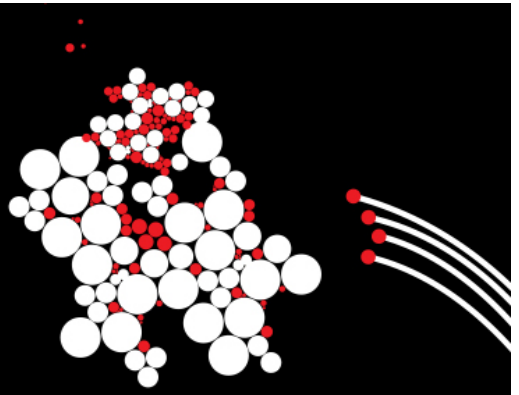
UNIVERSITY OF TWENTE.

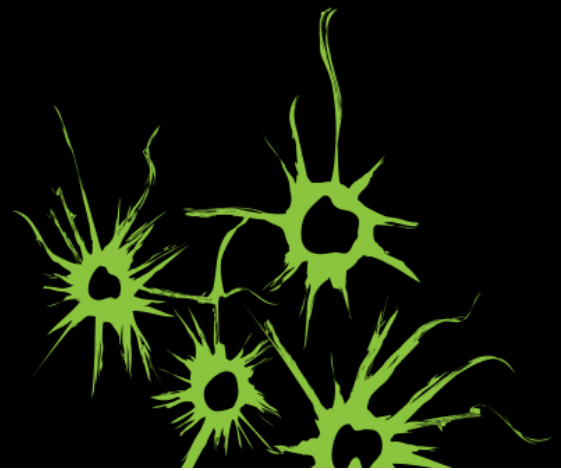


P5.1: EXCEPTIONS

LUÍS FERREIRA PIRES

201700117-1B MODULE 2: SOFTWARE SYSTEMS

10 DECEMBER 2019



PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Exceptions Stream I/O and MVC Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

RUNTIME EXCEPTIONS

WHAT CAN GO WRONG IN THE CODE BELOW?

```
int[] a =                  // some initial value
String s =                 // some initial value
Collection<Integer> c =    // some initial value

int i = Integer.parseInt(s);

a[i] = s.length();

double d = 10.0/i;

List<Integer> l = (List<Integer>) c;
```

`NumberFormatException`
if `s` is not formatted as an `int`

`IndexOutOfBoundsException`
if `i` is negative or `>= a.length`;
`NullPointerException` if `s` is null

`ArithmeticException` if `i` is zero

`ClassCastException` if `c` is not a `List`

- These are so-called runtime exceptions
- `NullPointerException`, `IndexOutOfBoundsException`, etc. are classes!

HANDLING EXCEPTIONS

TRY-CATCH

- Surround code that can generate exception with a `try`-block
- Define `catch`-block for exceptions you want to handle

```
try {
   int i = Integer.parseInt(s);
   a[i] = s.length();
   double d = 10.0/i;
   List l = (List) c;
} catch (NumberFormatException e) {
   System.out.println(s + "is not a number");
}
```

If exception occurs, program execution jumps immediately to `catch`-block

Exception object must be declared!

