptions

Exceptions

- Exceptions
- Unchecked Exceptions

EXCEPTIONS

- We have already seen some situations that cause a program to crash
  - Calling a method on a variable containing null
  - Indexing an array at an invalid position
  - Indexing an ArrayList at an invalid position
  - Converting a String containing no number to an int
  - …
- Technically speaking, these situations lead to *exceptions*
- An exception only leads to a crash if you don't *handle it*

EXCEPTIONS

```
public static void main(String[] args) {
    try {
        // code that may cause an exception
    } catch (Exception e) {
        // code that executes if the exception happened
    }
}
```

- Code that can causes an exception can be wrapped in a *try catch* construction

EXCEPTIONS

```
public static void main(String[] args) {
    Scanner reader = new Scanner(System.in);

    System.out.print("Enter a number: ");
    int number = -1;

    try {
        number = Integer.parseInt(reader.nextLine());
        System.out.println("Well done!");
    } catch (Exception e) {
        System.out.println("You did not enter a number!");
    }
}
```

- We talk about *catching* and handling exceptions
- In the catch block, you can use the parameter provided to get to the details of the exception

EXCEPTIONS

```
public static void main(String[] args) {
    try {
        // Lots of code
    } catch (NullPointerException e) {
        System.out.println("Woops");
    }
}
```

- You do not want to catch every exception
- For example, it is bad practice to wrap NullPointerExceptions in try catch (instead improve your code so it does not throw NullPointerExceptions)

EXCEPTIONS

```
public static void main(String[] args) {
    try {
        Integer.parseInt("Hello");

        // ... More code ...
    } catch (Exception e) {
        System.out.println("Woops");
    }
}
```

```
public static void main(String[] args) {
    try {
        Integer.parseInt("Hello");

        // ... More code ...
    } catch (NumberFormatException e) {
        System.out.println("Woops");
    }
}
```

- In your catch, you can specify any class that inherits from Exception
- Exceptions other than those you specify are not handled

EXCEPTIONS

```
public static void main(String[] args) {
    try {
        Integer.parseInt("Hello");

        // ... More code ...
    } catch (NumberFormatException e) {
        System.out.println("Woops");
    } finally {
        System.out.println("Always executed");
    }
}
```

- A try catch block can be supplemented with *finally*
- Code in the finally block is always executed, if the exception was thrown as well as if it was not thrown

EXCEPTIONS

```
public static void main(String[] args) {
    readNumber();
}

private static void readNumber() throws Exception {
    Integer.parseInt("Hello");

    // ... More code ...
}
```

- If you don't handle an exception with catch you can add it to your method signature to *throw* it
- It is then up to the calling method to handle the exception or to throw it again

EXCEPTIONS

```
public static void main(String[] args) {
    readNumber();
}

private static void readNumber() throws Exception {
    Integer.parseInt("Hello");

    // ... More code ...
}
```

- If an exception is not caught in your own program and reaches the JVM your program crashes
- The JVM stops your program and prints the exception that led to the crash

EXCEPTIONS

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hallo"
        at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:661)
        at java.base/java.lang.Integer.parseInt(Integer.java:777)
        at excepties.Applicatie.leesNummer(Applicatie.java:12)
        at excepties.Applicatie.main(Applicatie.java:8)
```

- The exception crashing the program leads to a printed stacktrace
- This lists all methods from the crashed method to the starting point of your program (this is called a *stacktrace*)

EXCEPTIONS

```
private static void readNumber() throws OwnException {
    Integer.parseInt("Hello");

    // ... More code ...
}
```

```
public class OwnException extends Exception {

    // ...
}
```

- You can define your own exceptions by writing a class that inherits from Exception
- Can be useful to simplify error-handling in complex programs

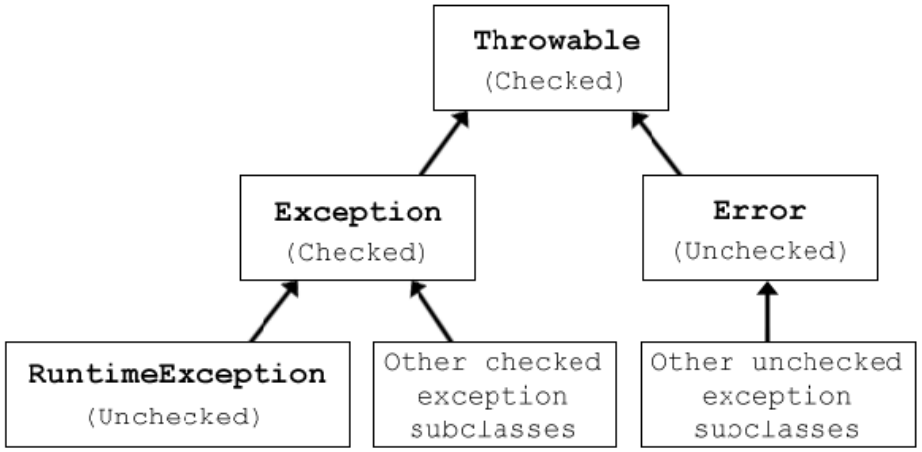
EXCEPTIONS

