(8) Java Abstractions: Exceptions

Nico Ludwig (@ersatzteilchen)

TOC

- (8) Java Abstractions: Exceptions
  - Communicating Run Time Errors with *Exception*s
  - Control flow with *Exception*s
  - Creating own *Exception* Types
  - Checked and unchecked *Exception*s
- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Types of Errors when Writing Programs

- Syntax errors are errors that are found by the compiler, e.g. typos in symbol names or keywords.
- Logic errors are errors in the code that are not found by the compiler but result in wrong behavior of our code.
- Run time errors are a kind of logic errors that result in a potentially irrecoverable state of the program.
- An example of a run time error, we had already handled was dealing with invalid months of *Date* objects. Consider:

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) {        // The setter checks the
            if (1<= month && month <= 12) {      // validity of the value to be
                this.month = month;              // set for month.
            }
        }
}
```

- We use this example to start our discussion of exceptions to handle run time errors.

Reviewing clever Setters – Part 1

- Remember, when we have implemented a "clever setter" to avoid a programmer setting an invalid value for a month:

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) {        // The setter checks the
            if (1<= month && month <= 12) {      // validity of the value to be
                this.month = month;              // set for month.
            }
        }
}
```

  - And the setter *Date.setMonth()* is then clever enough to stop a programmer from setting the month "14":

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
myDate.setMonth(14); // Try to set quattrodecember.
myDate.print(); // myDate remains 17.10.2012!
// >17.10.2012
```

- Sure, it works and it is stable, but there is a problem!

- How should the caller of *Date.setMonth()* know, that what he tried to set was wrong and was even rejected to be set?
  - Mind, that the month to be set could be a calculated value, instead of a constant 14.
  - Concretely, the value for month could stem from user input and user input can be wrong at any time!
  - Bottom line: this "ignorant setter" could hide errors and bugs!

- So, how can we improve the situation?

Reviewing clever Setters – Part 2

- Before improving, we must formulate the problem: the caller of *Date.setMonth()* must know, that he did something wrong!
- The general question is: How should a method inform the caller, that there is a problem?
  - In our case the special question: How should *Date.setMonth()* inform the caller, that the passed month-value is invalid?
- Another way to signal an "error" can be done by adding a return value to *Date.setMonth()*:

```
// <Date.java>
public class Date { // (members hidden)
        public boolean setMonth(int month) {
            if (1<= month && month <= 12) {
                this.month = month;
                return true; // month was ok and was set.
            }
            return false; // month was invalid and was _not_ set.
        }
}
```

  - So, in case *Date.setMonth()*'s argument is a valid month it will be set and true will be returned, otherwise false will be returned:

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
boolean wasSuccessful = myDate.setMonth(14); // Try to set quattrodecember.
if (wasSuccessful) {
        myDate.print(); // We won't get here, because it was not successful!
}
```

- This is an improvement, but still has some serious issues...

Communicating Errors via return Values is problematic

- One problem is more "cosmetic", namely, a setter should usually not return a value (Command Query Separation (CQS)).
  - There are cases, in which it makes sense, but not in this example.
- The really serious problem is, that a caller of *Date.setMonth()* could forget to check, if the setter was not successful:

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
myDate.setMonth(14); // Try to set quattrodecember.
// Oups! We forget, that Date.setMonth() returns a value, we should check!
myDate.print();
// >17.10.2012
```

  - Classically, after an operation was done, the result should be checked.
    - E.g. the content of a register is checked (in an assembly language) or the returned value of a method is checked.
    - If the result indicates error, the code has to branch...
  - If an error condition was ignored (not checked), unexpected things may happen.
- Dilemma: Effective code and error handling code is mixed.
- Using return values to signal errors seems to be "not enough", not "enforcing enough".
- Java provides a very advanced feature to signal errors during run time, so called exceptions.

- CQS: Operations of an object should either only mutate the state of an object or query the state of an object.
- Before, we learn how to apply *Exception*s in our situation, let's remember, where we already had to deal with *Exception*s.

Exceptions – The Way Java deals with Run Time Errors

- During the last lectures, we've encountered several exceptions, which were raised by the JDK when an error occurred.
- Example 1: Dividing and int by zero raises an *ArithmeticException*:

```
int zero = 0;
int oddResult = 42/zero;
```

Terminal

```
NicosMBP:src nico$ java Program
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Program.main(Program.java:8)
NicosMBP:src nico$
```

- Example 2: Reading an int from a *Scanner*, if the *Scanner* scanned a text from the input raises an *InputMismatchException*:

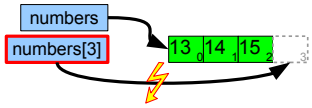
```
try (Scanner inputScanner = new Scanner(System.in)) {
        System.out.println("Please enter your age:");
        int yourAge = inputScanner.nextInt();
        if (yourAge < 0) {
                System.out.println("Your age mustn't be less than 0!");
        }
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter your age:
Nico
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Program.main(Program.java:11)
NicosMBP:src nico$
```

- Example 3: If we excess an array's bounds, the Java VM will raise an *ArrayIndexOutOfBoundsException*.

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```



A clever Setter throwing Exceptions – Part 1

- The idea is not to return a value in case of an error situation, but instead raise, or throw an *Exception* object:

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) {
            if (1<= month && month <= 12) {
                this.month = month;
            }
        }
}
```

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception {
            if (1<= month && month <= 12) {
                this.month = month;
            } else {
                throw new Exception();
            }
        }
}
```

  - As can be seen, we have changed quite a lot in the code of *Date.setMonth()*.

- Calling the new variant of *Date.setMonth()* with an invalid month, throws an *Exception* and terminates the program:

```
// <Program.java>
public class Program { // (members hidden)
        public static void main(String[] args) throws Exception {
            Date myDate = new Date(17, 10, 2012); // Construct the Date.
            myDate.setMonth(14); // Invalid! Throws an Exception.
            myDate.print(); // We won't get here!
        }
}
```

Terminal

```
NicosMBP:src nico$ java Program
Exception in thread "main" java.lang.Exception
    at Date.setMonth(Date.java:8)
    at Program.main(Program.java:10)
NicosMBP:src nico$
```

A clever Setter throwing Exceptions – Part 2

- We have changed a lot in *Date.setMonth()* to support *Exception*s:

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception { // Add throws Exception to the method head.
            if (1<= month && month <= 12) {
                this.month = month;
            } else {
                throw new Exception(); // Throw an Exception if the argument is invalid.
            }
        }
}
```

- In the method body we added an else clause to handle the invalid-argument-case, which throws an *Exception*.
  - As can be seen, we have to create a new *Exception*! This indicates, that an *Exception* is an object and *Exception* is a UDT/class!
  - The throw statement is used like a return statement, but it doesn't return something, i.e. we have no declared return type.
  - When a throw statement is hit (invalid value), the method stops execution and transports the *Exception*-object to the caller.
  - In Java lingo, we rather say an *Exception* is thrown instead of it is raised.
  - Since throw kind of works like return, it counts to Java's control flow statements.

- In the method head we needed to add a "throws *Exception*" specification.
  - It is required for the compiler to declare this, it says "I might throw an *Exception*".

- Methods, that declare the *Exception*s they might throw, are sometimes called "methods with checked *Exception*s".

Guard Clause as Alternative Style

- Alternatively, we can check a setter's arguments for invalidity and then throw an *Exception*.
  - When we want to code it this way, we must revert the condition on the arguments and throw immediately if the condition is met.

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception {
            if (1<= month && month <= 12) {
                this.month = month;
            } else {
                throw new Exception();
            }
        }
}
```



```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception {
            if (1 > month || month > 12) { // Guard clause
                throw new Exception();
            }
            this.month = month;
        }
}
```

- Because the if-branch here works as if it protects the "good" code from bad input, it is sometimes called guard clause.
  - Code using guard clauses is sometimes called "fail fast code", because it'll "fail" before any "good" code is executed.
  - "Failing fast" not necessarily means "fail with an *Exception*", it can also mean returning a value signaling error or a default value.

- It doesn't matter which style, i.e. "positive check" or "guard clause" is used, but it can be agreed upon as coding guideline.
  - Guard clauses can reduce the depth of cascading/nesting because the happy path is at the same level as the guard clauses.

Exceptions in Constructors – Part 1

- If a run time error is happening in a ctor, throwing *Exception*s is the only way to communicate the problem to the caller.
- We can rewrite *Date*'s ctor to throw *Exception* in case of erroneous arguments:
  - We can reuse and just call *Date.setMonth()*, it'll "forward" *Exception* to the caller of the ctor, if the passed month-value is invalid.

```
// <Date.java>
public class Date { // (members hidden)
        private int day;
        private int month;
        private int year;

        public Date(int d, int m, int y) throws Exception { // Add throws Exception to the ctor head.
            day = d;
            setMonth(m); // Just call the "exceptional" Date.setMonth() method.
            year = y;
        }
        public void setMonth(int month) throws Exception {
            if (1<= month && month <= 12) {
                this.month = month;
            } else {
                throw new Exception(); // Throw an Exception, if the argument is invalid.
            }
        }
}
```

- If an *Exception* is thrown from a ctor (or "escapes" a ctor), the object will not be created at all!

```
Date myDate = new Date(22, 45, 2011); // Will throw an exception, won't create a new Date object.
myDate.print(); // This statement will not be reached at all.
```

Exceptions in Constructors – Part 2

- If instance fields are initialized in a class definition with exceptional methods, all ctors must forward these *Exceptions*.
  - Here, *Date.readInt()* is an exceptional method which is called to initialize the fields in the class definition:

```
// <Date.java>
public class Date { // (members hidden)
        private int day = readInt("Please enter a day:");
        private int month = readInt("Please enter a month:");
        private int year = readInt("Please enter a year:");

        private static int readInt(String message) throws Exception {
            try (Scanner inputScanner = new Scanner(System.in)) {
                    System.out.println(message);
                    return inputScanner.nextInt();
            }
        }

        public Date() throws Exception {
            // pass
        }
}
```

Exceptions in Constructors – Part 3

- static fields cannot be initialized in a class definition with exceptional methods!

- Instead they must be initialized and potential *Exception*s be handled in the classes static block:

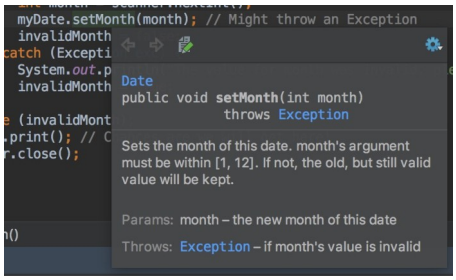
```
// <Game.java>
public class Game {
        private static int playTimeInms;

        static {
            try {
                playTimeInms = readInt("Enter your desired playtime:");
            } catch (Exception e) {
                System.out.println("Oops!");
            }
        }

        private static int readInt(String message) throws Exception {
            try (Scanner inputScanner = new Scanner(System.in)) {
                System.out.println(message);
                return inputScanner.nextInt();
            }
        }
}
```

- If a *RuntimeException* is thrown while initializing a static field or from a static block, it'll be wrapped by an *ExceptionInInitializerError*.

Javadoc's @throws Tag

- Java allows to document, under which circumstances, esp. checked *Exceptions* are thrown:

