326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Exceptions, Constants, Enumerations

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Donald Ervin Knuth

Concludes a memo (addressed to Peter van Emde Boas) by the words:
Beware of bugs in the above code; I have only proved it correct, not tried it

Figure: Knuth at reception for the Open Content Alliance, 2005 by J. Appelbaum

What is an Exception

ex·cep·tion

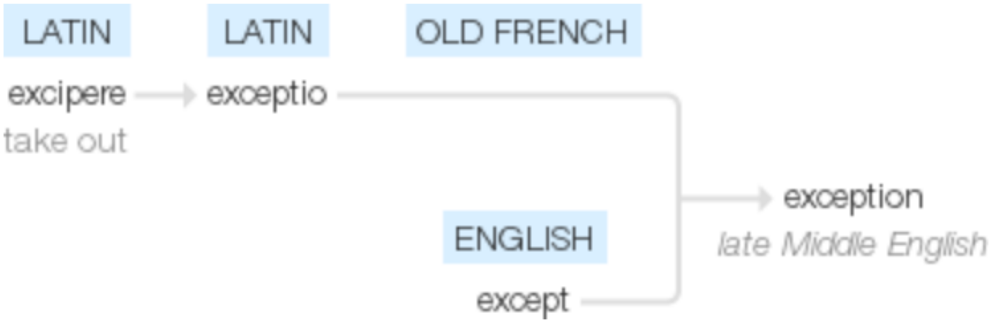
/ikˈsepSH(ə)n/

noun

noun: **exception**; plural noun: **exceptions**

a person or thing that is excluded from a general statement or does not follow a rule.
"the drives between towns are a delight, and the journey to Graz is no exception"
synonyms: anomaly, irregularity, deviation, special case, isolated example, peculiarity, abnormality, oddity; More

Origin



late Middle English: via Old French from Latin *exceptio(n-)*, from *excipere* 'take out' (see except).

Figure: Screenshot from Google

Example of an Exception

a person or thing that is excluded from a general statement or does not follow a rule.

```
1 public static double computeSomething(...) {
2     ...
3     z = x / y;
4     ...
5 }
```

Computes something except for z being zero.

- What if y is zero?
- What if x and y are zero?
- $y = 0$ is an exception.

Exceptions in Java

- Objects which inherit from **java.lang.Throwable**.
- Exceptions are thrown by the keyword **throw**.
- Two types of Exceptions.
  - **Checked** exceptions: **Need** to be declared in a method or constructor's **throws** clause.

```
1 public FileImageInputStream(File f)
2     throws FileNotFoundException, IOException {
3     ...
4 }
```

Constructor of FileImageInputStream declares two Exceptions

  - **Unchecked** exceptions: Do **not** need to be declared in a throws clause.
    - **Error:** Abnormal conditions that should not occur.
    - **RuntimeException:** Thrown during the normal operation of the JVM.

API Example: Throwing an Exception

The method divide of BigDecimal throws an ArithmeticException:

```
1  public BigDecimal divide(BigDecimal divisor) {
2    if (divisor.signum() == 0) {    // x/0
3      if (this.signum() == 0)       // 0/0
4        throw new ArithmeticException("Division undefined");
5      throw new ArithmeticException("Division by zero");
6    }
7    ...
```

BigDecimal.divide throws an ArithmeticException.

- What if $divisor$ is zero?
- What if $this$ and $divisor$ are zero?
- $divisor = 0$ is an exception $\Longrightarrow$ **throw new** Arithmetic**Exception**(...)

```
1  public class ArithmeticException extends RuntimeException {
2      ...
3  }
```

ArithmeticException is of type RuntimeException.

