

Object Oriented Programming 2

Topic 3 – Exceptions

Prof. Annett Laube

(adapted by Andres Scheidegger, FS 2018)

Motivation



Explosion of the Ariane 5 1996



<http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>

Goals

- ▶ You can explain and apply **Java's exception handling concept**
- ▶ You know how to program and when to trigger exceptions
- ▶ You can explain and apply the concept of exceptions that are checked/unchecked by the compiler.
- ▶ You know how to program blocks of code that might **throw an exception** and when to use such blocks
- ▶ You know how to program and where to place **exception handlers**
- ▶ You know how to program, where to place, and when to use imperative **clean up operations** that forcibly need to run before program termination

Outline

- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ Exception Handling
- ▶ Finally Clause
- ▶ Self-made Exception Types

Outline

- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ Exception Handling
- ▶ Finally Clause
- ▶ Self-made Exception Types

Review questions

1. What is an exception?
2. Why do I need exceptions?
3. Which exceptions should be handled?
4. What is the difference between checked and unchecked exceptions?
5. Why it is important to know the exception class hierarchy?

What is an Exception?

- ▶ "Exceptional event"
- ▶ **Definition:**

An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program.
- ▶ When an error occurs within a method, the method creates an **object** and hands it off to the runtime system.
- ▶ This **exception object** contains information about the error, including its type and the state of the program when the error occurred.
- ▶ Creating an exception object and handing it to the runtime system is called **throwing an exception**.

Outline

- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ Exception Handling
- ▶ Finally Clause
- ▶ Self-made Exception Types

Throw Exception Example

Method throws exception, e.g.

```
public class BankAccount {  
    ...  
    public void withdraw(double amount) {  
        if (amount > balance) {  
            throw new IllegalArgumentException(  
                "Amount exceeds balance");  
        }  
        balance = balance - amount;  
    }  
    ...  
}
```

Not executed,
when exception is
thrown



Throwing an Exception

- ▶ Example: argument has an illegal value

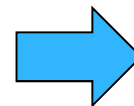
```
IllegalArgumentException exception =  
new IllegalArgumentException(  
    "Amount exceeds balance");  
throw exception;
```

- ▶ No need to store exception object in a variable:

```
throw new IllegalArgumentException(  
    "Amount exceeds balance");
```

- ▶ When an exception is thrown, the **program flow** in a method **terminates immediately**.

The program flow resumes in an **exception handler**.