```
// <Date.java>
public class Date { // (members hidden)
        /**
         * Sets the month of this date. month's argument must be within
         * [1, 12]. If not, the old, but still valid value will be kept.
         *
         * @param month the new month of this date
         * @throws Exception if month's value is invalid
         */
        public void setMonth(int month) throws Exception {
                if (1<= month && month <= 12) {
                        this.month = month;
                } else {
                        throw new Exception();
                }
        }
}
```

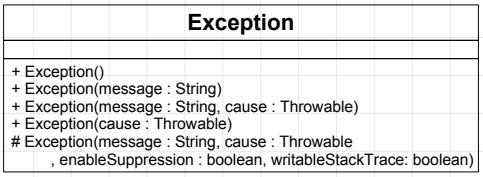


- We can use the Javadoc tag @throws to document the *Exception*s being potentially thrown from a method.

The Ctors of Class Exception – Part 1

- We have mentioned, that *Exception* is a class, so lets have a look into *Exception*'s code:

```
// Somewhere in the JDK
public class Exception extends Throwable {
        public Exception() {
        }

        public Exception(String message) {
            super(message);
        }

        public Exception(String message, Throwable cause) {
            super(message, cause);
        }

        public Exception(Throwable cause) {
            super(cause);
        }

        protected Exception(String message, Throwable cause
            , boolean enableSuppression, boolean writableStackTrace) {
            super(message, cause, enableSuppression, writableStackTrace);
        }
}
```



- At least there are some interesting ctors we will apply now: throwing an *Exception* with a message is very handy.

- We'll discuss *Exception's* super class *Throwable*, esp. the methods *Exception*s derive from it at a later point.
  - => *Exception* inherits a lot of methods from *Throwable*, but *Exception* itself only offers ctors.

The Ctors of Class Exception – Part 2

- Alright, let's use *Exception*'s ctor accepting a message as *String* to create a message object with better information:

```
// <Date.java>
public class Date { // (members hidden)
	public void setMonth(int month) throws Exception {
		if (1<= month && month <= 12) {
			this.month = month;
		} else {
			throw new Exception("invalid argument for new month: " + month);
		}
	}
}
```

- The caller's code doesn't change:

```
// <Program.java>
public class Program { // (members hidden)
	public static void main(String[] args) throws Exception {
		Date myDate = new Date(17, 10, 2012); // Construct the Date.
		myDate.setMonth(14); // Invalid! Throws an Exception.
		myDate.print(); // We won't get here!
	}
}
```

- The improvement is visible at run time: the *Exception*'s message is printed on the console and it is self-explaining:
  - The message is not written to STDOUT but STDERR.

```
Terminal
NicosMBP:src nico$ java Program
Exception in thread "main" java.lang.Exception: invalid argument for new month: 14
    at Date.setMonth(Date.java:8)
    at Program.main(Program.java:10)
NicosMBP:src nico$
```

Handling Exceptions – Motivation

- Usually we don't want an *Exception* to terminate the program, but to give the user a chance for valid input.
  - In opposite to returned values a thrown *Exception* does terminate a program, if it is not handled! This is also called non-local exit.
  - Unhandled *Exception*s can also "just" end a thread.

- To do something reasonable with a thrown *Exception* in our program, we have to handle the *Exception*.
  - An unhandled *Exception* is still reasonable at run time, because it can give us a message, so that we can find the root cause.
  - That unhandled *Exception*s terminate a program, is also reasonable! – An invalid month may lead to more awful behavior of a program!

- In Java we handle *Exception*s with the keywords try and catch. Let's handle *Date.setMonth()*'s *Exception* more gently:

```
// <Program.java>
public class Program { // (members hidden)
        public static void main(String[] args) throws Exception {
            Date myDate = new Date(17, 10, 2012); // Construct the Date.
            try {
                myDate.setMonth(14); // Invalid! Throws an Exception.
            } catch (Exception exc) {
                System.out.println("Program was terminated, because of an invalid month");
            }
            myDate.print(); // This time, we will get here!
        }
}
```

Terminal

```
NicosMBP:src nico$ java Program
Program was terminated, because of an invalid month
NicosMBP:src nico$
```

  - Ok, it's childish, because the program still terminates and our "handling" is just a "non-exceptional" text on the console.

Handling Exceptions – New Means for Control Flow

- In our case it is better to give the user a chance to enter valid months, in case an invalid month was set:

```
public static void main(String[] args) throws Exception {
        Date myDate = new Date(17, 10, 2012); // Construct the Date.
        Scanner scanner = new Scanner(System.in);
        boolean invalidMonth = false;
        do {
            try {
                System.out.println("Please enter a value for month:");
                int month = scanner.nextInt();
                myDate.setMonth(month); // Might throw an Exception
                invalidMonth = false;
            } catch (Exception exc) {
                System.out.println("The value for month was invalid, please enter a new value.");
                invalidMonth = true;
            }
        } while (invalidMonth);
        myDate.print(); // Chances are we will get here!
        scanner.close();
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
14
The value for month was invalid, please enter a new value
Please enter a value for month:
10
17.10.2012
NicosMBP:src nico$
```

- It should be fairly clear right now, that try/catch is woven into the code and that it is basically a control flow statement.
  - If an *Exception* is thrown in *Date.setMonth()*, *invalidMonth* = *false* will not be reached at all.

- Now its time time to discuss try/catch in more depth.

Handling Exceptions – Control Flow Variations – Part 1

```
// <Program.java>
public class Program { // (members hidden)
	public static void main(String[] args) throws Exception {
		Date myDate = new Date(17, 10, 2012); // Construct the Date.
		try {
			myDate.setMonth(14); // Invalid! Throws an Exception.
		} catch (Exception exc) {
			System.out.println("Program was terminated, because of an invalid month.");
		}
		myDate.print(); // We won't get here!
	}
}
```

- The try-block encloses code, which might throw an *Exception*. The call to *Date.setMonth()* might throw *Exception*.
- The catch-clause "catches" the thrown *Exception*, if any.
  - Mind the semantic analogy of an *Exception* object to a ball, that is thrown and caught, hence the keywords.
  - The catch-block of the belonging to catch-clause is only executed, if an *Exception* was thrown from the belonging to try-block.
  - The catch-block works similar to a method, it accepts the thrown *Exception* object in its "argument" list (= the catch-clause).
- The try/catch syntax does clearly separate "normal code" from error handling code.
  - This is very different from using returning values, where it can be hard to tell error handling from "normal" code.

Handling Exceptions – Control Flow Variations – Part 2

- Since we are about to improve structured error handling, we should address other invalid input as well.

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
                System.out.println("Please enter a value for month:");
                int month = scanner.nextInt();
                myDate.setMonth(month); // Might throw an Exception
                invalidMonth = false;
        } catch (Exception exc) {
                System.out.println("The value for month was invalid, please enter a new value.");
                invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```

- With the code above, we are allowed to input a text instead of a number, which leads *Scanner.nextInt()* to fail:

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
June
The value for month was invalid, please enter a new value
Please enter a value for month:
The value for month was invalid, please enter a new value
Please enter a value for month:
The value for month was invalid, please enter a new value
...
```

- What is happening here?

Handling Exceptions – Control Flow Variations – Part 3

- (1) In case *Scanner.nextInt()* reads a the text "June", an *Exception* is thrown.

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
June
```

- (2) When this *Exception* is thrown, the control flow will enter the catch-block.
  - (2.1) A message is written to the console.
  - (2.2) Then *invalidMonth* is set to true.
- (3) do-while's condition evaluates to true and the loop starts over.

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
                System.out.println("Please enter a value for month:");
                int month = scanner.nextInt(); // Might also throw an Exception
                myDate.setMonth(month); // Might throw an Exception
                invalidMonth = false;
        } catch (Exception exc) {
                System.out.println("The value for month was invalid, please enter a new value.");
                invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```

- (4) The control flow enters the try-block again.
  - (4.1) The prompt "Please enter a value for month:" is written to the console.
  - (4.2) Then the control flow doesn't wait for input this time, but *Scanner.nextInt()* immediately throws an *Exception*.
- Why does the control flow just throws an *Exception* at *Scanner.nextInt()* and doesn't wait for input?
  - Well, *Scanner* now contains the text "June" in its internal buffer, which needs to cleared to consume the next int.
  - The program is in an infinity loop now!

Handling Exceptions – Control Flow Variations – Part 4

- The program is in an infinity loop because we handled *Scanner.nextInt()*'s *Exception* incorrectly!
- A simple solution is just to check a precondition, before *Scanner.nextInt()* is executed:
  - The idea is to use the method *Scanner.hasNextInt()* to check and ignore input (via *Scanner.next()*) until an int is in the buffer.
  - When having an int in the buffer, we can safely call *Scanner.nextInt()* to read the month.

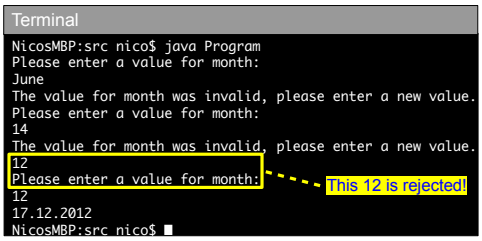
```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
            System.out.println("Please enter a value for month:");
            while (!scanner.hasNextInt()) {
                System.out.println("The value for month was invalid, please enter a new value:");
                scanner.next();
            }
            int month = scanner.nextInt();
            myDate.setMonth(month); // Might throw an Exception
            invalidMonth = false;
        } catch (Exception exc) {
            System.out.println("The value for month was invalid, please enter a new value.");
            invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```

- But of course we want to solve the problem using *Exception* handling instead of preconditions.
- In a sense checking preconditions is the opposite strategy to handling *Exceptions*!

Handling Exceptions – Control Flow Variations – Part 5

- So, the other way to cope with invalid input (*Scanner.nextInt()* reads text instead of int) is to handle the thrown *Exception*.
  - As we've already mentioned, we have to clear *Scanner*'s internal buffer from the obstacle text, we do this with *Scanner.next()*:

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might also throw an Exception
            myDate.setMonth(month); // Might throw an Exception
            invalidMonth = false;
        } catch (Exception exc) {
            System.out.println("The value for month was invalid, please enter a new value.");
            scanner.next();
            invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```



- Calling *Scanner.next()* in the catch-block handles our problem with the text "June", but introduces another problem.
- In case we enter an invalid int value, and try to enter a valid int value afterwards, our input is again rejected!
  - At the third try, our input is accepted.

Handling Exceptions – Control Flow Variations – Part 6

- (1) On the prompt, we enter "June", which is an invalid int for *Scanner.nextInt()*.
  - (1.1) *Scanner.nextInt()* throws an *Exception*.
  - (1.2) The *Exception* is handled in the catch-block by clearing *Scanner*'s internal buffer with *Scanner.next()*.
  - (1.3) *invalidMonth* is set to true
- (2) The do while loop starts over.
  - (2.1) We enter 14 at the prompt.
  - (2.2) *Date.setMonth()* throws an *Exception*, because 14 is an invalid month!
  - (2.3) The *Exception* is also handled in the catch block by clearing *Scanner*'s internal buffer with *Scanner.next()*.
  - (2.4) But this time, *Scanner.next()* waits for input and blocks the program! This is because *Scanner*'s buffer is empty, it now awaits input. Now, we have to enter a value, 12, but it is not "processed" on our *Scanner.next()* call.
  - (2.5) The do while loop starts over and reaches *Scanner.nextInt()* again.
- The problem: the *Exceptions* thrown by *Scanner.nextInt()* or *Date.setMonth()* must be handled differently!

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might also throw an Exception
            myDate.setMonth(month); // Might throw an Exception
            invalidMonth = false;
        } catch (Exception exc) {
            System.out.println("The value for month was invalid, please enter a new value.");
            scanner.next();
            invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```

can block control flow!