Different Types of Exceptions in Java

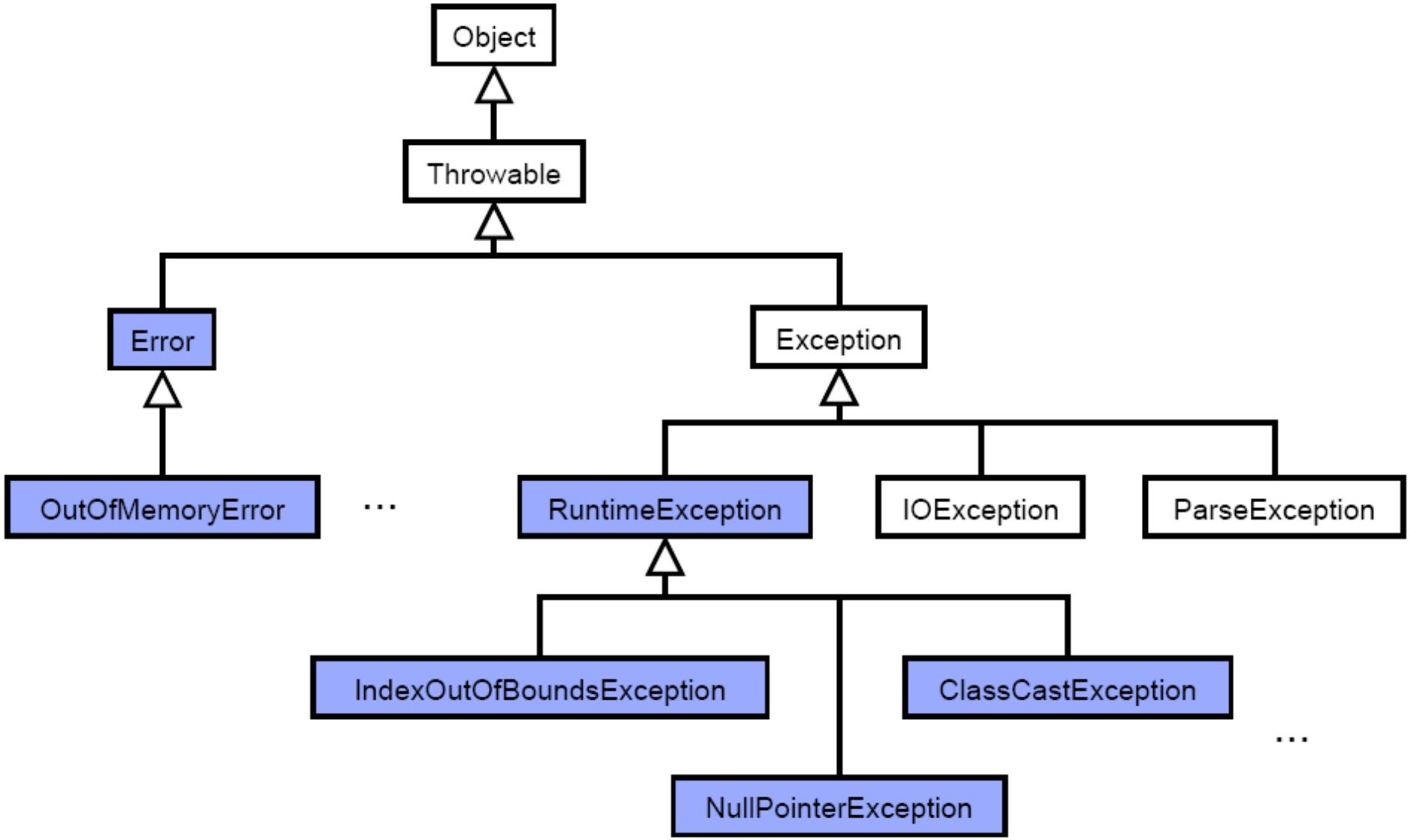


Figure: Unchecked exceptions are of type Error or RuntimeException.

Exceptions Can Be Caught

- **Exceptions propagate** up the entire caller hierarchy and the program terminates (interrupted termination), **unless they are caught.**
- To catch an exception, you must put the code in a **try** block.
- Below the block, you can **catch** exceptions of different types.
- The first catch block which "fits" the type of the thrown exception will be executed.

```
1  try {
2      // Code that might throw an Exception
3      ...
4  } catch (FileNotFoundException e) {
5      // FileNotFoundException e can be treated
6      ...
7  } catch (Exception e) {
8      // Catch more general exceptions below
9      ...
10 }
```

Catching all exceptions.

- The semantic is like: **if** (e **instanceof** Exeption1) {...} **else if** (e **instanceof** Exeption2) {...} **else if**...

Executing Code in Any Case

- A **finally** block can be declared, which will be executed under any circumstances, unless the thread will be terminated from outside.

```
1  try {
2      // Code that might throw an Exception
3      ...
4  } catch (FileNotFoundException e) {
5      // FileNotFoundException e can be treated
6      ...
7  } finally{
8      // Execute some cleanup code. E.g. closing a file
9      ...
10 }
```

Catch a FileNotFoundException and all sub-exceptions.