Error Handling in Java

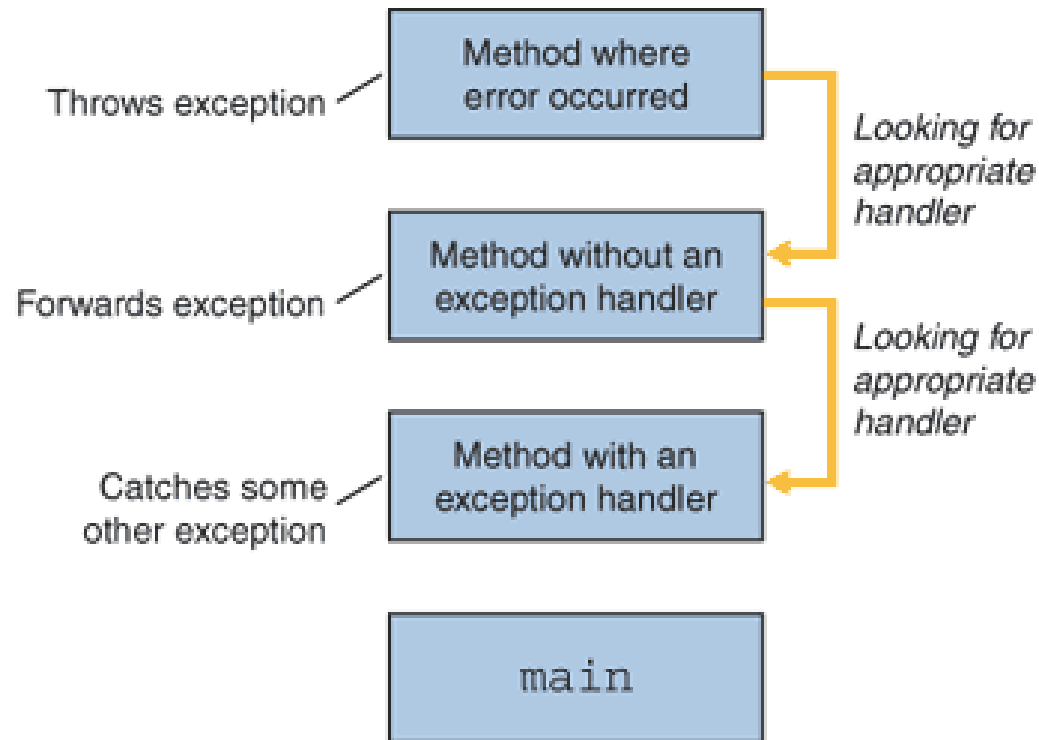


Fig.: Call Stack (upside down)

Source: java.sun.com

- Throw an exception if normal program flow is disrupted.
- Forward the exception upward the hierarchy in the call stack.
- Deal with the exception at a level where you can remedy the situation.
 - Start a dialog with the user, etc.