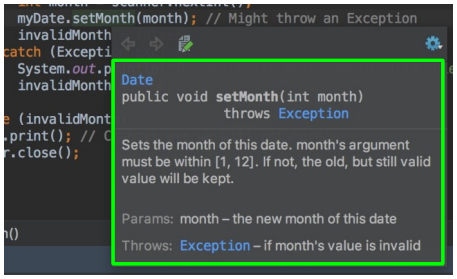
Terminal

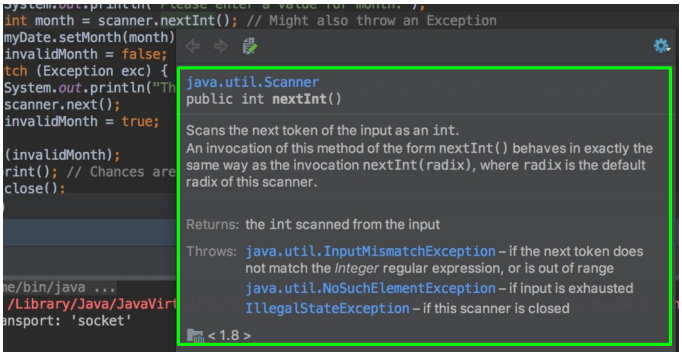
```
NicosMBP:src nico$ java Program
Please enter a value for month:
June
The value for month was invalid, please enter a new value
Please enter a value for month:
14
The value for month was invalid, please enter a new value
12
Please enter a value for month:
12
17.12.2012
NicosMBP:src nico$
```

Handling Exceptions – Different Exceptions – Part 1

- Fortunately, Java allows handling different types of *Exception*s in different ways!

- Of course, the precondition to make this work is, that *Scanner.nextInt()* and *Date.setMonth()* throw different *Exceptions*!
  - Actually, *Date.setMonth()* and *Scanner.nextInt()* do literally throw different types of *Exception*s.





  - *Date.setMonth()* throws an *Exception* in case of a wrong month argument, we programmed it ourselves.
  - But, *Scanner.nextInt()* throws a *java.util.InputMismatchException*, in case the input token is no int!

Handling Exceptions – Different Exceptions – Part 2

- *java.util.InputMismatchException*, is an ordinary class of the JDK of course:

```
// Somewhere in the JDK

public class InputMismatchException extends NoSuchElementException {
        public InputMismatchException() {
            super();
        }
        public InputMismatchException(String s) {
            super(s);
        }
}
```

- As can be seen, *InputMismatchException* inherits from *NoSuchElementException*:

```
// Somewhere in the JDK

public class NoSuchElementException extends RuntimeException {
        // pass
}
```

- And *NoSuchElementException* inherits from *RuntimeException*:

```
// Somewhere in the JDK

public class RuntimeException extends Exception {
        // pass
}
```

- … and *RuntimeException* inherits from *Exception*.

- => So, Java's *Exception* types are built in an inheritance hierarchy!

Handling Exceptions – Different Exceptions – Part 3

- To drive this point home, we can add multiple catch-clauses, incl. multiple catch-blocks to a try-block.
  - We can handle *Exception* and *InputMismatchException* just separately to solve our problem:

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
June
The input value was probably no int, please enter an int value.
Please enter a value for month:
14
The value for month was invalid, please enter a new value.
Please enter a value for month:
3
17.3.2012
NicosMBP:src nico$
```

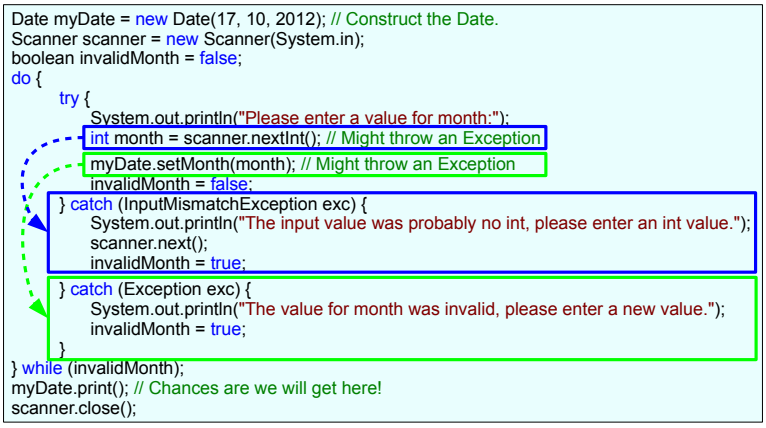
Here we see, how we can handle different errors with *Exceptions*: an *Exception* signals, that a run time error emerged and can describes, what happened (*Exception* type and possibly a message).

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
                System.out.println("Please enter a value for month:");
                int month = scanner.nextInt(); // Might also throw an Exception
                myDate.setMonth(month); // Might throw an Exception
                invalidMonth = false;
        } catch (InputMismatchException exc) {
                System.out.println("The input value was probably no int, please enter an int value.");
                scanner.next();
                invalidMonth = true;
        } catch (Exception exc) {
                System.out.println("The value for month was invalid, please enter a new value.");
                invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```

- In this code
  - *Scanner.nextInt()*'s *InputMismatchException* is handled correctly using *Scanner.next()*.
  - Whereas *Date.setMonth()*'s *Exception* is handled different from that (no call to *Scanner.next()*).
  - We have even used different console messages to communicate the problem more precisely to the user.

Handling Exceptions – Different Exceptions – Part 4

- When we overlay the possible flow of control, in case *Exception* or *InputMismatchException* …



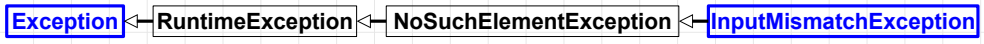
  - … we can see how the specific catch-handlers handle the matching *Exception* or *InputMismatchException*.

- Because the catch-clauses control, which *Exception* types they handle, they are also called *Exception*-/catch-filters

Handling Exceptions – Different Exceptions – Part 5

- In an early version of the code, the catch-filter for *Exception* handled *InputMismatchException* thrown from *Scanner.nextInt()* and *Exception* thrown from *Date.setMonth()*!
  - How can the same catch-filter "catch" different types of *Exceptions*?

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
Scanner scanner = new Scanner(System.in);
boolean invalidMonth = false;
do {
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an InputMismatchException
            myDate.setMonth(month); // Might throw an Exception
            invalidMonth = false;
        } catch (Exception exc) {
            System.out.println("The value for month was invalid, please enter a new value.");
            scanner.next();
            invalidMonth = true;
        }
} while (invalidMonth);
myDate.print(); // Chances are we will get here!
scanner.close();
```



- *InputMismatchException* inherits *Exception*, a catch-filter accepts a thrown *InputMismatchException* as substitute for *Exception*.
  - Thus the catch-filter for *Exception* matches for *InputMismatchException* as well!
  - The generalization-specialization relation between *Exception* derivates enables the LSP here!

Handling Exceptions – Different Exceptions – Part 6

- Actually, catch-filters work like methods, which accept objects of more special types!
- If we want to handle *Exception*s, we have to put the code that might throw *Exception*s into a try-block:

```
try {
        int month = scanner.nextInt(); // Might throw an InputMismatchException
        myDate.setMonth(month); // Might throw an Exception
} catch (InputMismatchException exc) {
        System.out.println("Handles InputMismatchException");
} catch (Exception exc) {
        System.out.println("Handles Exception");
}
```

  - The appended catch-blocks have the chance to handle thrown *Exception*s.
- The written order of the catch-blocks is important!
  - We are not allowed to simply put *catch (Exception exc)* before *catch (InputMismatchException exc)*!

```
try {
        int month = scanner.nextInt(); // Might throw an InputMismatchException
        myDate.setMonth(month); // Might throw an Exception
} catch (Exception exc) {
        System.out.println("Handles Exception");
} catch (InputMismatchException exc) {
        System.out.println("Handles InputMismatchException");
}
```

  - This wouldn't make sense: all more special *Exception*s, incl. *InputMismatchException* would be caught by *catch (Exception exc)*!
  - … and the filter *catch (InputMismatchException exc)* would never be reached!

Multi catch Clause – Part 1

- Often, we've to handle more than one *Exception* equivalently and others more specifically.
  - *Scanner.nextInt()* can throw *InputMismatchException*, *NoSuchElementException* and *IllegalStateException*.
  - I.e. a specific method can throw one of multiple *Exceptions*.
  - Let's handle *InputMismatchException* and *IllegalStateException* in a common way, but other *Exceptions* different from that:

```
try {
        int month = scanner.nextInt(); // Might throw InputMismatchException, NoSuchElementException or IllegalStateException
        myDate.setMonth(month); // Might throw an Exception
} catch (InputMismatchException exc) {
        System.out.println("Handles exc");
} catch (IllegalStateException exc) {
        System.out.println("Handles exc");
} catch (Exception exc) {
        System.out.println("Handles Exception");
}
```

- We can avoid the code repetition in the catch-blocks by using a multi-catch-clause:

```
try {
        int month = scanner.nextInt(); // Might throw InputMismatchException, NoSuchElementException or IllegalStateException
        myDate.setMonth(month); // Might throw an Exception
} catch (NoSuchElementException | IllegalStateException exc) {
        System.out.println("Handles exc");
} catch (Exception exc) {
        System.out.println("Handles Exception");
}
```

  - Multi-catch: if common handling is required for multiple *Exception* types, that have a common super *Exception*, but handling the common super *Exception* would catch too many cases.
  - So the *Exception* types declared in multi-catch-clause must have a common super *Exception*.

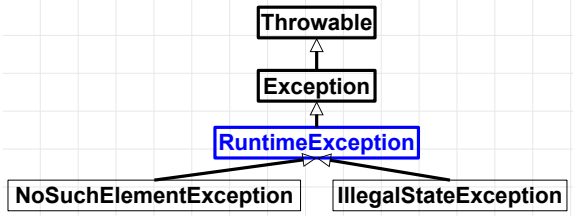
Multi catch Clause – Part 2

- When multi-catch is used, we can only call methods of the nearest common super *Exception* class.
  - However, the *Exception* types in multi-catch, must not be derived from each other!
  - The written order of the *Exception* types in multi-catch has no special meaning to Java.

- E.g. in case we *catch* (*NoSuchElementException* | *IllegalStateException)* we can only call methods of *RuntimeException*:

```
try {
        int month = scanner.nextInt();
} catch (NoSuchElementException | IllegalStateException exc) {
        System.out.println(exc.getMessage());
} catch (Exception exc) {
        System.out.println("Handles Exception");
}
```

  - The hierarchy shows, that *RuntimeException* is the nearest common super class of *NoSuchElementException* and *IllegalStateException*:



- Of course we can call any method, which *RuntimeException* inherits from in its super classes.

The thrown Exception Object

- Now we should discuss the *Exception* object a catch-handler receives, when an *Exception* is thrown:

```
try {
        int month = scanner.nextInt();
        myDate.setMonth(month); // Might throw an Exception
} catch (Exception exc) {
        System.out.println("Handles Exception");
}
```

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception {
                if (1<= month && month <= 12) {
                        this.month = month;
                } else {
                        throw new Exception("invalid argument for new month: " + month);
                }
        }
}
```

  - *exc* is a reference to exactly the same *Exception* object, that was thrown in *Date.setMonth()*.
  - Consequently, *exc* is just referring to an *Exception* object, so we can call methods on *exc* and also pass it to methods!

- As we said Java organizes *Exception*s in a class hierarchy. All *Exception* classes have one very super class: *Throwable*.

