University of Bamberg

Java-Tutorial for ISSS-Students

Chapter 9: Exceptions

Chapter 9: Exceptions

1. Exceptions
2. Throwing Exceptions
3. Catching Exceptions
4. Checked vs. Unchecked Exceptions
5. Defensive Programming
6. Errors
7. Excursus: User input

Warm-up: the party scenario

- Create the package *chapter9* with the following classes:
  - Interface *Location*
  - Class *Company* (implements *Location*) with the attribute *name* (String) + constructor, getters, setters
  - Class *Person* with the attributes *name* (String), *employment* (Company) and *salary* (int) + constructor, getters, setters
  - Interface *Party* with the methods *setLocation(Location location)*, *participate(Person person)* and *showGuestList()*
  - Class *CompanyParty* (implements *Party*) with the attributes *location* (Location) and *guests* (List<Person>) + constructor, getters, setters – implement the method *participate(Person person)* by adding the person to the List of guests

Exceptions

- *Exceptions* offer the possibility to manipulate the control flow of programs in case of unexpected events
- A lot of exceptions are predefined in Java – additionally it is possible to implement your own exceptions by extending the class *java.lang.Exception*:
  - *public class ExceptionName extends Exception { … }*
- You can use the constructor of the superclass to define a message that states why the exception occurs:
  - *public ExceptionName() { super("Reason for exception"); }*

Throwing Exceptions

- The class *java.lang.Exception* is a subclass of the class *java.lang.Throwable* – all objects of this class can be handed from the method/constructor where the problem occurs to the method/constructor calling this method – this is called *throwing* the object:
  - *modifiers returnValue methodName(parameters) throws ExceptionName { … }*
- Methods can throw more than one exception:
  - *modifiers returnValue methodName(parameters) throws ExceptionName1, ExceptionName2, …, ExceptionNameX { … }*

Catching Exceptions

- To handle exceptions successfully, you have to catch them with a *try-catch*-block:
  - *try { code where an exception might occur }*
    catch (ExceptionName e) { code to handle the exception }
- Add a *finally*-block for code that should be executed in every case, no matter if an exception occured or not:
  - *try { code where an exception might occur }*
    catch (ExceptionName e) { code to handle the exception }
    finally { code that is executed in every case }

Task 1: Writing an Exception

- Create a new exception *NotRichEnoughException* which is thrown when a person is not rich enough to attend a party
- Create a class *ManagementParty* which extends the class *CompanyParty* – override the *participate()*-method so that it throws a *NotRichEnoughException* each time a person trying to participate has a salary of less then 500.000 – but remember that you still have to throw the *NotInvitedException* if the person belongs to another company (even if the person would be rich enough)

Task 2: Writing an Exception – part 2

- As exceptions are normal classes, we can apply the rules of inheritance to them – refactor the code of the *NotInvitedException* and the *NotRichEnoughException* so that one extends the other
  - Remember that the more specific class should always extend the more general class
  - Arrange the *catch*-statements in the main-method in the right order so that the method always catches the correct exception

