合肥學院
HEFEI UNIVERSITY

OOP with Java
25. Exceptions

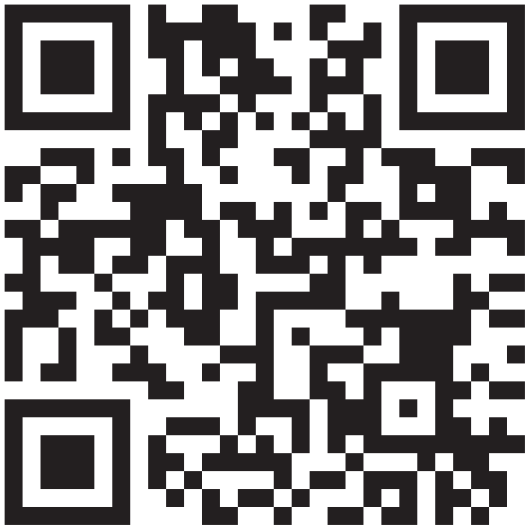
Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline



website

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - properly documenting out code, see Lesson 19: *Documentation with Javadoc*

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - properly documenting out code, see Lesson 19: *Documentation with Javadoc*
  - using generics to reduce the number of type casts in Lesson 21: *Generics*

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - properly documenting out code, see Lesson 19: *Documentation with Javadoc*
  - using generics to reduce the number of type casts in Lesson 21: *Generics*
  - using Java's collections instead of self-made classes, since they have been tested extremely thoroughly (Lesson 22: *Collections,* `equals`*, and* `hashCode`)

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - properly documenting out code, see Lesson 19: *Documentation with Javadoc*
  - using generics to reduce the number of type casts in Lesson 21: *Generics*
  - using Java's collections instead of self-made classes, since they have been tested extremely thoroughly (Lesson 22: *Collections,* `equals`*, and* `hashCode`)
- But we can still expect our code to be used incorrectly sooner or later

Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly and is used correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code focusing on different concerns into packages in Lesson 17: *Packages and* `import`
  - only allowing member variables and methods to be accessed if absolutely necessary and hiding them otherwise in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - properly documenting out code, see Lesson 19: *Documentation with Javadoc*
  - using generics to reduce the number of type casts in Lesson 21: *Generics*
  - using Java's collections instead of self-made classes, since they have been tested extremely thoroughly (Lesson 22: *Collections,* `equals`*, and* `hashCode`)
- But we can still expect our code to be used incorrectly sooner or later
- What do we do if our code receives invalid arguments?

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?
- Should we treat it as 10 or should we crash?

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?
- Should we treat it as 10 or should we crash?
- What should we do if our program is supposed to print the contents of a given file, but the path provided by the user does not exist?

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?
- Should we treat it as 10 or should we crash?
- What should we do if our program is supposed to print the contents of a given file, but the path provided by the user does not exist?
- Should we print nothing (treat it as empty file) or crash?

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?
- Should we treat it as 10 or should we crash?
- What should we do if our program is supposed to print the contents of a given file, but the path provided by the user does not exist?
- Should we print nothing (treat it as empty file) or crash?
- We should crash!

Policy of Dealing with Faulty Data

- Many error situations emerge because of faulty data.
- What should we do if our program expects an integer number between 1 and 10 but receives value 11?
- Should we treat it as 10 or should we crash?
- What should we do if our program is supposed to print the contents of a given file, but the path provided by the user does not exist?
- Should we print nothing (treat it as empty file) or crash?
- We should crash! ALWAYS crash!

Why should we crash?

- There are five major reasons why we should crash

Why should we crash?

- There are five major reasons why we should crash:
  1. If we receive data which violates the specification of what input we should get, the violation is on the user's side. If we do not crash but try to work with the faulty input, we violate the specification of our software as well.

Why should we crash?

- There are five major reasons why we should crash:
  1. If we receive data which violates the specification of what input we should get, the violation is on the user's side. If we do not crash but try to work with the faulty input, we violate the specification of our software as well.
  2. If the input is faulty, this is usually is the result of some unintend error. By crashing, we show to the user that something is wrong. Maybe there was an error in another program he used to generate the input of our program?

Why should we crash?

- There are five major reasons why we should crash:
  1. If we receive data which violates the specification of what input we should get, the violation is on the user's side. If we do not crash but try to work with the faulty input, we violate the specification of our software as well.
  2. If the input is faulty, this is usually is the result of some unintend error. By crashing, we show to the user that something is wrong. Maybe there was an error in another program he used to generate the input of our program?
  3. If we do not crash, the user will not notice that something is wrong. If we try to do the best we can do with the broken input data, what we do may actually be wrong and thus, the error in the input of our program has propagated to an error in the output.

Why should we crash?

- There are five major reasons why we should crash:
  1. If we receive data which violates the specification of what input we should get, the violation is on the user's side. If we do not crash but try to work with the faulty input, we violate the specification of our software as well.
  2. If the input is faulty, this is usually is the result of some unintend error. By crashing, we show to the user that something is wrong. Maybe there was an error in another program he used to generate the input of our program?
  3. If we do not crash, the user will not notice that something is wrong. If we try to do the best we can do with the broken input data, what we do may actually be wrong and thus, the error in the input of our program has propagated to an error in the output.
  4. If we try to "fix" the broken input data, we have implicitly extended the specification for the input of our program. We now need to maintain this extended specification and always consider it in all future updates.

Why should we crash?

- There are five major reasons why we should crash:
  1. If we receive data which violates the specification of what input we should get, the violation is on the user's side. If we do not crash but try to work with the faulty input, we violate the specification of our software as well.
  2. If the input is faulty, this is usually is the result of some unintend error. By crashing, we show to the user that something is wrong. Maybe there was an error in another program he used to generate the input of our program?
  3. If we do not crash, the user will not notice that something is wrong. If we try to do the best we can do with the broken input data, what we do may actually be wrong and thus, the error in the input of our program has propagated to an error in the output.
  4. If we try to "fix" the broken input data, we have implicitly extended the specification for the input of our program. We now need to maintain this extended specification and always consider it in all future updates.
  5. The earlier we crash, the fewer damage will be done.

How should we crash?

- OK, so we should crash.

How should we crash?

- OK, so we should crash.
- How should we crash?

How should we crash?

- OK, so we should crash.
- How should we crash?
- Gracefully

How should we crash?

- OK, so we should crash.
- How should we crash?
- Gracefully:
  - We should provide as much information about what went wrong as possible.

How should we crash?

- OK, so we should crash.
- How should we crash?
- Gracefully:
  - We should provide as much information about what went wrong as possible.
  - If possible, code at a lower level should hand control back to code at a higher level and let that code deal with what went wrong.

