Programmieren II

Exceptions

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from T. Bögel)

May 22, 2014

Outline

Outline

Outline

Format strings I

Format strings

- Format specifiers begin with a %
- End with a 1- or 2-character *conversion*

Examples for conversions

- d formats integer value as a decimal value
- f formats floating point values
- n formats platform-specific new line
- s formats any value as a string

Format strings II

Additional elements

- Precision (e.g. for floats)
- Width (minimum width)
- Flags (special formatting options)
- Argument index

Format strings III

Example

```
double amount = 34002005.2450;
System.out.format("Money gained/lost since last
                   statement: %,.2f", amount);
```

Output:

```
Money gained/lost since last statement: 34,002,005.25
```

printf and format are synonyms

Outline

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );
```

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );
```

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );
```

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );
```

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );

boolean result = m.matches(); // result = ?
```

Methods of the class Matcher

- 3 Methods:

`boolean`	`matches( )`	complete match
`boolean`	`lookingAt( )`	match with prefix
`boolean`	`find( )`	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );

result = m.lookingAt(); // result = ?
```

Methods of the class Matcher

- 3 Methods:

boolean	matches( )	complete match
boolean	lookingAt( )	match with prefix
boolean	find( )	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );

result = m.find(); // result = ?
```

Methods of the class Matcher

- 3 Methods:

boolean	matches( )	complete match
boolean	lookingAt( )	match with prefix
boolean	find( )	match with sub-sequence (iterative)

```
Pattern p = Pattern.compile( "ab" );
Matcher m = p.matcher( "abcdabcd" );

result = m.find(); // result = ?
result = m.find(); // result = ?
```

Outline

Reading a file line-by-line (HFJ, pp. 452 – 454)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class PlaintextReader {
  public static void main(String[] args) {
    try {
      BufferedReader br = new BufferedReader(new FileReader("
          test.txt"));
      String line = br.readLine();
      while (line != null) {
          line = br.readLine();
          System.out.println(line);
      }
      // always ensure that a stream is closed!
      br.close();
    } catch (IOException e) {e.printStackTrace();}
  }
}
```

code/PlaintextReader.java

Writing to a file

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class PlaintextWriter {
  public static void main(String[] args) {
    try {
      BufferedWriter bw = new BufferedWriter(new FileWriter("
          test.txt"));
      bw.write("bla, bla, bla\n");
      bw.close();
    } catch (IOException e) {
      e.printStackTrace();
    }

  }
}
```

code/PlaintextWriter.java

Outline

Outline

NIO.2

New Input/Output Classes introduced in Java 7

- Java NIO.2 introduced in Java 7
- Comprehensive support for file I/O and file system interaction
- Tutorial:
  `http://docs.oracle.com/javase/tutorial/essential/io/`

What is a Path?

- Hierarchical structure starting at the root node (/ or `C:\`, `D:\` etc.)
- **Absolute** path: `/home/root/statusReport.txt`
- **Relative** path: `joe/foo`

The Path class

java.nio.file.Path

- Primary entry point for file I/O
- Programmatic representation of a path in the file system
- Path reflects underlying OS

Obtaining a Path object

- Helper class: Paths
- `Path p1 = Paths.get("/tmp/foo");`
- `Path p2 = Paths.get("/home/root","Documents","world_formula.txt");`
- `Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");`

A selection of `Path` methods

`Path` methods

- `toString`: returns a string representation of the `Path`
- `getFileName`: returns the last element of the sequence of name elements
- `getParent`: returns the path of the parent directory

Joining two paths

- Paths can be joined with the `resolve` method
- Partial path given as a parameter is appended to original path

Example