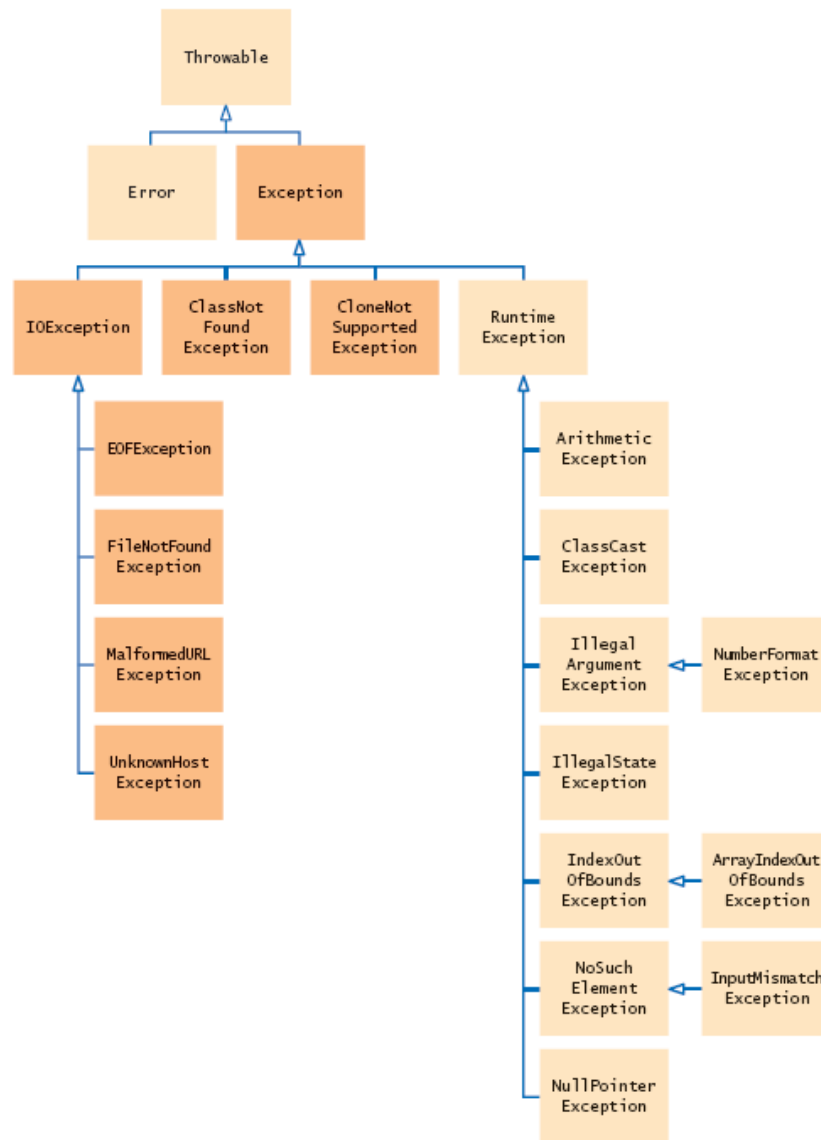
Two Kinds of Exceptions

- ▶ **Checked exceptions**
 - ▶ Events that a well-written application should **anticipate and recover** from.
 - ▶ E.g. user supplies the name of a nonexistent file, and the constructor throws **java.io.FileNotFoundException**

Two Kinds of Exceptions

- ▶ **Unchecked exceptions**
 - ▶ Runtime exceptions
 - ▶ Events that are **internal** to the application, and that the application usually **cannot anticipate or recover** from.
 - ▶ E.g., if a logic error causes a null to be passed to the constructor, the constructor throws a **NullPointerException**.
 - ▶ Errors
 - ▶ Events that are **external** to the application, and that the application usually **cannot anticipate or recover** from.
 - ▶ E.g., application is unable to read a file because of a hardware or system malfunction.

Hierarchy of Exception Classes



- **Checked exceptions** extend the classes
 - IOException
 - ClassNotFoundException
 - CloneNotFoundException
 - Majority occur when dealing with input and output
→ anticipate and recover

- **Unchecked exceptions** extend the classes
 - RuntimeException
 - Programming errors → avoid by careful programming!
 - Error (system internal)

Figure 1 The Hierarchy of Exception Classes

Outline

- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ **Exception Handling**
- ▶ Finally Clause
- ▶ Self-made Exception Types

Exception Handling and I/O

- ▶ When doing IO, there are many “opportunities” to get confronted with exceptions
- ▶ Example: Deal with checked exceptions when opening a file (the file may not exist)

```
String filename = ...;  
FileReader reader = new FileReader(filename);
```

- ▶ Deal with the fact that the **FileReader** constructor can throw a **FileNotFoundException**.
See <https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html>
- ▶ The application is expected to anticipate and recover from this situation!

Exception Handling and I/O

- ▶ Example: Deal with unchecked exceptions when reading data from a **Scanner** object
 - ▶ `public int nextInt()` throws unchecked **InputMismatchException**
See <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>
- ▶ However: If you get unchecked exceptions, this may be due to careless programming!
 - ▶ Example: **Scanner** provides methods to check, whether data of the expected type is ready to be read:
`public boolean hasNextInt()`
- ▶ Careful programming: Check the conditions first (if possible)!

Dealing with Exceptions 1/2

Two choices:

1. Handle the exception locally

- ▶ Use try/catch blocks

```
try {  
    FileReader reader = new FileReader(filename);  
    ...  
} catch (FileNotFoundException e) {  
    // handle FileNotFoundException ...  
}
```

Dealing with Exceptions 2/2

2. Handle the exception **further up** the in call stack hierarchy

- ▶ Use **throws** specifier in the method declaration

```
public void read(String filename) throws  
FileNotFoundException {  
    FileReader reader = new FileReader(filename);  
    ...  
}
```

- ▶ The method could throw **multiple** types of exceptions

```
public void read(String filename) throws IOException,  
ClassNotFoundException
```

Rule of thumb: Handle exceptions as local as possible! Throw exceptions further up, if you need more information about the context to handle them.

Programming with Exceptions

- ▶ Execute statements that may cause an exception in the try block
- ▶ Provide catch clause for each type of exception and handle the exception

- ▶ Example:

```
try {
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    String input = in.next();
    int value = Integer.parseInt(input);
    ...
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (NumberFormatException e) {
    System.out.println("Input was not a number");
}
```

- ▶ **NoSuchElementException** (next(), unchecked!) remains thrown until caught by a surrounding try block.