```
// Somewhere in the JDK:
public class Throwable implements Serializable {
        public Throwable() { /* pass */ }
        public Throwable(String message) { /* pass */ }
        public String getMessage() { /* pass */ }
        public Throwable getCause() { /* pass */ }
        public void printStackTrace() { /* pass */ }
        public StackTraceElement[] getStackTrace() { /* pass */ }
}
```

Throwable
+ Throwable() + Throwable(message : String) + getMessage() : String + getCause() : Throwable + printStackTrace() + getStackTrace() : StackTraceElement[]

Good to know:
Most types, which end with *-able* are interfaces, *Throwable* is a notable exception of this rule.

  - On the next slides we will discuss some of the methods, which are provided by *Throwable*.

Obtaining Information from an Exception Object: the Message

- *Exception* inherits *Throwable.getMessage()*:

```
try {
        int month = scanner.nextInt();
        myDate.setMonth(month); // Might throw an Exception
} catch (Exception exc) {
        System.out.println("The exception message: " + exc.getMessage());
}
```

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws Exception {
                if (1<= month && month <= 12) {
                        this.month = month;
                } else {
                        throw new Exception("invalid argument for new month: " + month);
                }
        }
}
```

  - *Exception.getMessage()* will just return the textual message, we passed to *Exception*'s ctor, when we threw the *Exception*:

```
Terminal
NicosMBP:src nico$ java Program
Please enter a value for month:
14
The exception message: invalid argument for new month: 14
Please enter a value for month:
```

- *Throwable.getMessage()* is also inherited by *InputMismatchException*, therefor we can call it in the respective handler:

```
try {
        int month = scanner.nextInt(); // Might throw an InputMismatchException
        myDate.setMonth(month); // Might throw an Exception
} catch (InputMismatchException exc) {
        System.out.println("InputMismatchException: " + exc.getMessage());
} catch (Exception exc) {
        System.out.println("Handles Exception: " + exc.getMessage());
}
```

The Method Call Stack revisited – Part 1

- When an *Exception* is thrown, the situation is comparable to control flow using return values.
- There is an important difference: we don't know when an *Exception* is thrown!
  - When a value is returned, we can clearly identify the call stack. This is more difficult, when an *Exception* is thrown.
- Assume this code, in which we put a simplified variant of reading a month of *Date* into an own method:

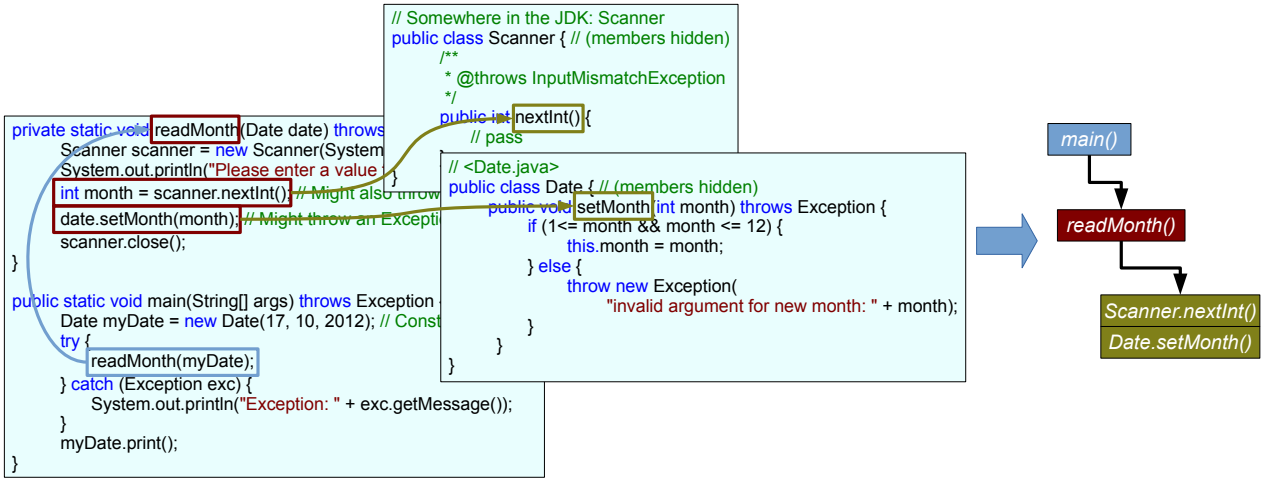
```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might also throw an Exception
        date.setMonth(month); // Might throw an Exception
        scanner.close();
}
```

  - This method doesn't do any own *Exception* handling, instead it declares "throws *Exception*".

```
public static void main(String[] args) throws Exception {
        Date myDate = new Date(17, 10, 2012); // Construct the Date.
        try {
            readMonth(myDate);
        } catch (Exception exc) {
            System.out.println("Exception: " + exc.getMessage());
        }
        myDate.print();
}
```

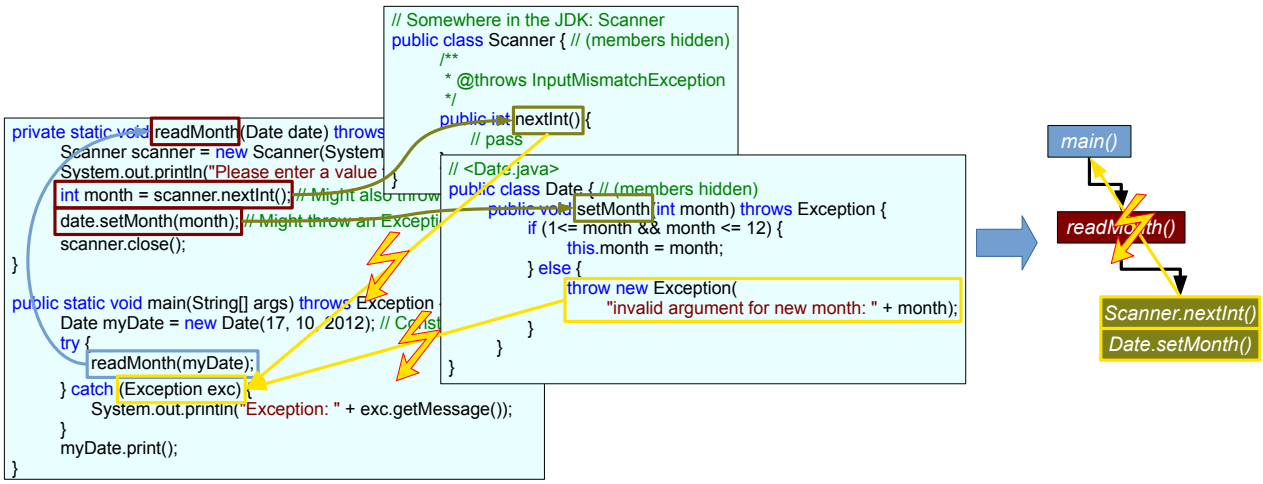
  - So, when an *Exception* is thrown, which "path" did it take to hit the catch-block?

The Method Call Stack revisited – Part 2



- Calling *readMonth()* in *main()* calls *Scanner.nextInt()* and then *Date.setMonth()*, this builds up a call stack.
  - When those methods end their execution and return, they give back control to the calling method.

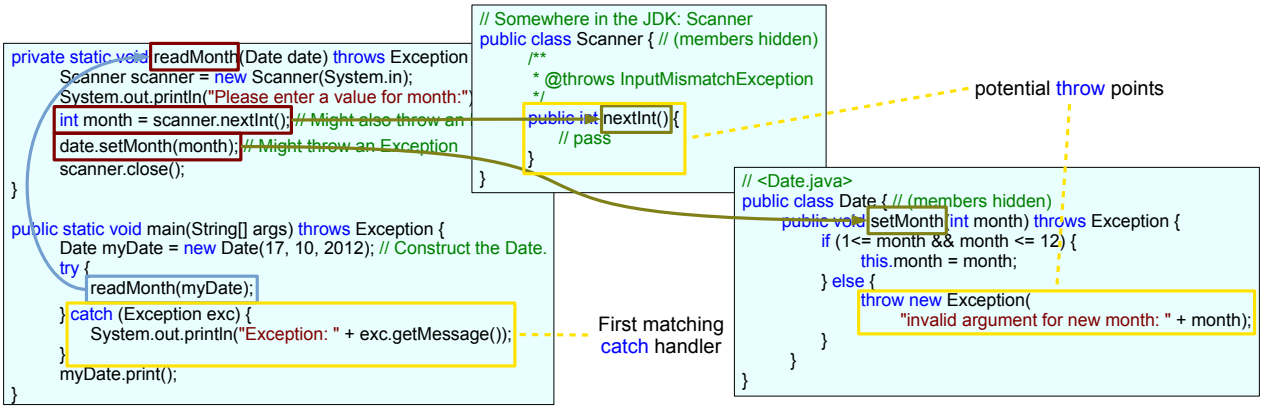
The Method Call Stack revisited – Part 3



- If an *Exception* is thrown from *Scanner.nextInt()* or *Date.setMonth()* the control flow is different from a "returning method".
  - The control flow somehow "crosses" *readMonth()* and flow from *Scanner.nextInt()* or *Date.setMonth()* directly to *main()*!

- When an *Exception* is thrown, the JVM stops normal program execution and unwinds the call stack to make this happen.

Stack Unwinding

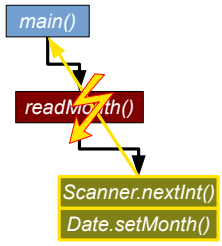


- More exactly, when an *Exception* is thrown, the call stack is unwound until the first matching catch-filter is found.
  - The call stack is unwound from the point the *Exception* was thrown, this is called throw point.
  - When an *Exception* is thrown, and is not handled, the enclosing method won't return regularly.
  - The unwinding ends in the first try-block, that has a matching catch-clause and enters that catch-clause.
  - When a matching catch-block is entered the *Exception* counts as handled and unwinding ends.
  - If *Exception*s lead to unwinding until *main()*, we can usually not do a lot more than logging the exception state and exit the program.

Call Stack and Stack Trace

- Initially, we asked the question, how to know the path of the thrown *Exception*. – We want to know the unwound call stack!
  - In addition to the message of an *Exception* we can also get the (unwound) call stack from a caught *Exception* object.

- The method *Exception.printStackTrace()* prints the stack trace to STDERR and *Exception.getStackTrace()* retrieves an array representing the steps of the call stack for more control.




```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
try {
        readMonth(myDate);
} catch (Exception exc) {
        System.out.println("The call stack:");
        exc.printStackTrace();
}
myDate.print();
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
14
The call stack:
java.lang.Exception: invalid argument for new month: 14
      at Date.setMonth(Date.java:33)
      at Program.readMonth(Program.java:55)
      at Program.main(Program.java:42)
17.10.2012
NicosMBP:src nico$
```

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
try {
        readMonth(myDate);
} catch (Exception exc) {
        System.out.println("The call stack:");
        StackTraceElement[] callStack = exc.getStackTrace();
        for (int i = 0; i < callStack.length; ++i) {
            System.out.printf("%d. %s%n", i + 1, callStack[i]);
        }
}
myDate.print();
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
14
The call stack:
1. Date.setMonth(Date.java:33)
2. Program.readMonth(Program.java:58)
3. Program.main(Program.java:42)
17.10.2012
NicosMBP:src nico$
```

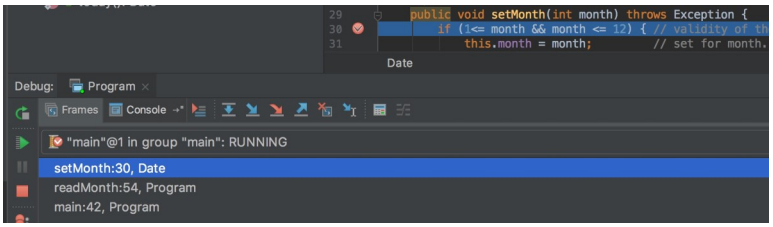
Getting the Call Stack without Exceptions

- From within a called method, we can also get the call stack independently of any thrown *Exception*.
  - The method call *Thread.currentThread().getStackTrace()* provides a *StackTraceElement[]*, that can be further processed.