```
public static int calcLength(String text) {
    if (text == null) {
        throw new IllegalArgumentException(
            "Argument should not be null");
    }

    return text.length();
}
```

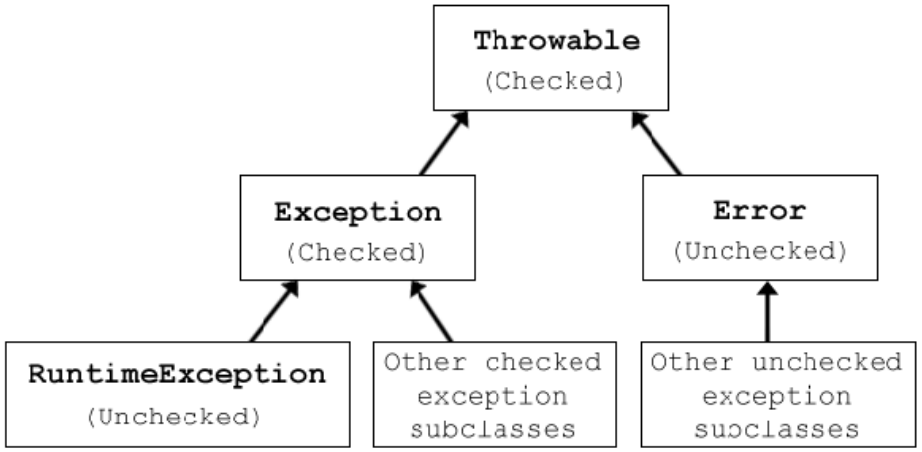
- You can throw an exception yourself via the keyword *throw*
  - Technically speaking, you are throwing a newly created object of a certain exception class

UNCHECKED EXCEPTIONS



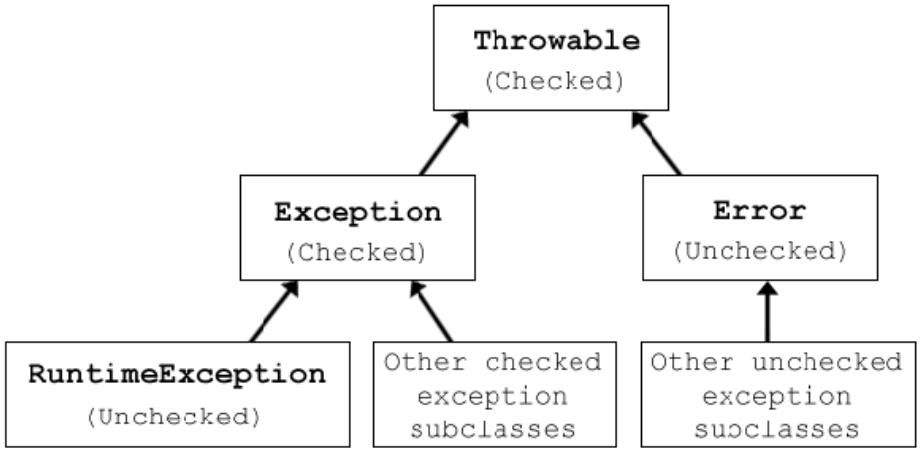
- Exceptions actually come in two types, checked and unchecked exceptions
- Both are very similar
- There is one big difference
  - Checked exceptions should be added to the method signature if you don't catch them
  - Unchecked exceptions do not need to be added and do not need to be catched

UNCHECKED EXCEPTIONS



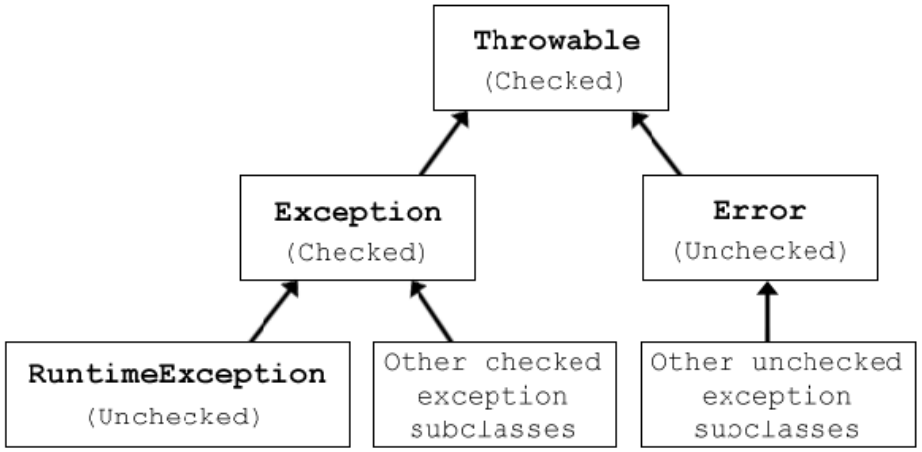
- For example, NullPointerException inherits from RuntimeException, so not every method needs to explicitly state how it will handle a NullPointerException

UNCHECKED EXCEPTIONS



- Does your own exception inherit from Exception -> checked
- Does your own exception inherit from RuntimeException -> unchecked

THROWABLE AND ERROR



- In your own code, only catch “Exception” and subclasses
- Never catch “Error” (and therefore also not “Throwable”)
  - Errors are fundamental problems that should cause your program to crash
  - For example, a lack of memory leads to an Error
  - Catching errors leads to extremely unstable behavior

EXERCISE @CLASS

- exercise5
- exercise6



EXERCISE @HOME

- exercise2
- exercise3