Code to deal with exception defined in the `catch`-block

HANDLING EXCEPTIONS

TRY-CATCH

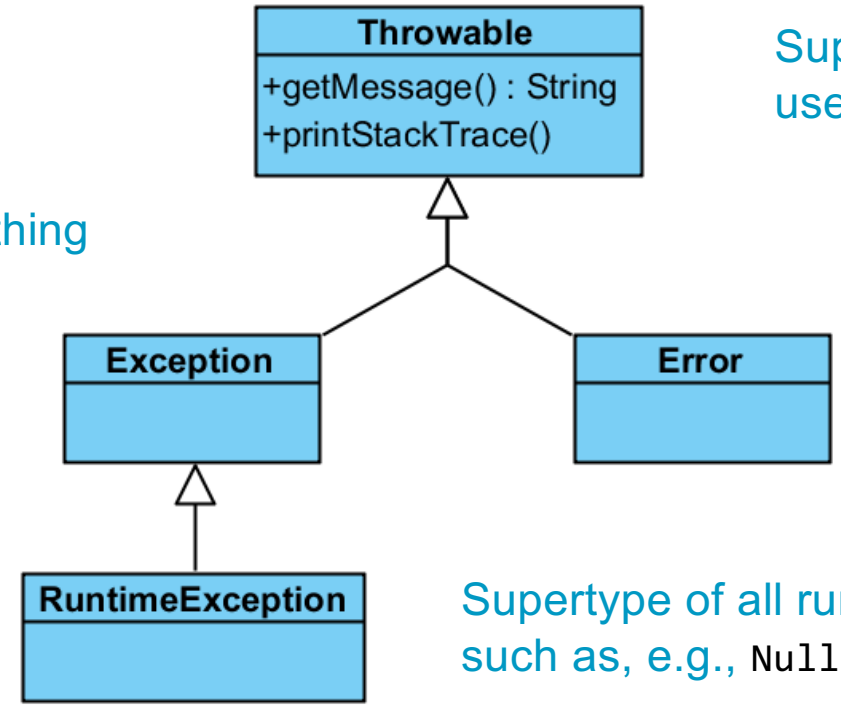
```
try {
   int i = Integer.parseInt(s);
   a[i] = s.length();
   double d = 10.0/i;
   List l = (List) c;
} catch (NumberFormatException e) {
   System.out.println(s + "is not a number");
} catch (ArithmeticException e) {
   System.out.println("Cannot calculate i, as s is zero");
} catch (IndexOutOfBoundsException | NullPointerException e) {
   e.printStackTrace(); // Prints error message to System.err
}
```

Multiple `catch`-blocks may be defined

Exceptions can be handled separately or combined in a single `catch`-block

EXCEPTION HIERARCHY

LEVELS OF EXCEPTIONS



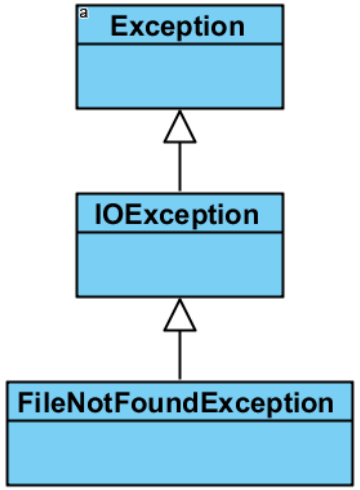
Supertype of everything that can be thrown

Supertype of system-defined or used-defined exceptions

Supertype of system errors, typically not repairable
e.g., `OutOfMemoryError`

Supertype of all runtime exceptions, such as, e.g., `NullPointerException`

CHECKED EXCEPTIONS



- Exceptions that must be caught (unlike `RuntimeException`)
- Typically occur when doing I/O operations!
- For example, code below gives compilation errors

```
String name = System.console().readLine();
BufferedReader r = new BufferedReader(new FileReader(name));
System.out.println("First line: " + r.readLine());
```

Only works with command line not IDE (started with `java`, not `javaw`)

Why?

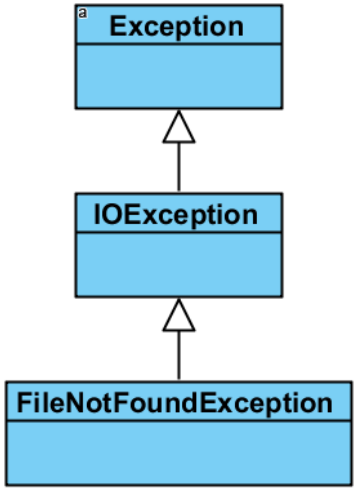
CHECKED EXCEPTIONS

POSSIBLE SOLUTION

```
BufferedReader r = null;
while (r == null) {
    String name = System.console().readLine();
    try {
        r = new BufferedReader(new FileReader(name));
        System.out.println("First line: " + r.readLine());
    } catch (IOException e) {
        System.out.printf("File %s not suitable%n", name);
    }
}
```

Two exceptions potentially thrown!