```
public void setMonth(int month) throws Exception {
        StackTraceElement[] callStack = Thread.currentThread().getStackTrace();
        for (int i = 0; i < callStack.length; ++i) {
            System.out.printf("%d. %s%n", i + 1, callStack[i]);
        }
        if (1<= month && month <= 12) {
            this.month = month;
        } else {
            throw new Exception("invalid argument for new month: " + month);
        }
}
```

```
Terminal
NicosMBP:src nico$ java Program
Please enter a value for month:
12
1. java.lang.Thread.getStackTrace(Thread.java:1551)
2. Date.setMonth(Date.java:33)
3. Program.readMonth(Program.java:54)
4. Program.main(Program.java:42)
17.10.2012
NicosMBP:src nico$
```

- The debuggers of pretty all IDEs are able to show the call stack, e.g. on a hit breakpoint:

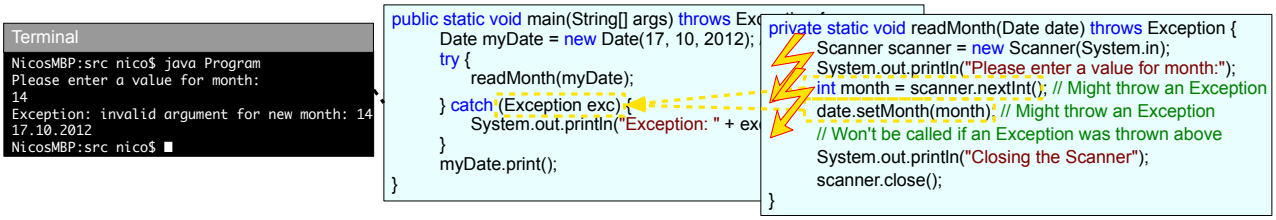


  - Mind, that the IDE interprets the call stack as a series of stack frames ("Frames").

- The question "From which method an *Exception* was thrown?" is still similar to "From which method the value was returned". *Exception*s just cross method-borders using a different control flow.

Tidying Up – Part 1

- In many cases it is required to do "tidy up" activities, after a try-block was executed, esp. if an *Exception* was thrown.
- Let's inspect the code of the method *readMonth()*:

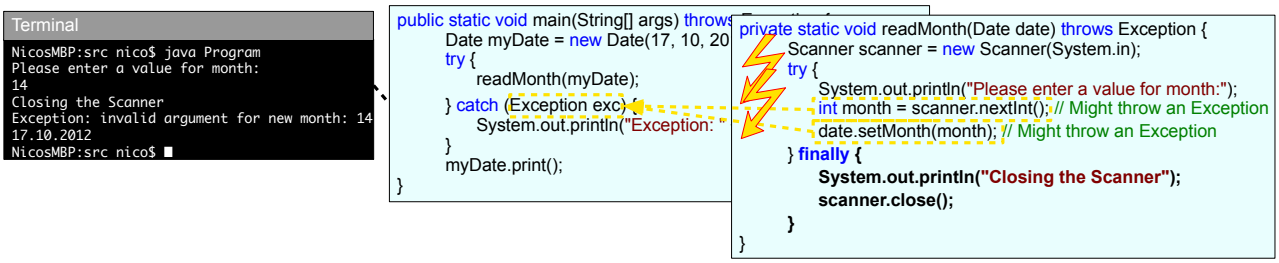


  - As can be seen, we not even use a try-block in *readMonth()*, because we rely on the caller (*main()*) to handle potential *Exceptions*!
  - The problem: if *Scanner.nextInt()* or *Date.setMonth()* throw an *Exception*, *readMonth()* exits immediately, before *scanner* is closed!
    - We can prove that, because "Closing the Scanner" is not written to the console, because the method was exited before.
- If *readMonth()* is called, and *Exceptions* are thrown in *Scanner.nextInt()* or *Date.setMonth()* we might have a resource leak.
  - Worse, in case *readMonth()* is called for multiple times in a program ending in *Exceptions*, this can cause a really big resource leak!
  - Although, Java has a GC, some resources must be told explicitly to do their tidy up, before the GC reclaims memory!
- Bottom line: *Exceptions* change the flow of control in a way, that resource/memory leaks might occur!

Tidying Up – Part 2

- Of course Java has a means to tidy up after *Exception*s have been thrown, by adding more control statements.
  - But we have to change *readMonth()* in a way, so that it does at least a little *Exception* handling itself.

- Java provides the finally-block to handle tidy up situations of associated try-blocks. Let's modify *readMonth()*:



- finally solves our problem: *scanner* is correctly closed, even if an *Exception* was thrown in the belonging to try-block.
  - We can generally prove that, because "Closing the Scanner" is now written to the console!

Tidying Up – Part 3

- finally-blocks are also executed, if no *Exception* is thrown at all!
  - Actually, this a very desired behavior! Mind that *scanner* should be closed always in *readMonth()*!

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an Exception
            date.setMonth(month); // Might throw an Exception
        } finally {
            System.out.println("Closing the Scanner");
            scanner.close();
        }
}
```

```
Terminal
NicosMBP:src nico$ java Program
Please enter a value for month:
3
Closing the Scanner
17.3.2012
NicosMBP:src nico$
```

  - As can be seen, *scanner* is correctly closed if no *Exception* was thrown.
    - Once again, we can generally prove that, because "Closing the Scanner" is written to the console!

- Please mind: finally-blocks are executed always, even if no *Exception* was thrown in the belonging to try-block!
  - This can be a source of bugs, so correct usage of finally requires some practice.

Tidying Up – Part 4

- We can append finally-blocks to one or more catch-handlers.
- E.g. the following code catches *InputMismatchException* from *scanner.nextInt()*, but lets all other *Exception*s pass:

```
public static void main(String[] args) throws Exception {
        Date myDate = new Date(17, 10, 2012); // Construct the Date.
        try {
            readMonth(myDate);
        } catch (Exception exc) {
            System.out.println("Exception: " + exc.getMessage());
        }
        myDate.print();
}
```

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an Exception
            date.setMonth(month); // Might throw an Exception
        } catch (InputMismatchException imex) {
            System.out.println("Maybe the entered value is no int!");
        } finally {
            System.out.println("Closing the Scanner");
            scanner.close();
        }
}
```

- *InputMismatchException* is not only handled, it is "almost" ignored and just a message is written to the console:

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
October
Maybe the entered value is no int!
Closing the Scanner
17.10.2012
NicosMBP:src nico$
```

  - Mind, the output text "Closing the Scanner" proves, that the finally-block is still executed, despite *InputMismatchException*.

Re-throwing Exceptions – Part 1

- Instead of "swallowing" an *Exception* in an *Exception* handler, we can re-throw it.
  - E.g. the following code catches *InputMismatchException* from *Scanner.nextInt()*, but lets all other *Exception*s pass:

```
public static void main(String[] args) throws Exception {
        Date myDate = new Date(17, 10, 2012); // Construct the Date.
        try {
            readMonth(myDate);
        } catch (Exception ex) {
            System.out.println("Exception handled in main: " + ex.getClass());
        }
        myDate.print();
}
```

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an Exception
            date.setMonth(month); // Might throw an Exception
        } catch (InputMismatchException imex) {
            System.out.println("Maybe the entered value is no int!");
            throw imex; // Re-throw this Exception
        } finally {
            System.out.println("Closing the Scanner");
            scanner.close();
        }
}
```

- But this time a message is written to the console and *InputMismatchException* is just re-thrown:

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
August
Maybe the entered value is no int!
Closing the Scanner
Exception handled in main: class java.util.InputMismatchException
17.10.2012
NicosMBP:src nico$
```

  - Mind, that the *InputMismatchException* reaches *main()*, but nevertheless, the *Scanner* is closed.
  - => finally blocks are also executed if another *Exception* was thrown from a belonging to catch handler!

Re-throwing Exceptions – Part 2

- When we re-throw *Exceptions*, this will (of course) enlarge their stack trace.
  - We can print the stack trace of a caught *Exception* to STDERR with the method *Throwable.printStackTrace()*.

```
public static void main(String[] args) throws Exception {
        Date myDate = new Date(17, 10, 2012);
        try {
            readMonth(myDate);
        } catch (Exception ex) {
            ex.printStackTrace(); // Prints the full stack trace to STDERR.
        }
}
```

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            date.setMonth(scanner.nextInt());
        } catch (InputMismatchException imex) {
            throw imex; // Re-throw this Exception
        } finally {
            scanner.close();
        }
}
```

- The stack trace of the *InputMismatchException* is really long

```
Terminal
NicosMBP:src nico$ java Program
Please enter a value for month:
What?
java.util.InputMismatchException
    at java.util.Scanner.throwFor()
    at java.util.Scanner.next()
    at java.util.Scanner.nextInt()
    at java.util.Scanner.nextInt()
    at Program.readMonth()
    at Program.main()
NicosMBP:src nico$
```

  - In such a case, it would be better the let the stack trace restart in *readMonth()* to make it shorter and hide its origin.

Re-throwing Exceptions – Part 3

- Restarting/rewriting or "filling in" an *Exception*'s stack trace is simple: just call the method *Throwable.fillInStackTrace()*:

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            date.setMonth(scanner.nextInt());
        } catch (InputMismatchException imex) {
            imex.fillInStackTrace();
            throw imex; // Re-throw this Exception
        } finally {
            scanner.close();
        }
}
```

- *Throwable.fillInStackTrace()* deletes the stack frame recorded up to this point and starts recording another stack frame.
  - After calling *fillInStackTrace()* in *readMonth()*, the old stack frame (from *Scanner*) is deleted and started over in *readMonth()*.
  - So, we get this stack frame in *Program.main()*:

Terminal

```
NicosMBP:src nico$ java Program
Please enter a value for month:
What?
java.util.InputMismatchException
    at Program.readMonth()
    at Program.main()
NicosMBP:src nico$
```

  - (The method *Throwable.setStackTrace()* allows to explicitly set a *StackTraceElement[]*.)

try-with-Resource-Blocks – Part 1

- As we just discussed, there exist objects, which must be tidied up explicitly, before the GC can "reclaim them".
  - Using a finally-block to execute such tidy up activities is a common pattern in Java:

```
private static void readMonth(Date date) throws Exception {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an Exception
            date.setMonth(month); // Might throw an Exception
        } finally {
            scanner.close();
        }
}
```

- Java allows us to write this in a simpler way by using a try-with-resource-block:

```
private static void readMonth(Date date) throws Exception {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw an Exception
            date.setMonth(month); // Might throw an Exception
        } // Scanner.close() is automatically called here
}
```

  - Let's discuss how the try-with-resource-block works …
  - The feature behind the try-with-resource-block is often called Automatic Resource Management (ARM).

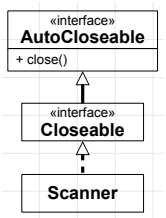
try-with-Resource-Blocks – Part 2

- try-with-resource-blocks works with all classes, that implement the interface *AutoCloseable*.
  - *AutoCloseable* provides only one method, *close()*.
  - *Scanner* implements the interface *Closeable*, and *Closeable* extends *AutoCloseable*:

```
// Somewhere in the JDK:
public interface AutoCloseable {
        void close() throws Exception;
}
```

```
// Somewhere in the JDK:
public interface Closeable extends AutoCloseable {
        void close() throws Exception;
}
```

```
// Somewhere in the JDK:
public final class Scanner implements Closeable {
        /**
         * Closes this scanner.
         */
        public void close() { /*pass*/ }
}
```



- And then the try-with-resource-block in action awaits an *AutoCloseable* object in the resource-specification:

resource-specification

```
try (Scanner scanner = new Scanner(System.in) ) {
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might throw an Exception
        date.setMonth(month); // Might throw an Exception
} // AutoCloseable.close() is automatically called here
```

try-with-Resource-Blocks – Part 3

- We can define extended try-with-resource-blocks, they can have additional catch-blocks and a finally-block:
  - *close()* is called before a matching catch-handler is entered and before the finally-block is entered.

```
try (Scanner scanner = new Scanner(System.in)) {
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might throw an Exception
        date.setMonth(month); // Might throw an Exception
} catch (InputMismatchException imexc) {
        System.out.println("an InputMismatchException");
} finally {
        System.out.println("in finally");
}
```

- An individual try-with-resource-block can manage multiple resource-specifications:

```
try (   Scanner scanner = new Scanner(System.in);
        Scanner scanner2 = new Scanner(System.in)) {
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might throw an Exception
        date.setMonth(month); // Might throw an Exception
}
```

  - Mind, that we must specify a list of resource-specification statements (separated with semicolon) for the try-with-resource-block.
  - *close()* is called in the opposite order of the resource-specification statements: *scanner2.close()* is called before *scanner.close()*.