```
Path p1 = Paths.get("/home/joe/foo");
// Result is /home/joe/foo/bar
System.out.format("%s%n", p1.resolve("bar"));
```

Additional methods

Check additional methods in the official Java API:
`http://docs.oracle.com/javase/7/docs/api/`

Outline

Files methods

- Class `Files` provides many methods
- Read the Java API to get an overview: `http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html`

Checking a file or directory

Verifying existence of file/directory

- Static method: `public static boolean exists(Path path,LinkOption... options)`
- Equivalently: `. . . notExists . . .`

Checking File Accessibility

- Verifying that a file is accessible:
  - `isReadable(Path)`
  - `isWritable(Path)`
  - `isExecutable(Path)`

Deleting a file or directory

`Files` methods

- Two methods to delete files, directories and links:
  1. `delete(Path)`: deletes the file or throws an exception, if the deletion fails
  2. `deleteIfExists(Path)`: deletes the file. No exception, if the file does not exist

Commonly used methods for small files I

Reading all bytes or lines at once

- Multiple small files
- Read all lines: `Files.readAllLines(Path path, Charset cs) throws IOException`
- Read all bytes: `Files.readAllBytes(Path path)`

Writing all bytes or lines to a file

- `File.write(Path, byte[], OpenOption...)`
- `Files.write(Path, Iterable<extends CharSequence>, Charset, OpenOption...)`

Commonly used methods for small files II

Example

```
Path file = Paths.get("test.txt");
byte[] buf = ...;
Files.write(file, buf);
```

Buffered I/O I

Conveniently reading a file

- Convenience method:
  `Files.newBufferedReader(Path, Charset)`
- Opens a file for reading
- Returns a `BufferedReader`
- Similar for `BufferedWriter`:
  `Files.newBufferedWriter(Path, Charset, OpenOption...)`

Buffered I/O II

Example

```
Charset charset = Charset.defaultCharset();
try (BufferedReader reader = Files.newBufferedReader(file,
    charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Creating files I

Creating files

- Method: `Files.createFile(Path, FileAttribute<?>)`
- Creates a file with an initial set of attributes
- If no attributes are specified: default attributes

Creating files II

Example

```
Path file = ...;
try {
    // Create the empty file with default permissions, etc.
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("file named %s" +
        " already exists%n", file);
} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

Creating temporary files

Temporary files

- Platform-specific creation of a temp file
- Two methods:
  - `createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`
  - `createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`

Example: creating a temp file

```
try {
    Path tempFile = Files.createTempFile(null, ".tmp");
    System.out.format("The temporary file" +
        " has been created: %s%n", tempFile);
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

That's just the beginning...

Other NIO.2 methods

- Java NIO.2 provides many methods for commonly used file operations
- You've seen just a few of them
- Other things you might be interested in:
  - Walking the file tree

    `http://docs.oracle.com/javase/tutorial/essential/io/walk.html`
  - Finding files

    `http://docs.oracle.com/javase/tutorial/essential/io/find.html`
  - Watching directory for changes

    `http://docs.oracle.com/javase/tutorial/essential/io/notification.html`
  - ...
- Read the NIO.2 documentation

Outline

Summary

- `java.io`: classes for reading and writing files
- Sequential access streams for bytes and strings