How should we crash?

- OK, so we should crash.
- How should we crash?
- Gracefully:
  - We should provide as much information about what went wrong as possible.
  - If possible, code at a lower level should hand control back to code at a higher level and let that code deal with what went wrong.
  - Ideally, we should ask the user for what she wants to do to fix the error.

How should we crash?

- OK, so we should crash.
- How should we crash?
- Gracefully:
  - We should provide as much information about what went wrong as possible.
  - If possible, code at a lower level should hand control back to code at a higher level and let that code deal with what went wrong.
  - Ideally, we should ask the user for what she wants to do to fix the error.
- "Crash" does not mean that our whole program needs to die/exit abnormally, instead, an `Exception` should be generated.

- `Exception`s are special Java classes which store information about an error

- `Exception`s are special Java classes which store information about an error
- If a piece of code detects that something is wrong, it should `throw` an `Exception`

- `Exception`s are special Java classes which store information about an error
- If a piece of code detects that something is wrong, it should `throw` an `Exception`
- Code at a higher level can then `catch` the `Exception` and analyse the error information to find what to do

- Exceptions are special Java classes which store information about an error
- If a piece of code detects that something is wrong, it should throw an Exception
- Code at a higher level can then catch the Exception and analyse the error information to find what to do
- If an Exception is not caught anywhere, the Exception's information is printed to stderr and the calling Thread is terminated (if it is the main Thread of the process, the whole process dies, we learn what Threads are at another time)

- `Exception`s are special Java classes which store information about an error
- If a piece of code detects that something is wrong, it should `throw` an `Exception`
- Code at a higher level can then `catch` the `Exception` and analyse the error information to find what to do
- If an `Exception` is not caught anywhere, the `Exception`'s information is printed to `stderr` and the calling `Thread` is terminated (if it is the main `Thread` of the process, the whole process dies, we learn what `Thread`s are at another time)
- Many actions in Java can cause `Exception`s

Example for integer division by 0

Listing: Example program dividing an integer by 0

```
/** This class shows what happens if we divide an integer by 0
    */
public class IntegerDivisionByZero {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    System.out.println(10 / 0);
  }
}
```

Listing: Output of this program to stderr

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

Example for a wrong class cast

Listing: Example for a wrong class cast

```
/** This class shows what happens if cast an object to a wrong
    class */
public class WrongTypeCast {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    Object o = new Integer(34);
    String s = ((String)o);
    System.out.println(s);
  }
}
```

Listing: Output of this program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

Example for working with a `null` pointer

Listing: Example for working with a `null` pointer

```
/** What happens if we access a method of a null object? */
public class NullPointer {
  /** get the fifth array element */
  static void printToString(final Object o) {
    System.out.println(o.toString());
  }
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    printToString(null);
  }
}
```

Listing: Output of this program to stderr

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

Accessing an array element outside the valid range

Listing: Example for accessing an array element outside the valid range

```
/** This class shows what happens if we access an array element out of the
    bounds */
public class ArrayIndexOutOfBounds {
  /** get the fifth array element */
  static int getFifth(final int[] array) {
    return array[4];
  }
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    System.out.println(getFifth(new int[] {1, 2, 3}));
  }
}
```

Listing: Output of this program to stderr

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

How to Read Exceptions

- OK, good, if we do something nasty, our program will hopefully crash with an `Exception`

How to Read Exceptions

- OK, good, if we do something nasty, our program will hopefully crash with an `Exception`
- How do we read an `Exception`?

How to Read Exceptions

- OK, good, if we do something nasty, our program will hopefully crash with an `Exception`
- How do we read an `Exception`?
- It happens soo often that a program crashes with an `Exception`, and the (student) programmer then says: Oh, the program has crashed. Where could the error be?

How to Read Exceptions

- OK, good, if we do something nasty, our program will hopefully crash with an `Exception`
- How do we read an `Exception`?
- It happens soo often that a program crashes with an `Exception`, and the (student) programmer then says: Oh, the program has crashed. Where could the error be?
- In 99% of the cases, the `Exception` print tells you where it is.

How to Read Exceptions

- OK, good, if we do something nasty, our program will hopefully crash with an `Exception`
- How do we read an `Exception`?
- It happens soo often that a program crashes with an `Exception`, and the (student) programmer then says: Oh, the program has crashed. Where could the error be?
- In 99% of the cases, the `Exception` print tells you where it is.
- In the remaining 1%, it at least tells you where to look.

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

- "There was a division by 0"

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

- "There was a division by 0"
- It took place in class/file IntegerDivisionByZero.java at code line 6

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

- "There was a division by 0"
- It took place in class/file IntegerDivisionByZero.java at code line 6
- Which happens to be in method `IntegerDivisionByZero.main`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at IntegerDivisionByZero.main(IntegerDivisionByZero.java:6)
```

- "There was a division by 0"
- It took place in class/file IntegerDivisionByZero.java at code line 6
- Which happens to be in method `IntegerDivisionByZero.main`
- How about putting a break point there and using the debugger?

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"
- It seems that you tried to cast an instance of type `java.lang.Integer` to `java.lang.String`

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"
- It seems that you tried to cast an instance of type `java.lang.Integer` to `java.lang.String`
- `java.lang.Integer` appears to not be a subclass of `java.lang.String`, so you are not allowed to make such a class cast

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"
- It seems that you tried to cast an instance of type `java.lang.Integer` to `java.lang.String`
- `java.lang.Integer` appears to not be a subclass of `java.lang.String`, so you are not allowed to make such a class cast
- It took place in class/file WrongTypeCast.java at code line 7

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"
- It seems that you tried to cast an instance of type `java.lang.Integer` to `java.lang.String`
- `java.lang.Integer` appears to not be a subclass of `java.lang.String`, so you are not allowed to make such a class cast
- It took place in class/file WrongTypeCast.java at code line 7
- Which happens to be in method `WrongTypeCast.main`

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
    cast to java.lang.String
  at WrongTypeCast.main(WrongTypeCast.java:7)
```

- "There was a wrong class cast"
- It seems that you tried to cast an instance of type `java.lang.Integer` to `java.lang.String`
- `java.lang.Integer` appears to not be a subclass of `java.lang.String`, so you are not allowed to make such a class cast
- It took place in class/file WrongTypeCast.java at code line 7
- Which happens to be in method `WrongTypeCast.main`
- How about putting a break point there and using the debugger?