- try-with-resource-blocks can also handle *Exception*s from the resource-specification's ctors.
  - In this case *close()* is not called, because the resource object was not created!

- What if multiple *close()*-calls throw *Exception*s when multiple resources are specified in the resource specification?
  - If *scanner2.close()* throws an *Exception*, this *Exception* will be thrown from the try-with-resources-block. Nevertheless the JVM will also try to call *scanner.close()* (mind: opposite order of declaration in the resource specification), but if *scanner.close()* also throws an *Exception* this *Exception* will be suppressed.
  - We will discuss *Exception* suppression on the next slides.

Exception Suppression – Part 1

- try-with-resources calls *AutoCloseable.close()* on the resource in question, if an *Exception* is thrown in the try-block.
  - But … what happens, if *AutoCloseable.close()* itself throws an *Exception*?

- If try-with-resources doesn't have catch-blocks, *Exceptions* from *AutoCloseable.close()* will be forwarded to the caller:

```
// <MyResource.java>
public class MyResource implements AutoCloseable {
        @Override
        public void close() throws Exception {
            throw new Exception("close()");
        }
}
```

```
// <Program.java>
public class Program {
        public static void main(String[] args) throws Exception {
            try (MyResource myResource = new MyResource()) {
                // pass
            }
        }
}
```

Terminal

```
NicosMBP:src nico$ java Program
Exception in thread "main" java.lang.Exception: close()
    at MyResource.close()
    at Program.main()
NicosMBP:src nico$
```

Exception Suppression – Part 2

- What happens, if an *Exception* is thrown in the try-block and *AutoCloseable.close()* also throws an (another) *Exception*?

```
// <MyResource.java>
public class MyResource implements AutoCloseable {
        public void doIt() {
            throw new Exception("doIt()");
        }
        @Override
        public void close() throws Exception {
            throw new Exception("close()");
        }
}
```

```
// <Program.java>
public class Program {
        public static void main(String[] args) throws Exception {
            try (MyResource myResource = new MyResource()) {
                myResource.doIt();
            } catch (Exception e) {
                System.out.println("Caught: "+e);
            }
        }
}
```

- In this case, only the *Exception*, which escaped the try-block (thrown by *myResource.doIt()*) will be caught:

```
Terminal
NicosMBP:src nico$ java Program
Caught: java.lang.Exception: doIt()
NicosMBP:src nico$
```

  - The *Exception* thrown by *MyResource.close()* is said to be suppressed.

Exception Suppression – Part 3

- Don't worry! We can also inspect *Exception*s, which have been suppressed.
  - Therefor the super class *Throwable* provides the method *getSuppressed()*:

```
// <Program.java>
public class Program {
	public static void main(String[] args) throws Exception {
		try (MyResource myResource = new MyResource()) {
			myResource.doIt();
		} catch (Exception e) {
			System.out.println("Caught: "+e);
			if (null != e.getSuppressed()) {
				for (Throwable suppressedException : e.getSuppressed()) {
					System.out.println("Suppressed exception: "+suppressedException);
				}
			}
		}
	}
}
```

- *Throwable.getSuppressed()* returns a *Throwable[]* of exceptions suppressed on "their way to delivery":

Terminal

```
NicosMBP:src nico$ java Program
Caught: java.lang.Exception: doIt()
Suppressed exception: java.lang.Exception: close()
NicosMBP:src nico$
```

Creating Custom Exceptions – Part 1

- Now, let's think the other way around. This code handles all *Exception*s as if they were thrown from *Date.setMonth()*:

```
try (Scanner scanner = new Scanner(System.in)) {
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might also throw an Exception
        myDate.setMonth(month); // Might throw an Exception
} catch (InputMismatchException exc) {
        System.out.println("The input value was probably no int.");
        scanner.next();
} catch (Exception exc) {
        System.out.println("The value for month was invalid, please enter a new value.");
}
```

- If an *Exception* is thrown, which is not an *InputMismatchException* we handle it always as run time error from *Date.setMonth()*.
- The problem: if any other piece of code in the try-block throws *Exception*, we might handle it wrong!
  - I.e. we'll handle it as if it was an error from *Date.setMonth()*, which might not be the case!
- Java's way to solve this is just using OO! We'll just create a new own *Exception* class and let it represent the "*Date* error"!
  - *Exception* classes we as developers create on our own are called custom *Exception*s.

Creating Custom Exceptions – Part 2

- So, we will create a new class, that derives from *Exception*, we call it *InvalidDateException*:

```
// <InvalidDateException.java>
public class InvalidDateException extends Exception {
        public InvalidDateException() {
                super();
        }
        public InvalidDateException(String message) {
                super(message);
        }
}
```

```
// <Date.java>
public class Date { // (members hidden)
        /**
         * @throws InvalidDateException if month's value is invalid
         */
        public void setMonth(int month) throws InvalidDateException {
                if (1<= month && month <= 12) {
                        this.month = month;
                } else {
                        throw new InvalidDateException("invalid argument for new month: " + month);
                }
        }
}
```

  - Conventionally, the name of a custom *Exception* class ends with "Exception".
  - A dctor is provided that delegates to *Exception*'s dctor and *Exception*'s ctor accepting a message is also passed through.

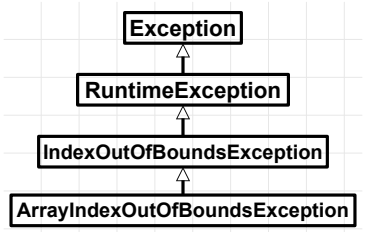
- A (user defined) class derived from *Exception* or *Throwable* takes part in Java's *Exception* idiom.

- The gained benefit is, that we can handle *InvalidDateException* specifically as run time error from *Date.setMonth()*:

```
try (Scanner scanner = new Scanner(System.in)) {
        System.out.println("Please enter a value for month:");
        int month = scanner.nextInt(); // Might throw an InputMismatchException
        myDate.setMonth(month); // Might throw an InvalidDateException
} catch (InputMismatchException exc) {
        System.out.println("The input value was probably no int.");
        scanner.next();
} catch (InvalidDateException exc) { // Handle run time errors from Date.setMonth()
        System.out.println("The value for month was invalid, please enter a new value.");
}
```

Naming Exception Types

- The names of *Exception* classes are a good example of names being jammed together to show more special types.



- Example: *ArrayIndexOutOfBoundsException* extends *IndexOutOfBoundsException*, which in turn extends *Exception*.
  - *Exception* is, well, a very common *Exception* type.
  - *IndexOutOfBoundsException* is a more special *Exception* type, it is thrown, if an index accessing an *ArrayList* is out of bounds.
  - *ArrayIndexOutOfBoundsException* is an even more specialized *Exception*, thrown, if an index accessing an array is out of bounds.
  - => As can be seen, on each specialization level the *Exception* type's name is prepended with more special words.

- Jamming words together to build type names is not always a good practice, but it is typically ok for *Exceptions*.

Throws Clause – Part 1

- Up to now, we have missed the fact, that we have to add a throws clause to a method that throws an *Exception*.

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) throws InvalidDateException {
            if (1<= month && month <= 12) {
                this.month = month;
            } else {
                throw new InvalidDateException("invalid argument for new month: " + month);
            }
        }
}
```

- In a callee method, we have two options to deal with *InvalidDateException* potentially thrown from *Date.setMonth()*:
  - (1) We can handle *InvalidDateException* in the called method:

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw an InvalidDateException
        } catch (InvalidDateException exc) {
            System.out.println("Handles InvalidDateException");
        }
}
```

  - (2) Or we can forward *InvalidDateException*, in this case it is thrown to the caller:

```
public static void readMonth(Date date) throws InvalidDateException {
        try (Scanner scanner = new Scanner(System.in)) {
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw an InvalidDateException
        }
}
```

Throws Clause – Part 2

- Handling *InvalidDateException* in the called method, just means, that the called method handles the *Exception* itself.
  - This means, that yet other methods calling the called method, won't have to deal with the handled *Exception*:

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw an InvalidDateException
        } catch (InvalidDateException exc) {
            System.out.println("Handles InvalidDateException");
        }
}
```

- When we forward *InvalidDateException*, it just means, that the called method does not handle the *Exception* itself.
  - When a method decides not to handle an *Exception* it must generally declare, that the *Exception* "can escape" from the method:

```
public static void readMonth(Date date) throws InvalidDateException {
        try (Scanner scanner = new Scanner(System.in)) {
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw an InvalidDateException
        }
}
```

throws clause

  - We have to declare *Exception*s, that "can escape" from a method in the throws clause of a method.
    - That's all! – No *Exception* handler in form of a try-block in *readMonth()* comes into play!

- In this course we call code, that can be a source of an *Exception*, that is handled or forwarded "*Exception* aware code".

Throws Clause – Part 3

- throws clauses can be set on all kinds of methods, incl. ctors in classes.
  - And throws clauses can be set on all methods in interfaces and enums.

- Methods only differing in throws clauses, do not overload! Thus, this will not compile:

```
public interface X {
        void b() throws Exception;
        void b() throws InvalidDateException; // Invalid! 'b()' is already defined in 'X'
}
```

- A method can set a list of potentially thrown *Exceptions* in the throws clause:

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw InputMismatchException
            date.setMonth(month); // Might throw InvalidDateException
        } catch (InputMismatchException exc) {
            System.out.println("handles InputMismatchException");
            scanner.next();
        } catch (InvalidDateException exc) {
            System.out.println("handles InvalidDateException");
        }
}
```

list in throws clause

```
public static void readMonth(Date date)
        throws InputMismatchException, InvalidDateException {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt(); // Might throw InputMismatchException
            date.setMonth(month); // Might throw InvalidDateException
        }
}
```

  - Instead of handling *InputMismatchException* and *InvalidDateException*, we can put them into the list of the throws clause.
  - The written order of the *Exception* types in the throws clause has no special meaning to Java.

- Lists in throws clauses show another difference to return values: a method can throw one of many *Exception* types.