`java.nio.file`

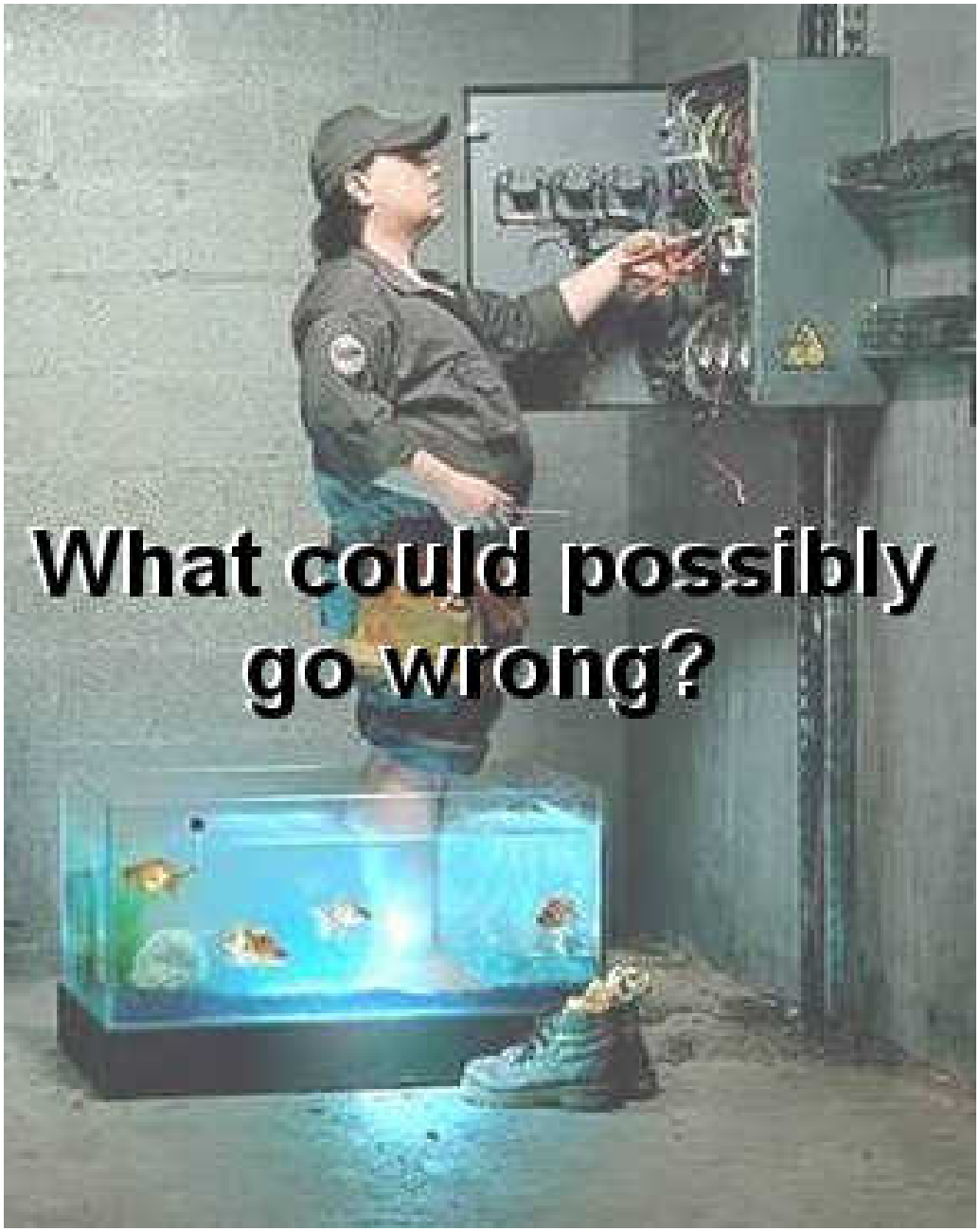
- Extensive support for file system I/O
- Comprehensive API as a starting point
  - `Path` class: manipulating a path
  - `Files` class: file operations, such as moving, copy, deleting, and also methods for retrieving and setting file attributes
- More information on NIO.2:
  `http://openjdk.java.net/projects/nio/`

Outline

What you should know about exceptions

- Why do we need exceptions?
- What's the difference between checked and unchecked exceptions?
- What different possibilities do you have to handle checked exceptions?
- How do you throw your own exception?

Let's talk about errors and mistakes...



source: http://software.intel.com/sites/default/files/race.jpg

Outline

Exceptions

What is an *exception*?

- Shorthand for *exceptional event*
- Definition:

  Event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions

What happens at an exception?