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5
- Which happens to be in method `NullPointer.printToString`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5
- Which happens to be in method `NullPointer.printToString`
- This method was invoked from method `NullPointer.main` at line 10 of class/file NullPointer.java

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5
- Which happens to be in method `NullPointer.printToString`
- This method was invoked from method `NullPointer.main` at line 10 of class/file NullPointer.java
- So either the code in `NullPointer.printToString` is doing something wrong

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5
- Which happens to be in method `NullPointer.printToString`
- This method was invoked from method `NullPointer.main` at line 10 of class/file NullPointer.java
- So either the code in `NullPointer.printToString` is doing something wrong
- Or it was called with wrong parameters by `NullPointer.main`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.NullPointerException
  at NullPointer.printToString(NullPointer.java:5)
  at NullPointer.main(NullPointer.java:10)
```

- "There was an attempt to de-reference a `null` pointer"
- Some variable was `null` but obviously should not have been `null`
- The error took place in class/file NullPointer.java at code line 5
- Which happens to be in method `NullPointer.printToString`
- This method was invoked from method `NullPointer.main` at line 10 of class/file NullPointer.java
- So either the code in `NullPointer.printToString` is doing something wrong
- Or it was called with wrong parameters by `NullPointer.main`
- You should put a break point at line 10 of class/file NullPointer.java to see what's going on

Reading Exceptions

Listing: Output of a program to stderr

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5
- Which happens to be in method `ArrayIndexOutOfBounds.getFifth`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5
- Which happens to be in method `ArrayIndexOutOfBounds.getFifth`
- This method was invoked from method `ArrayIndexOutOfBounds.main` at line 10 of class/file ArrayIndexOutOfBounds.java

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5
- Which happens to be in method `ArrayIndexOutOfBounds.getFifth`
- This method was invoked from method `ArrayIndexOutOfBounds.main` at line 10 of class/file ArrayIndexOutOfBounds.java
- So either the code in `ArrayIndexOutOfBounds.getFifth` is doing something wrong

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5
- Which happens to be in method `ArrayIndexOutOfBounds.getFifth`
- This method was invoked from method `ArrayIndexOutOfBounds.main` at line 10 of class/file ArrayIndexOutOfBounds.java
- So either the code in `ArrayIndexOutOfBounds.getFifth` is doing something wrong
- Or it was called with wrong parameters by `ArrayIndexOutOfBounds.main`

Reading Exceptions

Listing: Output of a program to `stderr`

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
  at ArrayIndexOutOfBounds.getFifth(ArrayIndexOutOfBounds.java:5)
  at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:10)
```

- "There was an attempt to access an array element with a invalid index"
- The index `4` is out of bounds of the array (which must have had less than 5 elements)
- The error took place in class/file ArrayIndexOutOfBounds.java at code line 5
- Which happens to be in method `ArrayIndexOutOfBounds.getFifth`
- This method was invoked from method `ArrayIndexOutOfBounds.main` at line 10 of class/file ArrayIndexOutOfBounds.java
- So either the code in `ArrayIndexOutOfBounds.getFifth` is doing something wrong
- Or it was called with wrong parameters by `ArrayIndexOutOfBounds.main`
- You should put a break point at line 10 of class/file ArrayIndexOutOfBounds.java to see what's going on

Debugging

- With the information from the `Exception`s, we have a good chance to find the bug

Debugging

- With the information from the `Exception`s, we have a good chance to find the bug
- Reading of exceptions is an important skill

Catching Exceptions

- Ok, so if an exception occurs, our program will die.

Catching Exceptions

- Ok, so if an exception occurs, our program will die.
- Does not look very useful, because I could as well do `System.err.println(...);` and then `System.exit(1);` to print error infos and then quit the program

Catching Exceptions

- Ok, so if an exception occurs, our program will die.
- Does not look very useful, because I could as well do `System.err.println(...);` and then `System.exit(1);` to print error infos and then quit the program
- Well, we can `catch` exceptions at a higher code level, read their information, and take actions.

Catching Exceptions

- Ok, so if an exception occurs, our program will die.
- Does not look very useful, because I could as well do `System.err.println(...);` and then `System.exit(1);` to print error infos and then quit the program
- Well, we can `catch` exceptions at a higher code level, read their information, and take actions.
- The code which might throw an exception of type `T` is therefore wrapped into

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`,
  - the program stores it in a variable called `error`

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`,
  - the program stores it in a variable called `error` and
  - jumps directly to the start of the code in the `catch(T error){ ... }` block

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`,
  - the program stores it in a variable called `error` and
  - jumps directly to the start of the code in the `catch(T error){ ... }` block,
  - skipping over whatever code in `try { ...code...}` comes ofter the instruction causing the error

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`,
  - the program stores it in a variable called `error` and
  - jumps directly to the start of the code in the `catch(T error){ ... }` block,
  - skipping over whatever code in `try { ...code...}` comes ofter the instruction causing the error
- If an exception (which is not a subclass of `T`) occurs, the control is passed to a corresponding `catch` block in a higher level of code, if any, or the program exits (Thread dies)

Catching Exceptions

```
try { ...code...} catch(T error){ ... do something (using infos from error)... }
```

- If an exception of type (or subclass of) `T` occurs somewhere in the code inside the `try { ...code...}`,
  - the program stores it in a variable called `error` and
  - jumps directly to the start of the code in the `catch(T error){ ... }` block,
  - skipping over whatever code in `try { ...code...}` comes ofter the instruction causing the error
- If an exception (which is not a subclass of `T`) occurs, the control is passed to a corresponding `catch` block in a higher level of code, if any, or the program exits (Thread dies)
- If no exception occurs, the `try { ...code...}` block completes normally and the `catch(T error){ ... }` block is ignored

An example for catching and processing an exception

Listing: An example for catching and processing an exception

```
/** a modifiable number */
public class ModifiableNumber {
  /** the number value */
  private int value;
  /** are we ok */
  private boolean ok;

  /** create */
  public ModifiableNumber() {  this.ok = true; }
  /** add a value */
  public void add(final int v) { this.value += v; }

  /** divide by a value */
  public void divide(final int v) {
    try {                                          // if an ArithmeticException occurs between here and the } two lines below
      this.value /= v;                             // the control jumps directly to the code in the catch block
      System.out.println("Correct division by " + v); // this line here will only be reached if the division went OK //$NON-NLS-1$
    } catch(ArithmeticException error) { // (only) iff an ArithmeticException occurs, it is stored in variable error and the 2 lines below are executed
      System.out.println("Error when trying to divide " + this.value + " by " + v + ": " + error.getMessage()); //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
      this.ok = false;
    }
  }

  /** convert to string */
  @Override
  public String toString() {
    return (this.ok ? ""+this.value : "error"); //$NON-NLS-1$  //$NON-NLS-2$
  }

  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    ModifiableNumber n = new ModifiableNumber();
    System.out.println(n); // 0
    n.add(100);
    System.out.println(n); // 100
    n.divide(10);          // Correct division by 10
    System.out.println(n); // 10
    n.add(1);
    System.out.println(n); // 11
    n.divide(0);           // Error when trying to divide 11 by 0: / by zero
    System.out.println(n); // error
    n.add(1);
    System.out.println(n); // error
  }
}
```