Throws Clause – Part 4

- If a super class or interface declares a method with a throws clause, sub classes/implementors can decide to take it over.

```
public interface X {
        void b() throws Exception;
}
```

  - We can have no throws clause at all or just take over the one offered in the super type:

```
// OK: Implement b() without throws declaration.
public class CX1 implements X {
        @Override
        public void b() {
            // pass
        }
}
```

```
public class CX2 implements X {
        // OK: Implement b() and take over X.b()'s throws declaration.
        @Override
        public void b() throws Exception {
            // pass
        }
}
```

  - We can also use a more special throws clause than the one offered in the super class:

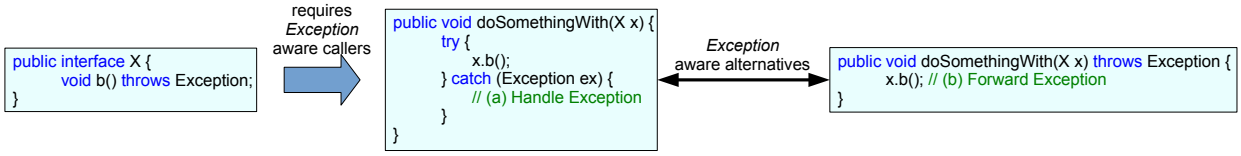
```
public class CX3 implements X {
        // OK: Implement b() and use a throws declaration, more special than on X.b().
        @Override
        public void b() throws InvalidDateException {
            // pass
        }
}
```

  - But we cannot have a more general throws clause than the one offered in the super class:

```
public class CX4 implements X {
        // Invalid: CX4 cannot implement b() in X overridden method does not throw java.lang.Throwable
        @Override
        public void b() throws Throwable {
            // pass
        }
}
```

Throws Clauses and Covariance – Motivation

- Because *X.b()* has a throws clause, any caller of *X.b()* must be *Exception* aware (handle or forward the *Exception*).
  - Esp. because *X.b()* throws *Exception*, *doSomethingWith(X)* must be *Exception* aware:



  - *doSomethingWith()* must be *Exception* aware: it doesn't know, which or if none *Exception* is thrown from an implementation of *X*!

- (1) An implementor of *X.b()* can declare to throw *Exception*, no *Exception* or a more special *Exception*.
- (2) But an implementor is not allowed to declare a more general *Exception*!
- (1) and (2) ensure, that *doSomethingWith(X)* works correctly with *Exceptions*!

```
public class CX1 implements X {
    @Override
    public void b() {
    }
}
```

```
public class CX2 implements X {
    @Override
    public void b() throws Exception {
    }
}
```

```
public class CX3 implements X {
    @Override
    public void b() throws InvalidDateException {
    }
}
```

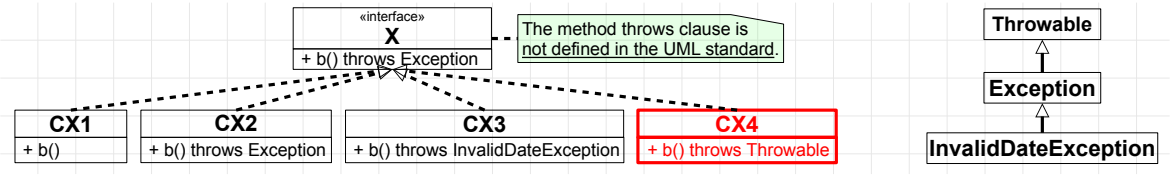
```
public class CX4 implements X {
    @Override
    public void b() throws Throwable {
    }
}
```

Throws Clauses and Covariance – Formal

```
public interface X {
        void b() throws Exception;
}
```

- Implementors of *X.b()* can declare to throw *Exception*, no *Exception*, a more special *Exception*, but not a more general *Exception*!

- Consider this architecture:



  - More special implementations/sub classes (*CX1* is more special than *X*) can only throw more special *Exceptions*.
  - But *Throwable*, that is thrown in *CX4.b()* is more general than *Exception* thrown from *X.b()* and hurts this rule.

- In oo terms we say, that a thrown *Exception* in a sub class is covariant to the *Exception* thrown in the super class.
  - Only with covariant method throws clauses, *Exception* aware code on super classes/interfaces will work properly!
  - I.e. we need covariant throws clauses to have Java's *Exception* idiom supporting the LSP!
  - Remember, that also return types are covariant in Java!

Run Time Selection of correct Exception Handler – Part 1

- Assume the interface *X* with *X.b()* throwing *Exception* and its implementation *CX3*, having *CX3.b()* throw *InvalidDateException*:

```
public interface X {
        void b() throws Exception;
}
```

```
public class CX3 implements X {
        @Override
        public void b() throws InvalidDateException {
            // pass
        }
}
```

  - This meets the consistency rules, because throwing the more special *InvalidDateException* from *CX3.b()* is fine!

- In the following code, *doSomethingWith(X)*:

```
doSomething(new CX3());
```

```
public void doSomethingWith(X x) {
        try {
            x.b();
        } catch (InvalidDateException ex) {
            // Handle Exception
        } catch (Exception ex) {
            // Handle InvalidDateException
        }
}
```

  - (1) just knows, that *X.b()* might throw an *Exception* of type *Exception*, but
  - (2) it doesn't know, that *CX3.b()* might throw an *InvalidDateException*.
  - However, we could add a catch-clause handling *InvalidDateException*, although *X* doesn't tell us about *InvalidDateException*!

- Java allows us to have some variance to match method throws and catch-handlers.

Run Time Selection of correct Exception Handler – Part 2

- When a method overload is resolved for execution, the compile time information of the argument list is evaluated.
  - Assume the class *B* and its super class *A*:

```
public class A {
}
```

```
public class B extends A {
}
```

  - Assume the overloaded method *doIt()*, accepting *A* or *B*:

```
public static void doIt(A a) {
        System.out.println("in doIt(A)");
}
public static void doIt(B b) {
        System.out.println("in doIt(B)");
}
```

  - When we call *doIt()* with an *A* or *B* argument, the correct overload is selected at compile time!

```
A a = new B();
doIt(a);
// >in doIt(A)
doIt(new B());
// >in doIt(B)
```

catch clauses resemble method parameter lists, but resolution works differently.
- Method arguments: compile time
- Exceptions: run time.

- When a catch-clause is selected for execution, the run time type of the thrown *Exception* is evaluated.
  - Assume *InvalidDateException* and its super class *Exception*:

```
public class Exception {
}
```

```
public class InvalidDateException extends Exception {
}
```

  - Assume the interface *X* and its implementation *CX3*.
  - *X.b()* throws *Exception* and *CX3.b()* throws *InvalidDateException*.

```
public interface X {
        void b() throws Exception;
}
```

```
public class CX3 implements X {
        @Override
        public void b() throws InvalidDateException {
            throw new InvalidDateException();
        }
}
```

  - When we call *doSomething()* with an *X* or *CX3* argument and *b()* throws, the catch-handler is selected at run time!

```
X x = new CX3();
doSomething(x);
// >in InvalidDateException
doSomething(new CX3());
// >in InvalidDateException
```

```
public void doSomethingWith(X x) {
        try {
            x.b();
        } catch (InvalidDateException ex) {
            System.out.println("in InvalidDateException");
        } catch (Exception ex) {
            System.out.println("in Exception");
        }
}
```

- Because the selection of the matching catch-handler happens at run time costly run time type checks must be performed. So *Exception* handling is relatively expensive.

Checked Exceptions

- Handling *Exception*s using throws clauses and catch-handlers is called checked *Exception* handling.
- Checked *Exception*s are a good idea in general: we have to care for *Exception*s, either declare or handle locally.
- But there are situations, esp. with idioms that came with Java 8, where checked *Exception*s are cumbersome.
  - A prominent example are lambdas, whose code is significantly blown up with catch-handlers for checked *Exception*s.
- On the other hand, we also encountered *Exception*s we didn't declare or handle, they "just occurred".
  - Just mind *NullPointerException*. Did we ever handle or add a throws clause for *NullPointerException*?
  - Another example is the method *Scanner.nextInt()*. It does not have any throws clause, but it does indeed throw *Exception*s:

```
// Somewhere in the JDK:
public class Scanner {
        /**
         * @throws InputMismatchException if the next toke is no int
         * @throws NoSuchElementException if input is exhausted
         * @throws IllegalStateException if this scanner is closed
         */
        public int nextInt() {
            // pass
        }
}
```

There is no throws clause!
We only know that *Scanner.nextInt()* might throw, by noticing the @throws tags in the comments!

- Obviously, Java does not only provide checked *Exception*s! We can also use unchecked *Exceptions*!

Checked and unchecked Exceptions

- *Exception* types can inherit from the class *Exception* or from the class *RuntimeException*.

- If an *Exception* type inherits from *RuntimeException*, it becomes an unchecked *Exception* type.
  - E.g. we can easily change *InvalidDateException* to an unchecked *Exception* type, by inheriting from *RuntimeException*:

```
// <InvalidDateException.java>
public class InvalidDateException extends Exception {
        public InvalidDateException() {
            super();
        }
        public InvalidDateException(String message) {
            super(message);
        }
}
```

```
// <InvalidDateException.java>
public class InvalidDateException extends RuntimeException {
        public InvalidDateException() {
            super();
        }
        public InvalidDateException(String message) {
            super(message);
        }
}
```

  - After this change, we can also change *readMonth()*, so that no explicit handling or forwarding of *InvalidDateException* is needed:

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw InvalidDateException
        } catch (InvalidDateException exc) {
            System.out.println("handles InvalidDateException");
        }
}
```

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw InvalidDateException
        }
}
```

Unchecked Exceptions are Exceptions!

- Although an *Exception* is "just" an unchecked *Exception*, it is still an *Exception*, that must be handled somewhere!

```
public static void readMonth(Date date) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw InvalidDateException
        }
}
```

  - If *main()* calls *readMonth()*, we're not forced to handle or forward (throws clause) any *Exception*, because only unchecked *Exceptions* are thrown:

```
public static void main(String[] args) throws Exception {
        readMonth(myDate);
        myDate.print();
}
```

  - If, *readMonth()* throws an *Exception*, it will "escape" *main()*, terminate the program and write the stack trace to STDERR.

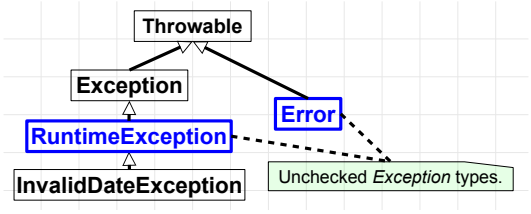
- From a practical standpoint, unchecked *Exceptions* exist for programmer's convenience.
  - (+) They allow certain Java core *Exceptions* to occur basically always without the need to handle them explicitly.
    - *NPEs* could happen every time we access members using '.'. Java would be very uncomfortable, if we had to handle *NPEs* all the time!
  - (+) For own *Exception* types, deriving them from *RuntimeException* reduces code, e.g. for lambdas and using method references.
  - (-) They are unchecked, they aren't needed to be handled: the compiler doesn't tell us, that there might be something to handle.

- In the Java community, the general trend seems to be mostly defining new *Exception* types as unchecked *Exceptions*.

- Obviously, .NET, which is only using unchecked *Exceptions*, was designed that way, because checked *Exceptions* were noticed to be very uncomfortable for programmers. – Esp. for beginners and programmers coming from other languages.
- To support lambdas and the *Stream*-API (both will be discussed in a future lecture) to work with exceptional code, types like *java.io.UncheckedIOException* (>= Java 8) and *java.util.concurrent.Callable<V>* (>= Java 5) can be used.

Unchecked Exceptions – The Error Exception

- Java also provides another kind of unchecked *Exception*: *Error*.
  - *Error* is not derived from *Exception*, but from *Throwable*, which is also the super class of *Exception*.



- It should be said, that *Error*-type *Exception*s are not meant to be extended by developers, they are "reserved" for the JVM.
  - When an *Error* is thrown in a program, this usually means, the program cannot recover from this situation.
  - This means *Error*s should also not be handled! – Therefor *Throwable* should never be handled as well, it might catch too much!

```
// <MyError.java>
public class MyError extends Error { // !! Don't extend Error !!
        // pass
}
```

```
public static void MyError() {
        try {
                // pass
        } catch (Error error) { // !! Don't catch Error !!
                // pass
        }
}
```

  - A typical class of *Error*s, are infrastructure problems, e.g. *OutOfMemoryError*, from which there is no way to recover from easily.
  - Esp. catching (and ignoring) *Error*s can easily hide serious problems, which are very tricky to find!