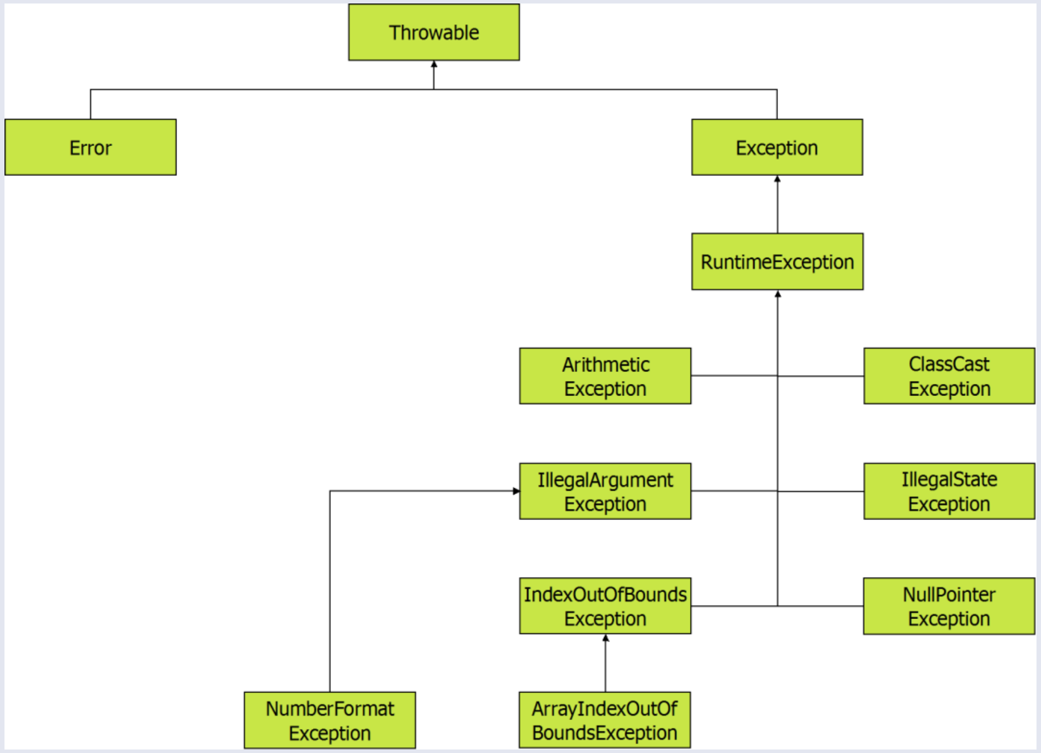
Task 3: Writing an Exception – part 3

- Create a new exception *NotInvitedPersonsException* which contains a list of the names of persons that are not invited as an attribute
- Create the method *public static void startParty(List<Person> guests, Location location)* in the class *Main* which takes a list of persons and a location as parameters, creates a new *ManagementParty* with the given location and lets every person in the list participate – if at least one of these persons is not able to participate (i. e. if an exception is thrown), the method *startParty* throws a *NotInvitedPersonsException* with the names of all persons unable to participate
- Create a list with at least five persons in the *main*-method and use the method *startParty()* with this list as parameter – catch the *NotInvitedPersonsException* and print the names of the persons that were not invited on the console if it is caught

Checked vs. Unchecked Exceptions

- *Checked Exceptions* (like the ones we created before) have to be thrown and caught in order to use them – otherwise the code cannot be compiled
- *Unchecked Exceptions* are not checked by the compiler and do not appear until the program is already running – therefore they are also called *RuntimeExceptions* (which is also the name of the superclass of all these exceptions)

Unchecked Exceptions



Defensive Programming

- *Unchecked Exceptions* are not caught because they point out mistakes/ weaknesses in our code which we can eliminate
- They can, but should not be caught in a *try-catch*-block – use *defensive programming* instead to prevent them from occuring

Unchecked Exceptions (Examples)

Exception	Occurs when …	Defensive Programming

ArithmeticException	… using invalid operations on numbers (for example division by 0)	*if (divisor != 0) {*
		...
		}

ArrayIndexOutOfBounds Exception	… trying to access an object at an array position that doesn‘t exist (index too high)	*if (index < array.length) {*
		…
		}