Catching Multiple Exceptions

- ▶ Multiple catch blocks with duplicated handlers can be combined

- ▶ Replace ...

```
catch(FileNotFoundException e) {  
    e.printStackTrace();  
} catch (NumberFormatException e) {  
    e.printStackTrace();  
}
```

- ▶ ... by ...

```
catch(FileNotFoundException | NumberFormatException e {  
    e.printStackTrace();  
}
```

- ▶ ... and prefer to

```
catch(Exception e {  
    e.printStackTrace();  
}
```

Programming with Exceptions

- ▶ Execute statements in **try block**.
- ▶ If no exceptions occur, catch clauses are skipped.
- ▶ If an **exception** of a matching type occurs, execution jumps to catch clause.
- ▶ If an exception of another type occurs, this exception remains thrown until it is caught by a surrounding try block (possibly further up in the call stack).
- ▶ **catch (IOException exception) block**
 - ▶ **exception** contains reference to the exception object that was thrown.
 - ▶ catch clause can analyze object to find out more details.
 - ▶ **exception.printStackTrace()** :
printout of chain of method calls that lead to exception.

Advantages of Try/Catch

- ▶ Separating error-handling code from “regular” code.
- ▶ Propagating errors up the call Stack to have more context
- ▶ Grouping and differentiating error types.
- ▶ For details see
<http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html>

Outline


- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ Exception Handling
- ▶ **Finally Clause**
- ▶ Self-made Exception Types

Finally Clause: Motivation

- ▶ Once a try block is entered, statements in a finally clause are **guaranteed** to be executed, **no matter if an exception is thrown or not**.

- ▶ Example:

```
reader = new FileReader(filename);  
Scanner in = new Scanner(reader);  
readData(in);  
reader.close();
```



May not get up to
here

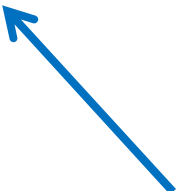
- ▶ **reader.close()** should be executed no matter if exception is thrown or not.
-> use finally clause for this purpose.

Finally Clause: Example

```
FileReader reader = new FileReader(filename);

try {
    Scanner in = new Scanner(reader);
    readData(in);
} finally {

    reader.close();
}
```



if an exception occurs, finally clause is also executed **before** exception is passed to its handler

Finally Clause

- ▶ The finally clause is executed when try block is exited in any of three ways:
 - ▶ After last statement of try block.
 - ▶ After last statement of catch clause, if this try block caught an exception.
 - ▶ When an exception was thrown in try block and not caught.
- ▶ **Recommendation:**
 - ▶ keep your code readable
 - ▶ don't mix catch and finally clauses in same try block.
 - ▶ Use nested try blocks.

Exercise

Which output produces the following program?

```
public static void main(String [] args)
{
    try
    {
        badMethod();
        System.out.print("A");
    }
    catch (RuntimeException ex)
    {
        System.out.print("B");
    }
    catch (Exception ex1)
    {
        System.out.print("C");
    }
    finally
    {
        System.out.print("D");
    }
    System.out.print("E");
}

public static void badMethod()
{
    throw new RuntimeException();
}
```

try-with-resources statement

- ▶ Available in Java 7
- ▶ The `try-with-resources statement` is a try statement that declares one or more resources.
- ▶ A `resource` is an object that must be closed after the program is finished with it.
- ▶ The try-with-resources statement ensures that each resource is closed at the end of the statement.
- ▶ Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.
- ▶ Replaces the `finally` statement that calls the `close()` method

try-with-resources statement

► Example PrintWriter ...

```
try {
    PrintWriter out =
        new PrintWriter(
            new FileWriter(
                "OutFile.txt"));
    for (int i=0; i<10; i++)
        out.println(i);
}
finally {
    out.close();
}
```