- Also *ExceptionInInitializerError*, which is potentially thrown from static field initialization or static blocks is an *Error*-type *Exception*.

Which Exception should or should not be handled?

- Generally, it is not a good idea to handle the general unchecked *Exceptions* *RuntimeException* and *Error*.
  - Esp. *NPE*s should never be handled! Instead the condition that a reference is null should be handled proactively!
  - It's ok to handle own *RuntimeException*s! Just handling of the JDK's *RuntimeException*s is not a good idea!
  - Also *Error*'s sub classes should not be handled, because we cannot recover from the fatal state they signal in most cases.

```
public static void readMonth(Date date) { // Swallows all unchecked Exceptions:
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw InvalidDateException
        } catch (RuntimeException | Error exc){
            System.out.println("handles everything");
        }
}
```

- Handling *Throwable* is suspicious, because it would hide any other *Exception*!

```
public static void readMonth(Date date) {  // Swallows all checked and unchecked Exceptions:
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Please enter a value for month:");
            int month = scanner.nextInt();
            date.setMonth(month); // Might throw InvalidDateException
        } catch (Throwable exc){
            System.out.println("handles everything");
        }
}
```

- Also handling the very general type *Exception* is not good! It could hide serious issues!

Partially handling RuntimeExceptions and re-throwing

- We just clarified, that some *Exception* types should not be handled. But we can still catch those *Exception*s and re-throw them!
- Assume, we have a mathematical algorithm, which might throw *ArithmeticException*s (e.g. division by zero).
  - To approach the problem, we can catch *ArithmeticException* and log a message:

```
public static void calculation() {
        for (int i = 0; i < maxIndex; ++i) {
            try {
                complexCalculation(coordinates[i]);
            } catch (ArithmeticException exc) {
                System.out.println("problem found for data " + coordinates[i] + ": " + exc);
            }
        }
}
```

- However, there is one issue: in case an *ArithmeticException*s is thrown the algorithm just continues!
  - Mind, that this is not what we want, it could hide further problems. – Handling *RuntimeException*s is fishy!
  - Instead of handling *ArithmeticException*, we can catch and re-throw it:

```
public static void calculation() {
        for (int i = 0; i < maxIndex; ++i) {
            try {
                complexCalculation(coordinates[i]);
            } catch (ArithmeticException exc) {
                System.out.println("problem found for data " + coordinates[i] + ": " + exc);
                throw exc;
            }
        }
}
```

Wrapping and re-throwing Exceptions – Part 1

- We can also refine partial exception handling by the combination with own exceptions, let's define *CalculationException*:

```
// <CalculationException.java>
public class CalculationException extends RuntimeException {
        public CalculationException(Throwable cause) {
            super(cause);
        }

        public CalculationException(String message, Throwable cause) {
            super(message, cause);
        }
}
```

RuntimeException

+ RuntimeException()
+ RuntimeException(message : String)
+ RuntimeException(message : String, cause : Throwable)
+ RuntimeException(cause : Throwable)
# RuntimeException(message : String, cause : Throwable
, enableSuppression : boolean, writableStackTrace: boolean)

  - As can be seen, we write two ctors, which esp. accept another *Throwable*: the *cause*. The ctors just delegate to the super class.
    - Along the hierarchy, *RuntimeException*, *Exception* and *Throwable* offer those ctor-overloads dealing with the *cause*, incl. delegation.

- We can now throw the custom exception *CalculationException* and piggyback the original exception/*Throwable*:

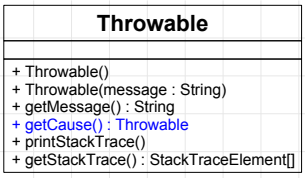
```
public static void calculation() {
        for (int i = 0; i < maxIndex; ++i) {
            try {
                complexCalculation(coordinates[i]);
            } catch (ArithmeticException exc) {
                throw new CalculationException("problem found for data " + coordinates[i], exc);
            }
        }
}
```

  - The idea of the *cause*-parameter is to wrap another exception (often the original exception) to also make it known to handlers.

Wrapping and re-throwing Exceptions – Part 2

- In handlers we can catch *CalculationException* and call its method *getCause()*, which it inherited from *Throwable*:

```
try {
        calculation();
} catch (CalculationException e) {
        System.out.printf("Calculation failed with %s, because of %s%n", e.getMessage(), e.getCause());
}
```



  - *Throwable.getCause()* just accesses the piggybacked *ArithmethicException* we caught and re-threw in *calculation()*.

- In practice piggybacking original exceptions when providing/delegating the relevant ctors in custom exceptions is very useful.

Creating own Exception Types or use JDKs Exception Types? – Part 1

- The other perspective is throwing new *Exception*s objects.
  - Basically, the same *Exception*s, which should not be handled, should also not be thrown as new *Exception*s from own code.
  - Don't throw *Exception* or *Error*.

- There are also some *Exception*s/*RuntimeException*s, which are present in the JDK to be reused in own code. Examples:
  - (1) *java.lang.IllegalStateException*: a "general" exception, it signals the caller that a method was called at an "inappropriate time".
  - (2) *java.lang.IllegalArgumentException*: the idea is to stop the caller from passing incorrect values.
    - We can easily replace our special *InvalidDateException* with the "JDK standard" *IllegalArgumentException*:

```
// <Date.java>
public class Date { // (members hidden)
        public void setMonth(int month) {
                if (1<= month && month <= 12) {
                        this.month = month;
                } else {
                        throw new IllegalArgumentException("illegal argument for new month: " + month);
                }
        }
}
```

  - (3) *java.lang.UnsupportedOperationException*, the idea of *UnsupportedOperationException* is to stop callers from calling unimplemented methods:

```
// A class like UnmodifiableList is defined somewhere in the JDK.
private static class UnmodifiableList implements List {
        public Object remove(int index) {
                throw new UnsupportedOperationException();
        }
}
```

Good to know:
Letting an unimplemented method throw *UnsupportedOperationException* is often called the "optional feature pattern".

Creating own Exception Types or use JDKs Exception Types? – Part 2

- More examples of JDK-*Exception*s to be reused in own code:
  - (4) *java.lang.NoSuchElementException*: signals the caller that a method could not provide a requested value.

The assert Statement – Part 1

- Sometimes, we want to ensure, that a certain condition in our program is always valid, assume following code:

```
public static double reciprocal(double x) {
        return 1/x;
}
```

- Sure, if we pass 0 to this method, the result is undefined, in this case (double) we'll get an "Infinity" value:

```
double cantHaveAResult = reciprocal(0);
// But cantHaveAResult has a result: Infinity
```

  - In most cases this is not what we want. Instead we could add a check and throw an *Exception*, if the argument is 0.

- Instead of checking arguments and throwing *Exception*s explicitly, we can formulate an assertion, which does this for us:

```
public static double reciprocal(double x) {
        assert 0 != x; // The assert statement.
        return 1/x;
}
```

- When we call this method passing 0 and execute this code, we have to set the JVM flag **-ea** (enable assertions):

```
Terminal
NicosMBP:src nico$ java -ea Program
...
Exception in thread "main" java.lang.AssertionError
    Program.reciprocal()
    at Program.main()
NicosMBP:src nico$
```

  - As can be seen, an *AssertionError* is thrown, because the assertion specified in the assert statement was not met.

The assert Statement – Part 2

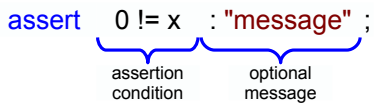
- The assert keyword allows to specify a message:

```
public static double reciprocal(double x) {
        assert 0 != x : "argument must be different from 0";
        return 1/x;
}
```

  - If the assertion is hurt, the thrown *AssertionError* is loaded with this message:

```
Terminal
NicosMBP:src nico$ java -ea Program
...
Exception in thread "main" java.lang.AssertionError: argument must be different from 0
    Program.reciprocal()
    at Program.main()
NicosMBP:src nico$
```

- This yields following formal syntax of the assert statement:



- asserts are not meant to be an alternative to *Exceptions*: they are not for control flow, but shall stop a program forcibly.
  - Notice, that assertions are not enabled by default (JVM flag -ea).
  - In doubt, rather use "ordinary" *Exception*s to handle error cases to be sure, they can be handled always.

null-checks and NPEs

- When we introduced reference types, we underscored the importance code being null-aware.
  - null-awareness means, that code should check, if passed arguments and returned values are null before accessing them.

- With exceptions, we can check values for being not null and just throw *NPE*s, if the check fails:

```
public static int printPrompt(String promptText) {
        if (null == promptText) {
            throw new NullPointerException("promptText mustn't be null");
        }
        System.out.println("Please enter a number:");
        System.out.println(promptText);
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
}
```

- The companion class *Objects* provides the method *requireNonNull()* to do the same in just one statement:

```
public static int printPrompt(String promptText) {
        Objects.requireNonNull(promptText, "promptText mustn't be null");
        System.out.println("Please enter a number:");
        System.out.println(promptText);
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
}
```

Closing Words on Exceptions – General

- (1) *Exception*s, are used to indicate a run time errors, but they can be handled in a structured way.
  - Java allows mighty control over *Exception* handling!
- (2) *Exception* code separates effective code from error handling code syntactically.
  - This is done by try and catch/finally blocks.
- (3) An *Exception* is not directly an error, therefor we say, a "program is in an exceptional state".
  - "The *Exception* was thrown from somewhere."
- (4) And the caller has a chance "deal with this exceptional state".
  - The caller uses try, catch, finally or throw to do this.
- When and where to handle an *Exception* cannot be cast into a set of simple rules.
  - The term *Exception* has nothing to do with frequency they "happen".
    - *Exception*s can be potentially thrown everywhere and every times, e.g. *StackOverflowError* and *OutOfMemoryError*.
  - Often, it is a good idea to handle *Exception*s near to the "location" they were thrown.
  - However *Exception* handling enables us to decide this flexibly.

Closing Words on Exceptions – Mechanics

- *Exception* objects represent error conditions, they can't be ignored, but just...
  - … "happen" (are being thrown) somewhere in the try block. And then:

- Alternative 1: Handled locally ("filtered") in the matching catch-block.

- Alternative 2: Ignored! – The method will be exited immediately and the *Exception* will be forwarded.
  - The stack of the *Exception* will be unwound.
  - If not caught along its stack, the thread will be terminated and the stack trace is written to STDERR.

- Alternative 3: Suppressed, results in a postcondition as if nothing has happened from the *Exception*-handling perspective.

Tips on using Exceptions

- *Exception*s should generally be used over return values to indicate run time errors!
- We can use *Exception*s to better control program execution, when a return statement is not enough:
  - If a very significant thing happens, we want to pass control across methods. return statements only exit the current method!
  - If a very significant thing happens, and it is not possible to guarantee further successful or secure execution.
- Sometimes we must *use Exceptions* to make signaling errors possible at all:
  - Error situations in ctors (incl. field initialization) and operators must be signaled with *Exception*s.
    - Because we can either return no value (ctors) or the return value is part of the operator.
  - Error situations in recursive methods, because we need a deep stack unwinding to transport errors.
  - Mind that in these cases we have no flexibility on the return types, which mandates using *Exception*s.
- It should also be said, that *Exception*s are somewhat problematic when used in lambda expressions.
  - *Exception*s and functional programming (a paradigm, that heavily uses lambda expressions) don't work well together.
  - We'll discuss lambda expressions and functional programming in a future lecture.

- Code using *Exception*s is less portable to other programming language. The control flow depends on specific types.

- In functional programming errors are signaled as special kind of data.

Thank you!