Throwing Exceptions

- Now we clearly do not just want to catch exceptions that come somewhere from the bowels of Java

Throwing Exceptions

- Now we clearly do not just want to catch exceptions that come somewhere from the bowels of Java
- We want to throw exceptions as soon as we detect an error

Throwing Exceptions

- Now we clearly do not just want to catch exceptions that come somewhere from the bowels of Java
- We want to throw exceptions as soon as we detect an error
- Rule of Thumb: Check all parameters and throw exceptions as soon as possible!

Throwing Exceptions

- Now we clearly do not just want to catch exceptions that come somewhere from the bowels of Java
- We want to throw exceptions as soon as we detect an error
- Rule of Thumb: Check all parameters and throw exceptions as soon as possible!
- Reason: Exception stack trace tells us where the exception was thrown.

Throwing Exceptions

- Now we clearly do not just want to catch exceptions that come somewhere from the bowels of Java
- We want to throw exceptions as soon as we detect an error
- Rule of Thumb: Check all parameters and throw exceptions as soon as possible!
- Reason: Exception stack trace tells us where the exception was thrown. The earlier we throw the exception, the closer this point will be to the source of the error.
- We can throw exception doing something like

```
throw new ExceptionType(exceptionParameters)
```

An example for computing factorials

Listing: An example for computing factorials: $21!$ is wrong

```
/** An example program computing factorials, defined as i! = ∏_{j=1}^{i} j. */
public class Factorial {

  /** Compute the factorial of i
   * @return i! = ∏_{j=1}^{i} j */
  static long factorial(int i) {
    long result = 1L;
    for(int v = i; v > 1; --v) {
      result *= v;
    }
    return result;
  }

  /** The main routine
   * @param args
   *           we ignore this parameter */
  public static final void main(String[] args) {
    for(int i = 0; i <= 21; i++){ // printing the first 22 factorials, where 21! is wrong
      System.out.print(i);
      System.out.print("! = "); //$NON-NLS-1$
      System.out.println(factorial(i));
    }
  }
}
```

Computing factorials and throwing exceptions

Listing: An example for computing factorials: exception at 21!

```
/** An example program computing factorials, defined as i! = ∏_{j=1}^{i} j. */
public class FactorialException {

  /** Compute the factorial of i
   * @return i! = ∏_{j=1}^{i} j */
  static long factorial(int i) {
    long result = 1L;
    if(i < 0)  { throw new IllegalArgumentException(i + "! is undefined."); } //$NON-NLS-1$
    if(i > 20) { throw new ArithmeticException(i + "! exceeds the range of long."); } //$NON-NLS-1$
    for(int v = i; v > 1; --v) {
      result *= v;
    }
    return result;
  }

  /** The main routine
   * @param args
   *          we ignore this parameter */
  public static final void main(String[] args) {
    for(int i = 0; i <= 21; i++){ // printing the first 22 factorials, exception at 21!
      System.out.print(i);
      System.out.print("! = "); //$NON-NLS-1$
      System.out.println(factorial(i));
    }
  }
}
```

Catching Multiple Different Exceptions

- We can catch multiple different exceptions by specifying multiple `catch` statements

Catching Multiple Different Exceptions

- We can catch multiple different exceptions by specifying multiple `catch` statements
- At most one of them will be executed, the first one whose type parameter is a superclass of the actual exception

Computing factorials and catching multiple exceptions

Listing: Computing factorials and catching multiple exceptions

```java
/** An example program computing factorials, defined as i! = ∏_{j=1}^{i} j. */
public class FactorialExceptionCatch {

  /** Compute the factorial of i
   * @return i! = ∏_{j=1}^{i} j */
  static long factorial(int i) {
    long result = 1L;
    if(i < 0)  { throw new IllegalArgumentException(i + "!␣is␣undefined."); } //$NON-NLS-1$
    if(i > 20) { throw new ArithmeticException(i + "!␣exceeds␣the␣range␣of␣long."); } //$NON-NLS-1$
    for(int v = i; v > 1; --v) {
      result *= v;
    }
    return result;
  }

  /** The main routine
   * @param args
   *          we ignore this parameter */
  public static final void main(String[] args) {
    for(int i = -1; i <= 21; i++){ // this time starting loop at -1...
      try {
        long f = factorial(i);
        System.out.println(i + "!␣=␣" + f); //$NON-NLS-1$
      } catch (IllegalArgumentException error) {
        System.out.println("Illegal␣argument:␣" + error.getMessage()); //$NON-NLS-1$
      } catch (ArithmeticException error) {
        System.out.println("Arithmetic␣error:␣" + error.getMessage()); //$NON-NLS-1$
      }
    }
  }
}
```

Making our Own Exceptions

- Exceptions are objects

Making our Own Exceptions

- Exceptions are objects
- There is a whole class hierarchy of exceptions

Making our Own Exceptions

- Exceptions are objects
- There is a whole class hierarchy of exceptions
- The base class of all (checked) exceptions in the class `Exception`

Making our Own Exceptions

- Exceptions are objects
- There is a whole class hierarchy of exceptions
- The base class of all (checked) exceptions in the class `Exception`
- We can make our own exception classes

Making our Own Exceptions

- Exceptions are objects
- There is a whole class hierarchy of exceptions
- The base class of all (checked) exceptions in the class `Exception`
- We can make our own exception classes
- Let us therefore revisit our `BankAccount` example from Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*