1. Error occurs within a method
2. Method creates an *exception object*
3. $\rightarrow$ **throwing an exception**

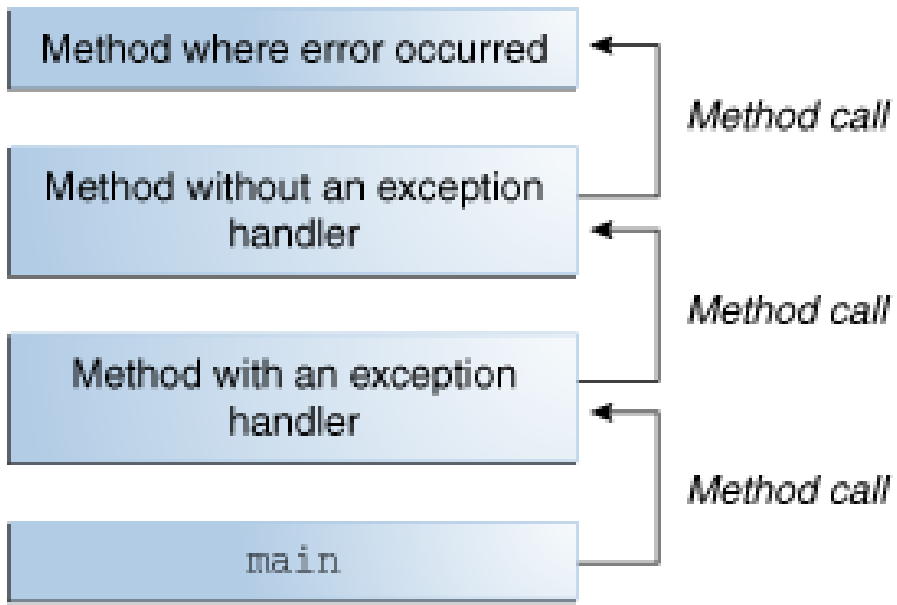
Exception Object

- Information about the error
- Type of the error
- State of the application at error time

Exception handling

Call stack

- Thrown exceptions need to be handled somehow
- Runtime system searches ordered list of methods (call stack) for exception handling code

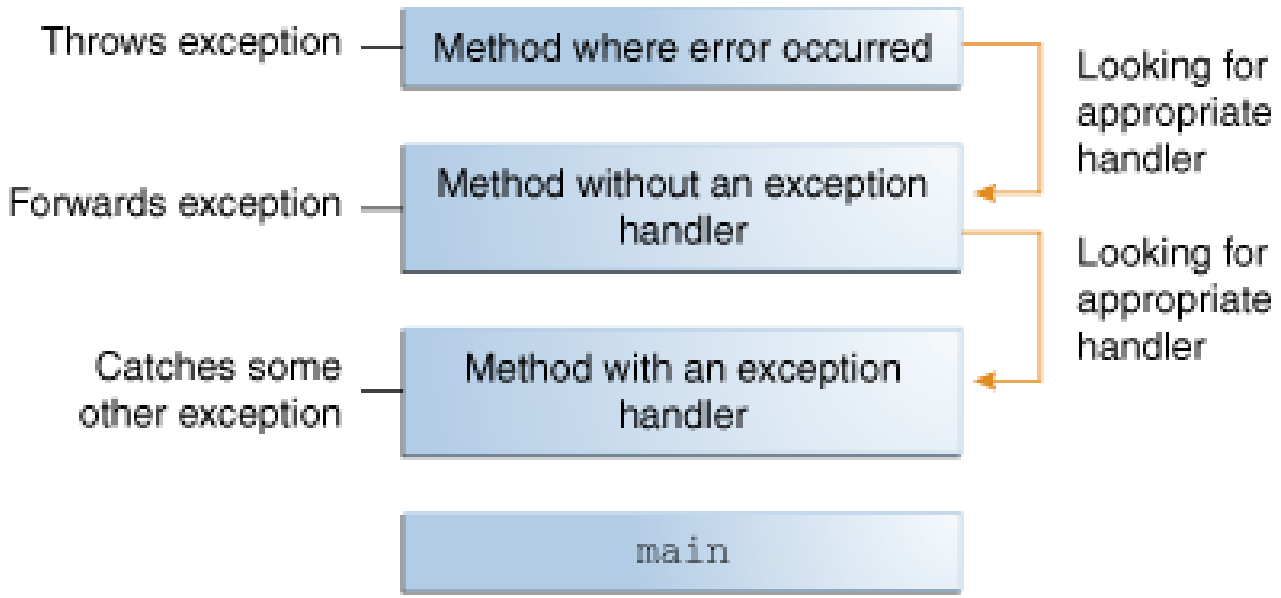


Call stack. Source:

`http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`

Throwing, forwarding and catching exceptions

- Proceed search through call stack in reversed order (**forward**)
- If appropriate handler is found: pass exception object to it
- Appropriate: correct type
- Exception handler **catches the exception**
- Without any appropriate exception handler: program terminates :(



Searching the call stack for the exception handler. Source:

`http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`

Outline

Catch or Specify Requirement I

1) Checked exceptions

- Exceptions that should be handled
- e.g. `java.io.FileNotFoundException`
- *Catch or specify requirement*

2) Errors

- Usually external conditions a program cannot recover from
- Example: `java.io.IOError`