...

```
try (
    PrintWriter out =
        new PrintWriter(
            new FileWriter(
                "OutFile.txt")) {
    for (int i=0; i<10; i++)
        out.println(i);

    // close ← Not needed
               anymore, but
               still can throw
               exceptions
}
```

► To be preferred!

try-with-resources statement

Example FileInputStream ...

```
try {
    InputStream in =
        new FileInputStream(
            "OutFile.txt");

    try {
        ...
        int next = in.read();
        ...
    } finally {
        in.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

```
...
try (
    InputStream in =
        new FileInputStream(
            "OutFile.txt")) {

    // no nested try
    ...
    int next = in.read();
    ...

    // no finally
    // no close()

} catch (IOException e) {
    e.printStackTrace();
}
```

Outline

- ▶ What are exceptions?
- ▶ Throwing exceptions
- ▶ Exception Handling
- ▶ Finally Clause
- ▶ Self-made Exception Types

Self-made Exception Types

- ▶ You can design your own exception classes as subclasses of
 - ▶ Exception (checked) or
 - ▶ RuntimeException (unchecked)
- ▶ However, try to re-use as much as possible existing exception classes (especially runtime exceptions)
- ▶ Use your own exception classes when throwing exceptions across abstraction boundaries (e.g. the user of the persistence layer might not be interested in a particular SQL-Exception)

Self-made Exception Types

- ▶ Example:

```
if (amount > balance) {  
    throw new InsufficientFundsException("withdrawal of "  
        + amount + " exceeds balance of " + balance);  
}
```

- ▶ Make this exception an unchecked exception
 - ▶ Programmer could have avoided it by calling *getBalance()* first.
- ▶ Extend **RuntimeException** or one of its subclasses.
- ▶ Supply two constructors
 - ▶ Default constructor.
 - ▶ A constructor that accepts a message string describing reason for exception.

Self-made Exception Types

- ▶ Code for defining this exception

```
public class InsufficientFundsException extends
    RuntimeException {

    public InsufficientFundsException() {}

    public InsufficientFundsException(String message) {
        super(message) ;
    }
}
```

Self-made Exception Types

- ▶ It is good practice to add further fields, in order to provide as much information about the exceptional state as possible

```
public class InsufficientFundsException extends
    RuntimeException {
    private double availableBalance;
    private double amountToWithdraw;

    public InsufficientFundsException(
        double balance, double amount) {
        availableBalance = balance;
        amountToWithdraw = amount;
    }

    public double getAmountToWithdraw() {...}
    ...
}
```

Chained Exceptions

- ▶ Used to transport lower level exceptions across abstraction levels
- ▶ Example: Persistence layer does throw **PersistenceException** instead of **SQLException** to layer above

```
try {  
    saveRecord(record);  
} catch (SQLException sqlException) {  
    throw new PersistenceException(sqlException);  
}
```

- ▶ The original **SQLException** is **wrapped** by the **PersistenceException**

Summary 1/2

▶ **Java's exception handling concept**

- ▶ Throw an exception if normal program flow is disrupted.
- ▶ Forward the exception upward the hierarchy in the call stack.
- ▶ Catch the exception at a level where you can handle the situation.
 - ▶ Start a dialog with the user, etc.

▶ Checked vs. unchecked exceptions

- ▶ **Checked exceptions** are exceptions that a well-written application should anticipate and recover from. Checked exceptions either need to be caught or need to be explicitly listed in method declarations to enable forwarding up the call stack.
- ▶ **Unchecked exceptions** are errors (external) and run time exceptions (internal). Unchecked exceptions need not to be explicitly listed in method declarations, but should be described in the documentation!

Summary

- ▶ **finally clause**

- ▶ used perform clean up operation before program termination, no matter whether this termination is regular or by exception.
- ▶ In Java 7 use [try-with-resources statement](#)