- It is possible to declare try {...} finally {...} without a catch block.
- The finally block is a key tool for preventing resource leaks.

Try with resources

- In Java 7 and later, you can specify resources which need to be closed.

```
1 try (Reader r = new FileReader(path)) {
2     // Do something with the file
3     ...
4 }
```

Try with resources which will be closed automatically.

- Such resources must implement the **interface AutoCloseable**.
- More resources are separated by the semicolon.

Catching More than one Exception

- In Java 7 and later, a single catch block can handle more than one type of exception.

```
1 try {
2     // Code that might throw an Exception
3     ...
4 } catch (IOException | InstantiationException e) {
5     // Exceptions of the above types can be treated
6     ...
7 }
```

Catch all exceptions of (sub)type IOException and InstantiationException.

Advantages: Exceptions vs. Error Codes

- Separating error-handling code from "regular" code.
- Propagating errors up the call stack.
- Grouping and differentiating error types.

Bad examples

- What is wrong with the below code?

```
1 try {
2     ...
3 } catch (Exception e) {
4     ...
5 } catch (ArithmeticException e) {
6     ...
7 }
```

- Errors indicate severe failures concerning the runtime environment.
- There is no need for a decent program to catch errors.

```
1 try {
2     ...
3 } catch (Throwable t) {
4     ...
5 }
```

Exceptions – Summary

- Use exceptions to indicate that an abnormality occurred.
- Use the try block to identify a block of code in which an exception can occur.
- Use the catch block to handle a particular type of exception.
- Use the finally block to close files, recover resources, and perform other clean up.

Constants in Java

- Constants are public static final fields.
- Use UPPERCASE letters for constants and separate words with _.

```
1  public class CalendarApp extends ... {
2      public static final int SUNDAY = 0;
3      public static final int MONDAY = 1;
4      public static final int TUESDAY = 2;
5      public static final int WEDNESDAY = 3;
6      public static final int THURSDAY = 4;
7      public static final int FRIDAY = 5;
8      public static final int SATURDAY = 6;
9      ...
10 }
```

Constants are public static final fields.

Declaring an Enumeration I

- Enumerations are collections of constants.
- Use the keyword **enum** to declare an enumeration.

```
1 public enum Day {
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
3     THURSDAY, FRIDAY, SATURDAY
4 }
```

Enumerations of weekdays.

- An enumeration defines a type.

```
1 public class EnumTest {
2     public Day day;
3 }
```

A class containing a field of type Day.

- You can access the values by EnumName.CONSTANT.

```
1 public static void main(String[] args) {
2     EnumTest e = new EnumTest();
3     e.day = Day.TUESDAY;
4 }
```

Accessing enum values.

Declaring an Enumeration II

- Put the enum type into a .java file with the same name, or
- it is also very common to define simple enum types inside of classes.

```
1 public class EnumTest {
2     public enum Day {
3         SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
4         THURSDAY, FRIDAY, SATURDAY
5     }
6     public Day day;
7 }
```

Enumerations of weekdays.

- You can access the values by ClassName.EnumName.CONSTANT.

```
1 public static void main(String[] args) {
2     EnumTest e = new EnumTest();
3     e.day = EnumTest.Day.TUESDAY;
4 }
```

Accessing enum values.

Working with Enumeration I

- You can use enum values in a switch-case statement.

```
1 String dayName;
2 switch (day) {
3     case SUNDAY: dayName = "Sunday"; break;
4     case MONDAY: dayName = "Mondays"; break;
5     ...
6 }
```

- You can access the name, ordinal number, hash value,...

```
1 public class EnumTest {
2     public Day day;
3     public String getDayName() {
4         return day.name();
5     }
6 }
```

Working with Enumeration II

- You can define complex enum types.
- Arbitrary many properties and constants. (Methods are also allowed.)

```
1  public enum Day {
2      SUNDAY("Sunday", false),
3      MONDAY("Monday", true),
4      ...
5      SATURDAY("Saturday", false);
6
7      // Properties of a Day.
8      public final String dayName;
9      public final boolean workday;
10     // Constant defining the first Day of a week.
11     public static final Day FIRST_DAY = SUNDAY;
12
13     Day(String dayName, boolean workday) {
14         this.dayName = dayName;
15         this.workday = workday;
16     }
17 }
```

- And access the properties.

```
1 Day d = Day.FIRST_DAY;
2 System.out.println(d.dayName);
3 System.out.println(d.workday);
```