Catches the most general exception!



PASSING UP EXCEPTIONS

METHOD DECLARATION

- Rather catching exceptions can be passed up
- Add `throws` clause to a method declaration

Inform that method throws `IOException`

```
public static String getLine() throws IOException {
   String name = System.console().readLine("Enter filename: ");
   BufferedReader r = new BufferedReader(new FileReader(name));
   return r.readLine();
}
```

This happens if exceptions are thrown by the `FileReader` constructor or `readLine`

PASSING UP EXCEPTIONS

CATCHING EXCEPTION

- Exception then is handled by the caller

```
BufferedReader r = null;
while (r == null) {
   try {
      System.out.println(getLine());
   } catch (IOException e) {
      System.console().printf("File %s not suitable%n", name);
   }
}
```

THROWING EXCEPTIONS

THROW STATEMENT

```
void setWord(String old, String newWord) throws Exception {
   if (!testWord(old)) {
      throw new Exception("Old password wrong");
   }
   if (!acceptable(newWord)) {
      throw new Exception("New password not acceptable");
   }
   setWord(newWord);
}
```

Generic `Exception` object!

DEFINING EXCEPTIONS

ALTERNATIVE TO GENERIC EXCEPTION CLASS

```
public class PasswordException extends Exception {
    public PasswordException(String message) {
        super(message);
    }
}
```

Define `PasswordException` class by extending `Exception`

```
void setWord(String old, String newWord) throws PasswordException {
    if (!testWord(old)) {
        throw new PasswordException("Old password wrong");
    }
    if (!acceptable(newWord)) {
        throw new PasswordException("New password not acceptable");
    }
    setWord(newWord);
}
```

Create `PasswordException` with more specific `message`

- Advantage: better readability and maintainability

CLOSING OF RESOURCES

CLEANING UP

- Some resource need to be closed after they are used, like network connections and I/O-based objects
  - Not closing them runs the risk of resource leaks
  - For example, a file may not be renamed or deleted while open
- However, this is not a problem with regular objects without references
  - Java has a garbage collector

FINALLY STATEMENT

BLOCK THAT IS ALWAYS EXECUTED

- Meant for closing network connections or streams

```
try {
    int data = 25 / 0;
    System.out.println(data);
} catch (RuntimeException e) {
    System.out.println(e);
} finally {
    System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
```

```
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

TRY WITH RESOURCES

- Requires `AutoCloseable` objects

```
try (<declare AutoClosable objects>)
{
   // do some stuff
} catch (<exceptions>) {
   // handling of exceptions
}
```

```
String name = System.console().readLine("Enter filename: ");
try (BufferedReader r = new BufferedReader(new FileReader(name))) {
   System.out.println("First line: " + r.readLine());
} catch (IOException e) {
   System.console().printf("File %s has a problem%n", name);
}
```

- Afterwards, `close()` is automatically called on all declared objects (`FileReader` and `BufferedReader`!)

DON'TS IN EXCEPTIONS

WORST PRACTICES

- Do not construct instances of `Exception`
  - Define and use proper subclasses (for maintainability)
- Do not throw `RuntimeException` yourself
  - Bypasses checking mechanism, let these for the JVM
- Never catch `Exception`
  - Too generic, also catches unexpected ones (`RuntimeExceptions`)
  - Catch the most specific exceptions

TAKE HOME MESSAGES

- `Exception`s are classes that have a hierarchy
- `RuntimeException` indicates occurrence of a program execution error
- `Exception`s may occur and be caught when executing statements in a `try`-block and should be handled in `catch`-blocks
- `Exception`s can be passed up (`throws`-clause)
- Programmers can define their own `Exception` classes
- (User-defined) `Exceptions` can be thrown at some specific pieces of code (`throw`-statement)