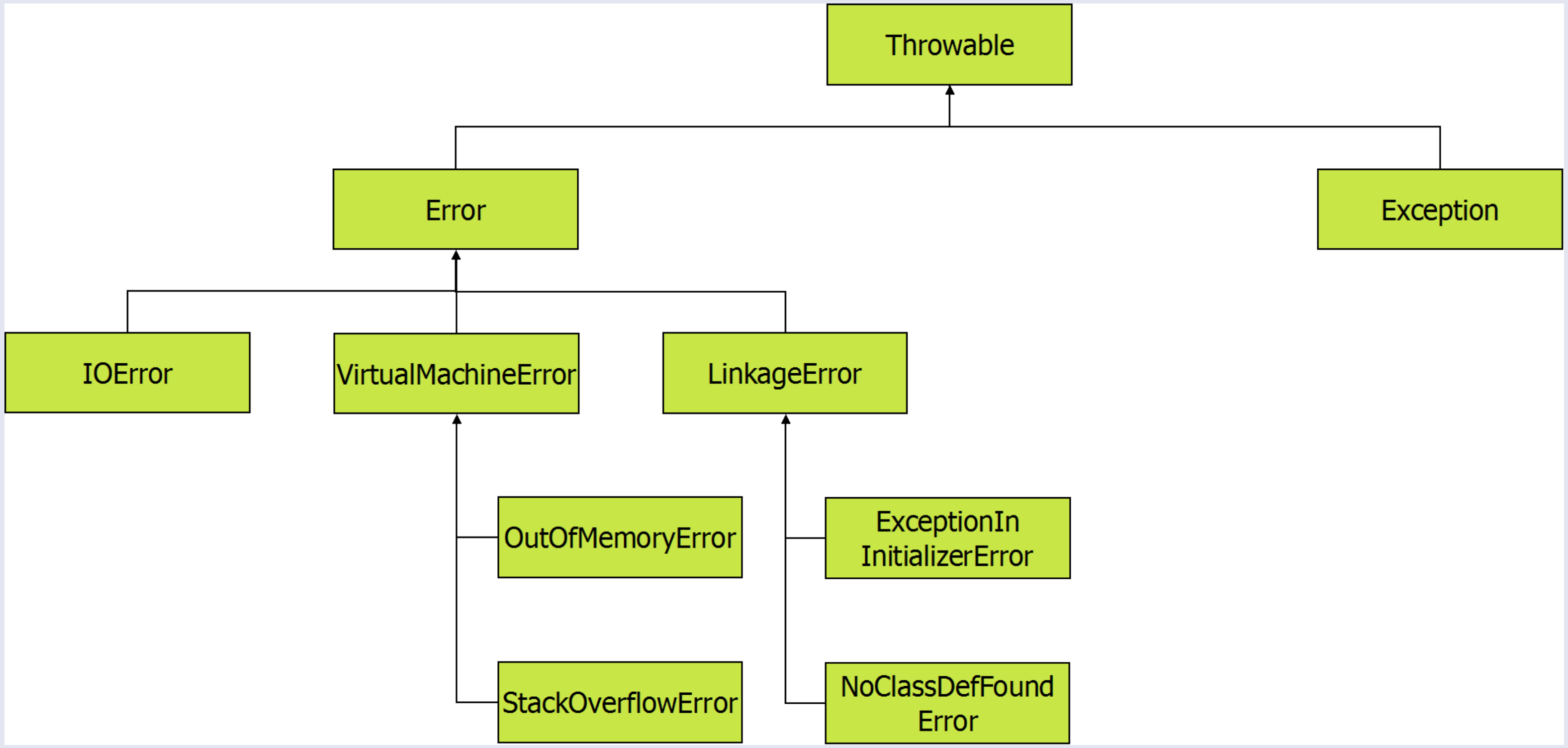
ClassCastException	… trying to cast an object into another object of incompatible type	*if (object instanceof Class) {*
		…
		}

NullPointerException	… trying to access an object that has the value *null*	*if (object != null) {*
		…
		}

Errors

- *Errors* are exceptional conditions that are external to the application, so they usually cannot be anticipated or recovered from – they can occur for example due to a hardware or system malfunction
- They can, but should not be caught in a *try-catch*-block – the problem causing the error cannot be solved within the program either way

Errors


Throwable
Error
Exception
IOError
VirtualMachineError
LinkageError
OutOfMemoryError
StackOverflowError
ExceptionIn InitializerError
NoClassDefFound Error

Excursus: User input

- A program can read input from the user via the class *java.util.Scanner*: at first a new *Scanner* has to be instantiated … :
  - *Scanner scanner = new Scanner(System.in);*
- … afterwards it can read *Strings* from the keyboard … :
  - *String s = scanner.next();*
- … these *Strings* can also be transformed into other data types – for example *int* by the method *parseInt(String s)* of the wrapper class *java.lang.Integer*:
  - *int i = Integer.parseInt(s);*
    - ➢ Warning: this can cause a *NumberFormatException*!