The original `BankAccount` class

Listing: The original `BankAccount` class

```
/** A class for a bank account with complete encapsulation and the final keyword */
public final class BankAccount { // we declare the class final, we don't allow subclassing

  /** the account number: clearly private, clearly never changes, so it should be final */
  private final String accountNumber;
  /** the amount of money in the account in cents: also private */
  private long balance; // we use long, not double, because an account cannot have "fractional" cents

  /** create a new bank account with balance 0 */
  public BankAccount(final String number){ // number parameter is final, it cannot be changed inside constructor
    this.accountNumber = number;                // why would we want to change it anyway..
  }

  /** get the account's balance */
  public final double getBalance() {  // the method is marked as final. If the class was not already marked as final,
    return this.balance;              // then it would still not be possible to override the method
  }

  /** add some money to the bank account */
  public final void deposit(final long amount) {
    if((amount > 0L) && (amount < 1_000_000_00L)) { // sanity check: you can only deposit a positive amount
      this.balance += amount;                       // of money, and anything above 1 million is probably an error
    } else { // an invalid amount cannot be put into the account
      System.out.println("Invalid deposit amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  /** withdraw some money from the bank account */
  public final void withdraw(final long amount) {
    if((amount > 0L) && (amount < 1_000_00L)) { // sanity check: you can only withdraw a positive amount of
      this.balance -= amount;                   // money and at most 1000 RMB at once
    } else {
      System.out.println("Invalid withdrawal amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  @Override
  public final String toString() {
    return '(' + this.accountNumber + ": " + this.balance + ')'; //$NON-NLS-1$
  }
}
```

A test of the `BankAccount` class

Listing: The `BankAccountTest`

```
/** testing the plain bank account class */
public class BankAccountTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    BankAccount account;

    account = new BankAccount("123"); //$NON-NLS-1$

    System.out.println(account);      // (123: 0)
    account.deposit(900_000_00L);
    System.out.println(account);      // (123: 90000000)
    account.deposit(11_000_000_00L); // Invalid deposit amount 1100000000 for account (123: 90000000)
    System.out.println(account);      // (123: 90000000) <---| notice how the program continues, although
                                      //                     | the 11'000'000 RMB have not been added to
    account.withdraw(900_00L);        //                     | the account, which now has a different value
    System.out.println(account);      // (123: 89910000)     | from what the user exceptions?
    account.withdraw(3_000_00L);      // Invalid withdrawal amount 300000 for account (123: 89910000)
    System.out.println(account);      // (123: 89910000)
  }
}
```

What is wrong

- The above code shows that faulty transactions are not performed

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- . . . where it might be ignored/overlooked by the user

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- ...where it might be ignored/overlooked by the user
- ...who would then think that the bank account should have well above 11'000'000 RMB

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- ...where it might be ignored/overlooked by the user
- ...who would then think that the bank account should have well above 11'000'000 RMB
- After a faulty transaction which was not carried out, more transactions are performed as if nothing happened

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- ...where it might be ignored/overlooked by the user
- ...who would then think that the bank account should have well above 11'000'000 RMB
- After a faulty transaction which was not carried out, more transactions are performed as if nothing happened
- We should throw an exception

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- ...where it might be ignored/overlooked by the user
- ...who would then think that the bank account should have well above 11'000'000 RMB
- After a faulty transaction which was not carried out, more transactions are performed as if nothing happened
- We should throw an exception
- And maybe even put some special data into this, say, about the bank account

What is wrong

- The above code shows that faulty transactions are not performed
- However, the program itself does never receive notice of that, it is just printed to `stdout`
- ...where it might be ignored/overlooked by the user
- ...who would then think that the bank account should have well above 11'000'000 RMB
- After a faulty transaction which was not carried out, more transactions are performed as if nothing happened
- We should throw an exception
- And maybe even put some special data into this, say, about the bank account
- To prevent further transactions and to force the program to recognize that something went wrong

A new type of exception: `TransactionException`

Listing: The `TransactionException` class

```
/** our own exception class */
public class TransactionException extends IllegalArgumentException {

  /**the serial version uid: don't worry about that right now */
  private static final long serialVersionUID = 1L;

  /** the bank account */
  final BankAccountWithExceptions account;

  /** create the exception */
  public TransactionException(String message, BankAccountWithExceptions _account) {
    super(message);
    this.account = _account;
  }
}
```

A Bank Account class using our new exception

Listing: The BankAccountWithExceptions class

```java
/** A class for a bank account throwing an (unchecked) exception on error */
public final class BankAccountWithExceptions { // we declare the class final, we don't allow subclassing

  /** the account number: clearly private, clearly never changes, so it should be final */
  private final String accountNumber;
  /** the amount of money in the account in cents: also private */
  private long balance; // we use long, not double, because an account cannot have "fractional" cents

  /** create a new bank account with balance 0 */
  public BankAccountWithExceptions(final String number){ // number parameter is final, it cannot be changed inside constructor
    this.accountNumber = number;                 // why would we want to change it anyway..
  }

  /** get the account's balance */
  public final double getBalance() {  // the method is marked as final. If the class was not already marked as final,
    return this.balance;              // then it would still not be possible to override the method
  }

  /** add some money to the bank account */
  public final void deposit(final long amount) {
    if((amount > 0L) && (amount < 1_000_000_00L)) { // sanity check: you can only deposit a positive amount
      this.balance += amount;                       // of money, and anything above 1 million is probably an error
    } else { // an invalid amount cannot be put into the account
      throw new TransactionException("Invalid deposit amount " + amount, this); //$NON-NLS-1$
    }
  }

  /** withdraw some money from the bank account */
  public final void withdraw(final long amount) {
    if((amount > 0L) && (amount < 1_000_00L)) { // sanity check: you can only withdraw a positive amount of
      this.balance -= amount;                   // money and at most 1000 RMB at once
    } else {
      throw new TransactionException("Invalid withdrawal amount " + amount, this); //$NON-NLS-1$
    }
  }

  @Override
  public final String toString() {
    return '(' + this.accountNumber + ": " + this.balance + ')'; //$NON-NLS-1$
  }
}
```

A test of the `BankAccountWithExceptions` class

Listing: The `BankAccountWithExceptionsTestWithoutTryCatch`

```
/** testing the exception-throwing bank account */
public class BankAccountWithExceptionsTestWithoutTryCatch {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    BankAccountWithExceptions account;

    account = new BankAccountWithExceptions("123"); //$NON-NLS-1$

    System.out.println(account);      // (123: 0)
    account.deposit(900_000_00L);
    System.out.println(account);      // (123: 90000000)
    account.deposit(11_000_000_00L); // here, an exception is thrown and all further transactions are skipped
    System.out.println(account);      // never reached, as the above line throws an exception and the program
                                      //                terminates, printing the stack trace to stderr
    account.withdraw(900_00L);        // never reached
    System.out.println(account);      // never reached
    account.withdraw(3_000_00L);      // never reached
    System.out.println(account);      // never reached
  }
}
```