3) Runtime exceptions

- Usually programming bugs
- Meet your friend, the `NullPointerException`
- **Unchecked exceptions**

Catch or Specify Requirement II

Checked exceptions need to be handled

1. Method has a `try` statement catching a specific exception
2. Method "announces" that it can throw an exception (passing it up the call stack)

- Otherwise: code won't compile
- Method can throw an exception via the `throws` clause

Outline

Dealing with exceptions

- Use `try`, `catch`, `finally` blocks to write an exception handler
- New in Java 7: `try`-with-resources

Example

- `ChatReader.java` (next slide)
- `readline`: call to a constructor to instantiate `FileReader`
- If file cannot be opened: constructor throws `FileNotFoundException`
- Class won't compile
- `IOException` is a *checked exception*
- `ArrayIndexOutOfBoundsException`: *unchecked exception*

ChatReader: checked vs. unchecked exceptions

```
import java.io.BufferedReader;

public class ChatReader {
    ...

  public String[] readlines() {
    /* This won't compile! FileReader throws a checked
        exception! */
    BufferedReader br = new BufferedReader(new FileReader(this.
        fileName));
        String[] lines = new String[1];
        /* Throws an unchecked exception (does not need to be
            handled) */
        lines[10] = "test";
  }


}
```

code/ChatReader.java

The try block

- Enclose code that might throw an exception within a try block

```
try {
    code
 }
   catch and finally blocks . . .
```

- Each line that might throw an exception: own line
- Or: single try block with multiple handlers for exceptions
- For each try block, you need to specify a catch block

The catch blocks I

- Goal: associate exception handlers with a `try` block
- Solution: define one (or more) `catch` blocks directly after the `try` block
- No code between the end of `try` and the beginning of `catch`

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

The catch blocks II

catch blocks

- Each catch block: exception handler that handles indicated type of exception
- ExceptionType is a name of a Throwable class
- Within the handler code, exception can be referred to by name variable
- Code in catch block is executed when exception handler is invoked
- Exception handler is triggered if handler is first one in call stack with correct ExceptionType

The catch blocks III

Catching more than one exception (Java $\geq$ 7)

- Java 7 and newer: one catch block for multiple exceptions
- Exception types are separated by |
- Example:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

The `finally` block

Cleaning up the mess...

- A `finally` block executes whenever a `try` block exists
- `finally` is *always* executed
- Useful for cleanup (even if no exceptions are expected)
- Use cases: closing connections, closing files (!) etc.
- Things in `finally` should not be done in the `catch` block
- Example:

```
finally {
  ...
}
```

Try-with-resources I

Files should **always** be closed

- `try` statement with one or more resources
- Resource: object that must be closed after usage
- Try-with-resources ensures that resources are closed

Try-with-resources II

Example

```
static String readFirstLineFromFile(String path) throws
    IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

- BufferedReader is the resource to be closed
- No matter what happens: br will be closed

The Exception class

Commonly used method

- Print the stack trace: `public void printStackTrace()`

Outline

Delegating exceptions I

Two different approaches

- Up to now: catching and handling exceptions within a method
- Alternative: delegate exceptions to handlers further up the call stack

Checked exceptions need to be taken care of

- Instead of catching: specifying that a method throws an exception
- Somebody calling the method needs to take care of the exception

Delegating exceptions II

Syntax

- throws statement in the method signature
- Specifies what kind of exception is thrown
- Note: **only checked exceptions** need to be specified (or caught)
- Example:

```
public String[] readlines() throws IOException {
        throw new IOException();
}
```

Outline

Let's throw exceptions

About the origin of . . . exceptions

- We know how to handle exceptions
- But: what's the actual origin of an exception?
- Each exception is thrown somewhere
- Exceptions are thrown with `throw`
- Each exception is a sub-class of `Throwable`
- Creating custom exceptions: sub-class `Throwable`

The throw statement

Throwing exceptions

- `throw` statement throws exceptions
- Single argument: throwable object
- Example:

```
throw someThrowableObject;
```

Example: stack I

- Example: pop method of a common stack object
- pop removes the top element from the stack and returns it

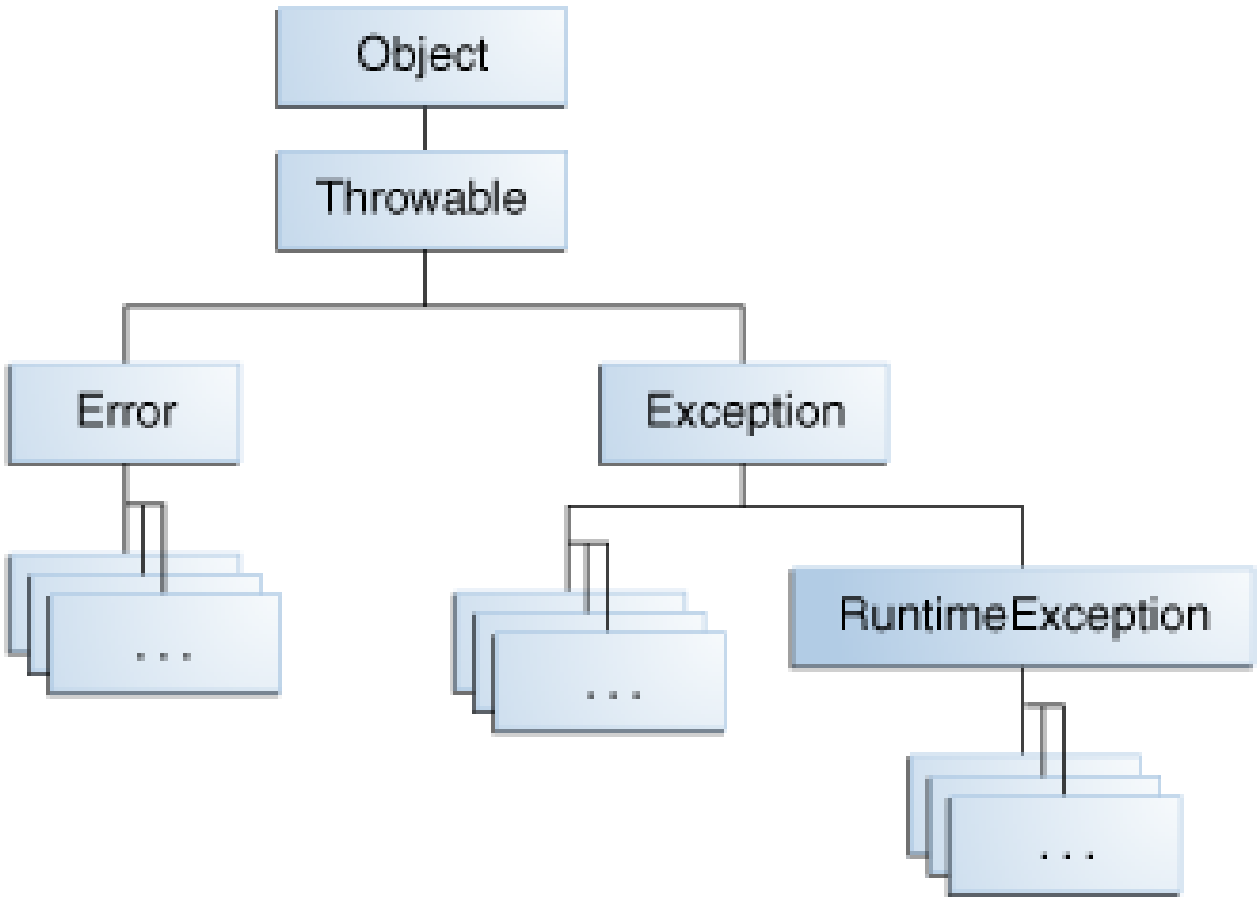
```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Example: stack II

pop throws an exception

- If the stack is empty: throw an `EmptyStackException` object
- Why does pop neither handle nor delegate the exception?
- `EmptyStackException` is not a checked exception!

Exception hierarchy I



`Throwable` class hierarchy.

source: `http://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html`

Exception hierarchy II

`Error` class

- Hard failure in the Java virtual machine
- E.g. dynamic linking failure
- Usually not recoverable

`Exception` class

- Most thrown and caught exceptions descendants of `Exception`
- Look at some exceptions: `http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html`
- Special case: `RuntimeException`
  - Indicate incorrect use of API
  - (Frequent) example: `NullPointerException`: accessing members of object through `null` reference
  - You normally should not throw `RuntimeExceptions`
  - Read through the list of exceptions!