A test of the `BankAccountWithExceptions` class with `try ... catch`

Listing: The `BankAccountWithExceptionsTest`

```
/** testing the exception-throwing bank account with try...catch */
public class BankAccountWithExceptionsTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    BankAccountWithExceptions account;

    account = new BankAccountWithExceptions("123"); //$NON-NLS-1$

    try {
      System.out.println(account);      // (123: 0)
      account.deposit(900_000_00L);
      System.out.println(account);      // (123: 90000000)
      account.deposit(11_000_000_00L); // here, an exception is thrown and all further transactions are skipped
      System.out.println(account);      // never reached

      account.withdraw(900_00L);        // never reached
      System.out.println(account);      // never reached
      account.withdraw(3_000_00L);      // never reached
      System.out.println(account);      // never reached
    } catch (TransactionException trans) { // Invalid deposit amount 1100000000 for bank account (123: 90000000)
      System.out.println(trans.getMessage() + "␣for␣bank␣account␣" + trans.account); //$NON-NLS-1$
    }
  }
}
```

Awareness of Exceptions

- If we think about it, the `TransactionException` is, sort of, part of the specification of the methods in `BankAccountWithExceptions`

Awareness of Exceptions

- If we think about it, the `TransactionException` is, sort of, part of the specification of the methods in `BankAccountWithExceptions`
- Whenever we use that code, we should be aware that `TransactionException`s may take place

Awareness of Exceptions

- If we think about it, the `TransactionException` is, sort of, part of the specification of the methods in `BankAccountWithExceptions`
- Whenever we use that code, we should be aware that `TransactionException`s may take place
- However, we could compile the `BankAccountWithExceptionsTest` just as fine without the `try ... catch` stuff...

Awareness of Exceptions

- If we think about it, the `TransactionException` is, sort of, part of the specification of the methods in `BankAccountWithExceptions`
- Whenever we use that code, we should be aware that `TransactionException`s may take place
- However, we could compile the `BankAccountWithExceptionsTest` just as fine without the `try ... catch` stuff...
- This is not a good idea

Awareness of Exceptions

- If we think about it, the `TransactionException` is, sort of, part of the specification of the methods in `BankAccountWithExceptions`
- Whenever we use that code, we should be aware that `TransactionException`s may take place
- However, we could compile the `BankAccountWithExceptionsTest` just as fine without the `try ... catch` stuff...
- This is not a good idea
- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*
  - Checked exceptions do not inherit from `RuntimeException` (our `TransactionException` does indirectly), but from `Exception`, `Throwable`, or another one of their subclasses

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*
  - Checked exceptions do not inherit from `RuntimeException` (our `TransactionException` does indirectly), but from `Exception`, `Throwable`, or another one of their subclasses
  - Every method that may throw such an exception must specify in its signature via `throws` that it may do so

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*
  - Checked exceptions do not inherit from `RuntimeException` (our `TransactionException` does indirectly), but from `Exception`, `Throwable`, or another one of their subclasses
  - Every method that may throw such an exception must specify in its signature via `throws` that it may do so
  - Any user of this method must wrap it in a corresponding `try ... catch` block

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*
  - Checked exceptions do not inherit from `RuntimeException` (our `TransactionException` does indirectly), but from `Exception`, `Throwable`, or another one of their subclasses
  - Every method that may throw such an exception must specify in its signature via `throws` that it may do so
  - Any user of this method must wrap it in a corresponding `try ... catch` block
  - Or otherwise the code does not compile.

Checked Exceptions

- Ideally, we want to force programmers using our code to be aware and to explicitly handle the case that our exceptions may be thrown
- For this, there a *checked exceptions*
  - Checked exceptions do not inherit from `RuntimeException` (our `TransactionException` does indirectly), but from `Exception`, `Throwable`, or another one of their subclasses
  - Every method that may throw such an exception must specify in its signature via `throws` that it may do so
  - Any user of this method must wrap it in a corresponding `try ... catch` block
  - Or otherwise the code does not compile.
- Checked exceptions of class `E` can be documented using `@throws E meaning` Javadoc

A new type of checked exceptions: `CheckedTransactionException`

Listing: The `CheckedTransactionException` class

```
/** our own exception class inheriting from Exception, i.e., being a checked
    exception */
public class CheckedTransactionException extends Exception {

  /**the serial version uid: don't worry about that right now */
  private static final long serialVersionUID = 1L;

  /** the bank account */
  final BankAccountWithCheckedExceptions account;

  /** create the exception */
  public CheckedTransactionException(String message,
      BankAccountWithCheckedExceptions _account) {
    super(message);
    this.account = _account;
  }
}
```

A Bank Account class using our new checked exception

Listing: The BankAccountWithCheckedExceptions class

```
/** A class for a bank account throwing a checked exception on error */
public final class BankAccountWithCheckedExceptions { // we declare the class final, we don't allow subclassing

  /** the account number: clearly private, clearly never changes, so it should be final */
  private final String accountNumber;
  /** the amount of money in the account in cents: also private */
  private long balance; // we use long, not double, because an account cannot have "fractional" cents

  /** create a new bank account with balance 0 */
  public BankAccountWithCheckedExceptions(final String number){ // number parameter is final, it cannot be changed inside constructor
    this.accountNumber = number;                // why would we want to change it anyway..
  }

  /** get the account's balance */
  public final double getBalance() {  // the method is marked as final. If the class was not already marked as final,
    return this.balance;              // then it would still not be possible to override the method
  }

  /** add some money to the bank account
   * @throws CheckedTransactionException if the amount if invalid */
  public final void deposit(final long amount) throws CheckedTransactionException { // throws declarion necessary!!!
    if((amount > 0L) && (amount < 1_000_000_00L)) { // sanity check: you can only deposit a positive amount
      this.balance += amount;                       // of money, and anything above 1 million is probably an error
    } else { // an invalid amount cannot be put into the account
      throw new CheckedTransactionException("Invalid deposit amount " + amount, this); //$NON-NLS-1$
    }
  }

  /** withdraw some money from the bank account
   * @throws CheckedTransactionException if the amount if invalid */
  public final void withdraw(final long amount) throws CheckedTransactionException { // throws declarion necessary!!!
    if((amount > 0L) && (amount < 1_000_00L)) { // sanity check: you can only withdraw a positive amount of
      this.balance -= amount;                   // money and at most 1000 RMB at once
    } else {
      throw new CheckedTransactionException("Invalid withdrawal amount " + amount, this); //$NON-NLS-1$
    }
  }

  @Override
  public final String toString() {
    return '(' + this.accountNumber + ": " + this.balance + ')'; //$NON-NLS-1$
  }
}
```

Listing: The BankAccountWithCheckedExceptionsTest

```
/** testing the checked-exception throwing bank account with check exceptions: MUST have try...catch */
public class BankAccountWithCheckedExceptionsTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    BankAccountWithCheckedExceptions account;

    account = new BankAccountWithCheckedExceptions("123"); //$NON-NLS-1$

    try { // without this try...catch, this class cannot be compiled anymore!
      System.out.println(account);      // (123: 0)
      account.deposit(900_000_00L);
      System.out.println(account);      // (123: 90000000)
      account.deposit(11_000_000_00L); // here, an exception is thrown and all further transactions are skipped
      System.out.println(account);      // never reached

      account.withdraw(900_00L);        // never reached
      System.out.println(account);      // never reached
      account.withdraw(3_000_00L);      // never reached
      System.out.println(account);      // never reached
    } catch (CheckedTransactionException trans) { // Invalid deposit amount 1100000000 for bank account (123:
        90000000)
      System.out.println(trans.getMessage() + "␣for␣bank␣account␣" + trans.account); //$NON-NLS-1$
    }
  }
}
```

Ensuring Final Action

- Sometimes, we want to ensure that a specific action is always performed

Ensuring Final Action

- Sometimes, we want to ensure that a specific action is always performed, regardless whether a `try` block fails or not

Ensuring Final Action

- Sometimes, we want to ensure that a specific action is always performed, regardless whether a `try` block fails or not
- For this, we can specify a `finally` block, either instead or after a `catch` block

Ensuring Final Action

- Sometimes, we want to ensure that a specific action is always performed, regardless whether a `try` block fails or not
- For this, we can specify a `finally` block, either instead or after a `catch` block
- The code in this block will always be executed

Ensuring Final Action

- Sometimes, we want to ensure that a specific action is always performed, regardless whether a `try` block fails or not
- For this, we can specify a `finally` block, either instead or after a `catch` block
- The code in this block will always be executed
- (well, except if you kill the Java process irregularly, e.g., if you plug the power cord from the PC. . .)

An example for a `try ... finally` block

Listing: An example for a `try ... finally` block

```
/** a try ... finally block*/
public class TryFinally {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    int a = 0;
    try {
      System.out.println(a); // 0
      a++;
      System.out.println(a); // 1
      a *= 3;
      System.out.println(a); // 3
      a /= 0;
      System.out.println(a); // never reached
      a -= 5;
      System.out.println(a); // never reached
    } finally {
      System.out.println(a); // 3
    }
  }
}
```

An example for a `try ... catch ... finally` block

Listing: An example for a `try ... catch ... finally` block

```
/** a try ... catch ... finally block*/
public class TryCatchFinally {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    int a = 0;
    try {
      System.out.println(a); // 0
      a++;
      System.out.println(a); // 1
      a *= 3;
      System.out.println(a); // 3
      a /= 0;
      System.out.println(a); // never reached
      a -= 5;
      System.out.println(a); // never reached
    } catch(ArithmeticException error) {
      System.out.println("Error: " + //$NON-NLS-1$
                         error.getMessage()); // Error: / by zero
    } finally {
      System.out.println(a); // 3
    }
  }
}
```

Dealing with Resources

- The most common case where we definitely want to do something regardless of how things go is when we deal with (operating system) resources such as files or network sockets

Dealing with Resources

- The most common case where we definitely want to do something regardless of how things go is when we deal with (operating system) resources such as files or network sockets:
- Such resources are limited in the number and require lots of, well, resources such as memory

Dealing with Resources

- The most common case where we definitely want to do something regardless of how things go is when we deal with (operating system) resources such as files or network sockets:
- Such resources are limited in the number and require lots of, well, resources such as memory
- We definitely want to close any file we have opened, regardless whether reading from it succeeded or not

Dealing with Resources

- The most common case where we definitely want to do something regardless of how things go is when we deal with (operating system) resources such as files or network sockets:
- Such resources are limited in the number and require lots of, well, resources such as memory
- We definitely want to close any file we have opened, regardless whether reading from it succeeded or not
- We definitely want to close all network connections we have opened, regardless whether they went well or not

Dealing with Resources

- The most common case where we definitely want to do something regardless of how things go is when we deal with (operating system) resources such as files or network sockets:
- Such resources are limited in the number and require lots of, well, resources such as memory
- We definitely want to close any file we have opened, regardless whether reading from it succeeded or not
- We definitely want to close all network connections we have opened, regardless whether they went well or not
- For this purpose, Java provides a special type of block: try-with-resource

try-with-ressource

- a try-with-ressource statement looks quite similar to a normal `try/catch` block

try-with-ressource

- a try-with-ressource statement looks quite similar to a normal `try/catch` block
- But in its head, you allocate a resource and at its bottom, it will automatically be disposed

- a try-with-ressource statement looks quite similar to a normal `try/catch` block
- But in its head, you allocate a resource and at its bottom, it will automatically be disposed
- In order to deal with a resource of type `R`, you do the following

```
try(R resource = ..create..){ ... }
```

try-with-ressource

- a try-with-ressource statement looks quite similar to a normal `try/catch` block
- But in its head, you allocate a resource and at its bottom, it will automatically be disposed
- In order to deal with a resource of type `R`, you do the following `try(R resource = ..create..){ ... }`
- At the closing `}`, `resource` is disposed

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`
- This interface only has one method, `void close()throws Exception`

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`
- This interface only has one method, `void close()throws Exception`:

Listing: The abridged code of `java.lang.AutoCloseable`

```
package java.lang;

/**
 * An object that may hold resources (such as file or socket handles) until it is closed. The {@link #close()}
 * method of an {@code AutoCloseable} object is called automatically when exiting a {@code try}-with-resources
 * block for which the object has been declared in the resource specification header. This construction ensures
 * prompt release, avoiding resource exhaustion exceptions and errors that may otherwise occur.
 * ...
 */
public interface AutoCloseable {
    /**
     * Closes this resource, relinquishing any underlying resources. This method is invoked automatically on
     * objects managed by the {@code try}-with-resources statement.
     *
     * ...
     *
     * However, implementers of this interface are strongly encouraged to make their {@code close} methods
     * idempotent.
     *
     * @throws Exception if this resource cannot be closed
     */
    void close() throws Exception;
}
```

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`
- This interface only has one method, `void close()throws Exception`
- The try-with-resource statement then is basically equivalent to

```
R resource = ...create...; try { ... } finally { resource.close(); resource = null; }
```

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`
- The try-with-resource statement then is basically equivalent to

```
R resource = ...create...; try { ... } finally { resource.close(); resource = null; }
```

- All of Java's resources classes implement either this interface directly, or its sub-interface `java.io.Closeable` for I/O resources

How does that work?