Chained Exceptions I

- Responding to an exception by throwing another one
- First exception *causes* second exception
- This can be done with *ChainedExceptions*

Useful methods for chained exceptions

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

Chained Exceptions II

Example

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

- When `IOException` occurs: new `SampleException` is thrown

Accessing stack trace information I

Stack trace

- Execution history of the current thread
- Lists names of classes and methods that were called when an exception occurred

Accessing stack trace information II

Accessing and formatting the stack trace

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

Outline

Writing `Exception` classes

When to write your own exception class

- Exception type not represented by Java platform?
- Does custom exception type help to differentiate exceptions from other classes?
- Does your code throw more than one related exception?

Alternative: `Exception(String m)`

- Exception constructor: `Exception(String message)`
- Provides additional information about the error
- Printed with the stack trace
- Example: `throw new Exception("Username invalid");`

Unchecked Exceptions

Programmers are lazy...

- Checked exceptions need to be taken care of
- Why not just use unchecked exceptions? (`RuntimeException`, `Error`)

Convenience vs. reliability

- Exceptions are part of a method's public interface
- Programmers using your classes: knowledge about what could go wrong
- Runtime exceptions: programming problems
- Runtime exceptions occur frequently
- Do not throw a `RuntimeException` or a sub-class thereof

Outline

Separating Error-Handling Code from "Regular" Code I

- Code for the case of exceptional events is separated from main program logic
- Example

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Separating Error-Handling Code from "Regular" Code II

What could possibly go wrong?

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
```

Separating Error-Handling Code from "Regular" Code III

```
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
```

Separating Error-Handling Code from "Regular" Code IV

```
    return errorCode;
}
```

Without exception framework

- Check each condition with `if` statements
- Original seven lines of code get completely cluttered
- You would not want to read such code

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
```

Separating Error-Handling Code from "Regular" Code V

```
    doSomething;
} catch (sizeDeterminationFailed) {
    doSomething;
} catch (memoryAllocationFailed) {
    doSomething;
} catch (readFailed) {
    doSomething;
} catch (fileCloseFailed) {
    doSomething;
}
```

With exception handling

- Errors still need to be detected, reported and handled
- But: actual code is much more organised

Propagating Errors Up the Call Stack I

- Imagine an error occurs in method `readFile`
- Only `method1` cares about errors

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

Propagating Errors Up the Call Stack II

Just propagate the error

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {  call method3; }

method3 throws exception { call readFile; }
```

- Other methods do not need to detect the exception
- Exception is automatically caught in method1

Grouping and differentiating errors I

Example: `java.io.IOException`

- `IOException` most general I/O related exception
- Descendants: more specific errors
- Example: `FileNotFoundException`
- Graunularity of exception handling can be adjusted

Catching specific exceptions

```
catch (FileNotFoundException e) {
...
}
```

Grouping and differentiating errors II

Catching more general exceptions

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

Too general exception handlers

Gonna catch 'em all

- It's possible to catch any exception:

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

- `Exception` very high in the class hierarchy
- In most situations: be *as specific as possible*
- Exceptions that are too general prohibit appropriate error handling

Outline

Summary

- Exceptions can be thrown with `throw` + exception object
- If a method throws a checked exception, its method signature must contain a `throws` clause
- Exceptions are caught via
  - `try` blocks, where exceptions might occur
  - `catch` blocks, where exceptions are caught and handled
  - `finally` blocks that are guaranteed to be executed and used for cleanup

Best practises for exception handling

- Recommendation: read about best practises for exception handling
- Using exceptions is not very difficult
- Using them appropriately is challenging in the beginning
- Some interesting aspects: `http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html`

Literature

Java Tutorials
`http://docs.oracle.com/javase/tutorial/essential/exceptions`

Sierra, K. & Bates, B.
Head First Java. (Ch. 2, 4)
O'Reilly Media, 2005.

Ullenboom, Ch.
Java ist auch eine Insel. (Ch. 7)
Galileo Computing, 2012.