- All resources which should be closeable via the try-with-resource statement must implement the interface `java.lang.AutoCloseable`
- The try-with-resource statement then is basically equivalent to

```
R resource = ...create...; try { ... } finally { resource.close(); resource = null; }
```

- All of Java's resources classes implement either this interface directly, or its sub-interface `java.io.Closeable` for I/O resources:

Listing: The abridged code of `java.io.Closeable`

```
package java.io;

import java.io.IOException;

/** A {@code Closeable} is a source or destination of data that can be closed. The close method is invoked to
 *  release resources that the object is holding (such as open files). ... */
public interface Closeable extends AutoCloseable {

    /** Closes this stream and releases any system resources associated with it. If the stream is already closed
     *  then invoking this method has no effect.
     *  <p> As noted in {@link AutoCloseable#close()}, cases where the close may fail require careful attention.
     *  It is strongly advised to relinquish the underlying resources and to internally <em>mark</em> the
     *  {@code Closeable} as closed, prior to throwing the {@code IOException}.
     *  @throws IOException if an I/O error occurs */
    public void close() throws IOException;
}
```

`java.util.Scanner`s are Resources

- So far, we have used instances of `java.util.Scanner` to read numbers from `stdin`

`java.util.Scanner`s are Resources

- So far, we have used instances of `java.util.Scanner` to read numbers from `stdin`

Listing: The code of our Vertical Ball Throw example

```
import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a x0m tall person
 * with velocity v0m/s. Where is it after t seconds? x(t) = x0 + v0 * t - 0.5 * g * t^2
 */
public class VerticalBallThrow {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    Scanner scanner = new Scanner(System.in); // initiate reading from System.in, ignore for now
    System.err.println("Enter size x0 of person in m:"); //$NON-NLS-1$
    double x0 = scanner.nextDouble();          // read initial vertical position x0
    System.err.println("Enter initial upward velocity v0 of ball in m/s:"); //$NON-NLS-1$
    double v0 = scanner.nextDouble();          // read initial velocity upwards v0
    double g = 9.80665d;                       // free fall acceleration downwards
    System.err.println("Enter time t in s:"); //$NON-NLS-1$
    double t = scanner.nextDouble();           // read the time $t$
    double xt = x0 + (v0*t) - 0.5d*g*t*t;      // x(t) = x0 + v0 * t - 0.5 * g * t^2
    System.out.println((xt > 0d) ? xt : 0d);   // prints result and makes sure the ball stops at ground
  }
}
```

`java.util.Scanner`s are Resources

- So far, we have used instances of `java.util.Scanner` to read numbers from `stdin`
- `java.util.Scanner`s are Resources implementing `java.io.Closeable`

`java.util.Scanner`s are Resources

- So far, we have used instances of `java.util.Scanner` to read numbers from `stdin`
- `java.util.Scanner`s are Resources implementing `java.io.Closeable`

Listing: The abridged code of our `java.util.Scanner`

```
package java.util;

import ...


/** A simple text scanner which can parse primitive types and strings using regular expressions. ... */
public final class Scanner implements Iterator<String>, Closeable {
    private boolean closed = false;
    private Readable source;
    private boolean sourceClosed = false;
  ...

  /** Closes this scanner. ... */
  public void close() {
    if (closed)
      return;
    if (source instanceof Closeable) {
      try {
        ((Closeable)source).close();
      } catch (IOException ioe) {
        lastException = ioe;
      }
    }
    sourceClosed = true;
    source = null;
    closed = true;
  }
}
```

`java.util.Scanner`s are Resources

- So far, we have used instances of `java.util.Scanner` to read numbers from `stdin`
- `java.util.Scanner`s are Resources implementing `java.io.Closeable`
- That's why we get warnings from the Eclipse compiler like "Resource leak: 'scanner' is never closed"

VerticalBallThrow using Scanner and try-with-resource

Listing: An example for a try-with-resource block

```
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a x0m tall person
 * with velocity v0m/s. Where is it after t seconds? x(t) = x0 + v0 * t - 0.5 * g * t^2
 * Applying try-with-resource to close the scanner
 */
public class VerticalBallThrowTryWithResource {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {//Scanner is resource, implements java.io.Closeable
      System.err.println("Enter size x0 of person in m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble();         // read initial vertical position x0
      System.err.println("Enter initial upward velocity v0 of ball in m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble();         // read initial velocity upwards v0
      double g = 9.80665d;                      // free fall acceleration downwards
      System.err.println("Enter time t in s:");//$NON-NLS-1$
      double t = scanner.nextDouble();          // read the time $t$
      double xt = x0 + (v0*t) - 0.5d*g*t*t;     // x(t) = x0 + v0 * t - 0.5 * g * t^2
      System.out.println((xt > 0d) ? xt : 0d); // prints result, makes sure the ball stops at ground
    } // scanner.close is automatically invoked when the code reaches this point
  }
}
```

Listing: An example for a try-with-resource block and catch

```
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a x0m tall person
 * with velocity v0m/s. Where is it after t seconds? x(t) = x0 + v0 * t - 0.5 * g * t^2
 * Applying try-with-resource to close the scanner
 */
public class VerticalBallThrowTryWithResourceAndCatch {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {// Scanner is resource, implements java.io.Closeable
      System.err.println("Enter size x0 of person in m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble();          // read initial vertical position x0
      System.err.println("Enter initial upward velocity v0 of ball in m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble();          // read initial velocity upwards v0
      double g = 9.80665d;                       // free fall acceleration downwards
      System.err.println("Enter time t in s:"); //$NON-NLS-1$
      double t = scanner.nextDouble();           // read the time $t$
      double xt = x0 + (v0*t) - 0.5d*g*t*t;      // x(t) = x0 + v0 * t - 0.5 * g * t^2
      System.out.println((xt > 0d) ? xt : 0d);   // prints result and makes sure the ball stops at ground
    } catch (InputMismatchException error) { // scanner.close is always invoked, even if catch is executed
      System.err.println("Sorry, you provided an incorrect input."); //$NON-NLS-1$
    }
  }
}
```

Summary

- We have learned about exceptions to deal with errors in programs
- Exceptions are special objects generated if something goes wrong, holding information about what and where the fault has happened
- We should check data as much as possible and `throw` exceptions as soon as possible to prevent greater mischief
- If an exception is thrown, the code afterwards is skipped and control passes to the next fitting `catch` statement
- We can `catch` exception objects to use the error information
- We can make sure that certain code is always executed by putting it into `finally`
- The try-with-resource statement is a special form of the `finally` statement to always close resources
- We can create our own exception classes
- We distinguish *checked* or *unchecked* exceptions

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog