EXCEPTIONS AND ASSERTIONS

CONTENT

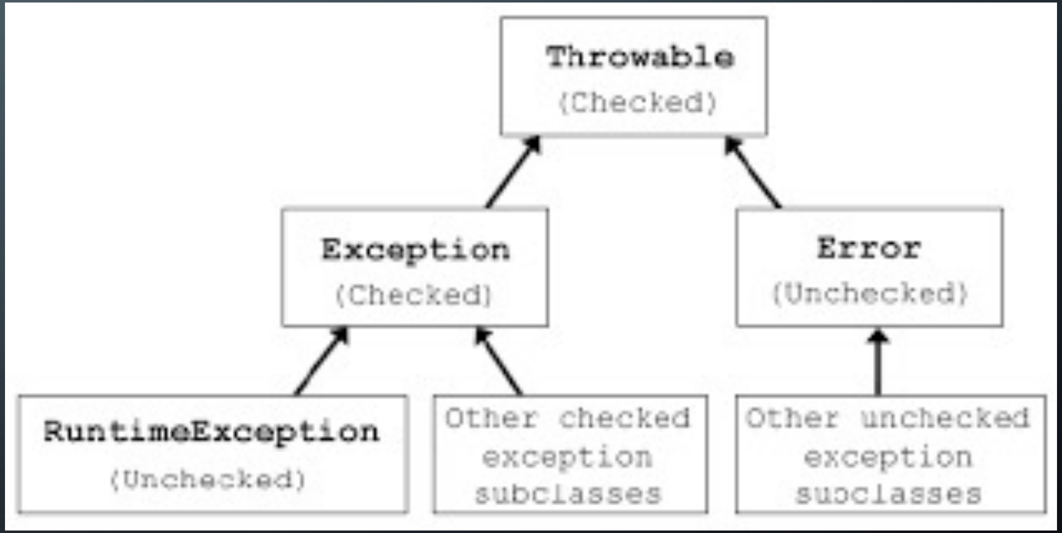
- Exceptions basics
- Custom exceptions
- Multi-catch
- Try-with-resources
- Suppressed exceptions
- Assertions

EXCEPTIONS BASICS

- Exceptions occur when the program takes the unhappy path
- For example:
  - Trying to access a resource that is not there
  - Using an index of an array that doesn’t exist
  - Calling a method on an object that is a null
- Exception is a protocol of what to do when unfortunately the happy path is going wrong
- Exceptions can occur due to problems with the code and problems that are beyond the control of a developer

EXCEPTIONS HIERARCHY

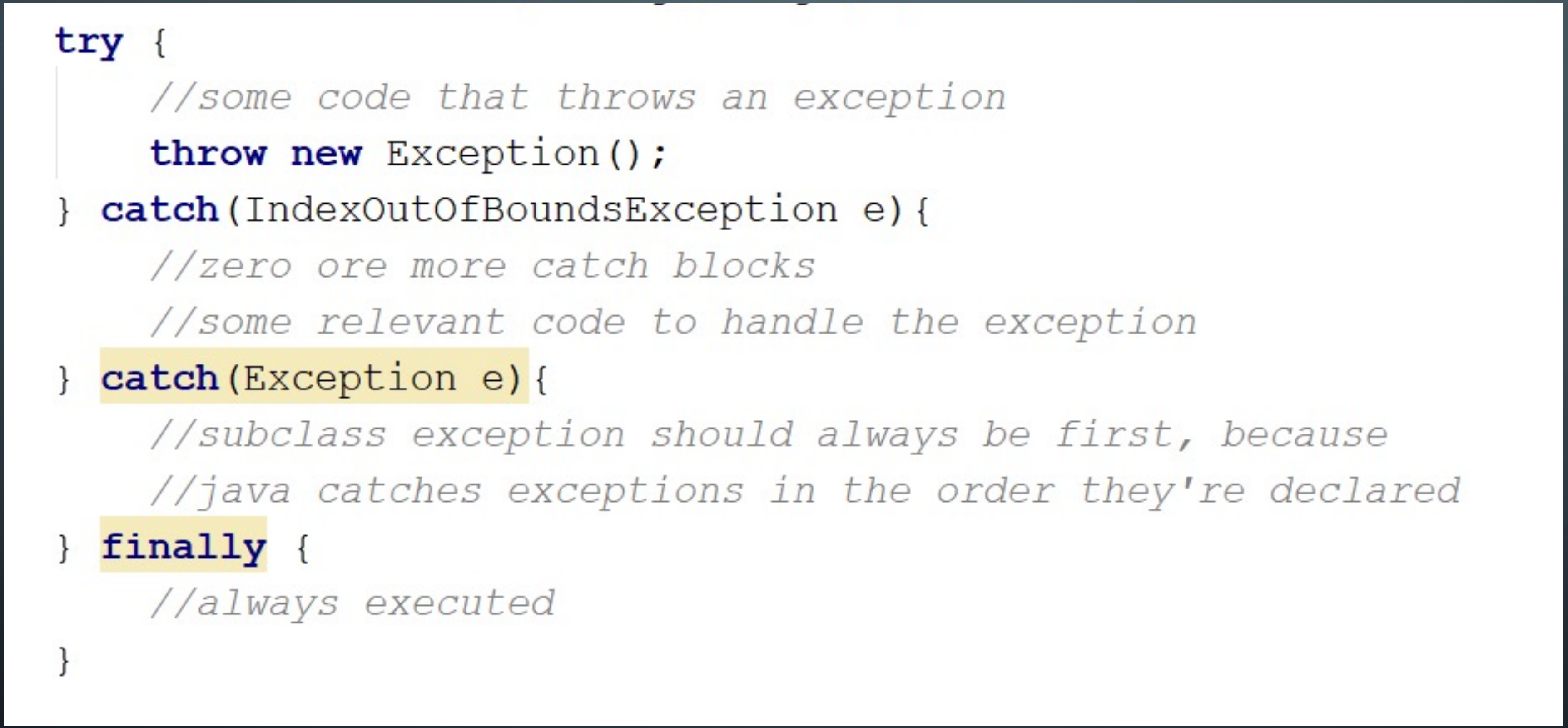
- Runtime = unchecked
- Compile time = checked >> must be handled or declared
- Error > not an exception, should not be caught, but can be caught



LIST OF EXCEPTIONS FOR OCP

Checked	Unchecked
ParseException	ArithmeticException
IOException	ArrayIndexOutOfBoundsException
FileNotFoundException	ClassCastException
NotSerializableException	IllegalArgumentException
SQLException	NullPointerException
	NumberFormatException
	ArrayStoreException
	DateTimeException
	MissingResourceException
	IllegalStateException
	UnsupportedOperationException

TRY STATEMENT



```
try {
    //some code that throws an exception
    throw new Exception();
} catch(IndexOutOfBoundsException e){
    //zero ore more catch blocks
    //some relevant code to handle the exception
} catch(Exception e){
    //subclass exception should always be first, because
    //java catches exceptions in the order they're declared
} finally {
    //always executed
}
```

WILL THIS COMPILE?

```
try {
            //some code that throws an exception
            throw new Exception();
        } catch(IndexOutOfBoundsException e){
            //zero or more catch blocks
            //some relevant code to handle the exception
        } finally {
            //always executed
        }
```

WILL THIS COMPILE?

```
try {
        //some code that throws an exception
    } catch(IndexOutOfBoundsException e){
        //zero or more catch blocks
        //some relevant code to handle the exception
    } finally {
        //always executed
    }
```

WILL THIS COMPILE?

```
try {
        //some code that throws an exception
        throw new Exception();
    }
```

WILL THIS COMPILE?

```
try {
        //some code that throws an exception
    } catch(Exception e){
        //zero or more catch blocks
        //some relevant code to handle the exception
    } catch(IndexOutOfBoundsException e) {
        //never executed
    } finally {
            //some other finally code
      }
```

WILL THIS COMPILE?

```
try {
        //some code that throws an exception
    } catch(Exception e){
        //zero or more catch blocks
        //some relevant code to handle the exception
    } catch(Exception e) {
    } finally {
          //some other finally code
     }
```

WILL THIS COMPILE?

```
try {
        //some code that throws an exception
    } catch(IndexOutOfBoundsException e){
        //zero or more catch blocks
        //some relevant code to handle the exception
    } finally {
        //always executed
    } finally {
            //some other finally code
      }
```

THROW VS THROWS

- Throw >> used to throw an exception
  - throw new Exception();
- Throws >> used to declare an exception in the method signature

  -
    ```
    public void throwSomething() throws IOException {
        // some code that might throw an IOException
    }
    ```
- Runtime exceptions don't need to be caught and therefore don't need to be declared when they're not caught

CUSTOM EXCEPTIONS

- Theoretically you can extend any exception class to create your own exception
- But it's common practice to use:
    - Exception for checked
    - RuntimeException for unchecked

EXAMPLE: CUSTOM EXCEPTION

```
public class CustomException extends Exception {
        public CustomException(){
                super();
        }
        public CustomException(Exception e){
                super(e);
        }
        public CustomException(String message){
                super(message);
        }
}
```

EXERCISE: CREATE YOUR OWN EXCEPTION CLASS

- Create two exceptions, one is checked and one is unchecked
- The checked one is thrown by a method in Cat, called catchSpider. This exception is called: UnexpectedStrongSuperSpiderException
- The unchecked exception is SpiderAteCatException
- Build some context around it so you can test it

- Bonus: justify the use of checked and unchecked exceptions for these cases

MULTI-CATCH

- You can catch multiple exceptions in the same catch block
  - This avoids double code
  - It avoids catching all exceptions as a workaround to not have double code
  - It avoids hard to read code

```
try {
        //some code that throws an IOexception
} catch(IndexOutOfBoundsException | IOException e){
            //some relevant code to handle the exception
}
```

Variable name should only appear once, and at the end

Exceptions in multi-catch cannot be related to eachother

The exception in the multi-catch block is final, and you're not allowed to reassign another exception to it

WILL THIS COMPILE?

```
try {
        //some code that throws an IOexception
} catch(IndexOutOfBoundsException | Exception e){
        //some relevant code to handle the exception
}
```

WILL THIS COMPILE?

```
try {
        //some code that throws an IOexception
} catch(IndexOutOfBoundsException e1| IOException e2){
        //some relevant code to handle the exception
}
```

WILL THIS COMPILE?

```
try {
        //some code that throws an IOexception
} catch(IndexOutOfBoundsException| IOException e){
        //some relevant code to handle the exception
}
```

WILL THIS COMPILE?

```
try {
        // some code that doesn't throw an exception
} catch(IndexOutOfBoundsException | IOException e){
        //some relevant code to handle the exception
}
```

WILL THIS COMPILE?

```
 try {
             //some code that throws an IOexception
 } catch(IndexOutOfBoundsException| IOException e){
            //some relevant code to handle the exception
            e = new IOException();
}
```

TRY-WITH-RESOURCES

- Resources that are opened in the try need(ed) to be closed in the finally, if they were indeed successfully opened
- Since Java 7 this can be written differently, called try-with-resources
- The resources opened in the try statement are automatically closed
- Resources opened in try(*resources*) have a scope that is limited to the try block
- Resources are closed in the reversed order they were opened

EXAMPLE: TRY AND TRY-WITH-RESOURCES

OLD WAY OF CLOSING RESOURCES IN TRY

```
BufferedReader in = null;

BufferedWriter out =  null;

try {

			//reading from in

			//writing to out

} finally {

			if(in != null) in.close();

			if(out != null) out.close();

}
```

NEW WAY OF CLOSING RESOURCES IN TRY

```
try(BufferedReader in = Files.newBufferedReadere(path1); BufferedWriter out = Files.newBufferedWriter(path2)) {

			//reading and writing

} //<<resources are closed here



//yes, that's right… Try-with-//resources doesn't necessarily need //a catch or explicit finally block
```

AUTOCLOSEABLE

- Resources that are used in the try-with-resources must be autocloseable. This means they implement the interface AutoCloseable
- This means the method *public void close() throws Exception* {} must be implemented
- Close method should be idempotent >> don't have side effects, can be called multiple times with same result and no side effects
- Better for close method to not throw exception, but to throw a more specific exception

EXERCISE

- Create a class CatHuntAction
- The class should be autocloseable
- The class should have a Cat and a Spider property
- The CatHuntAction logs the action of the cat hunting the spider
- In a main method, open the CatHuntAction, and call methods on the specific cat
- These methods should be able to throw your custom exceptions, handle them in a proper manner

SUPPRESSED EXCEPTIONS

- When multiple expressions are thrown, only the first one is handled, the others are suppressed
- For example: try-with-resources throws an exception in try block, and closing the resources throws another example: only the first one is handled
- This only applies to exceptions thrown in try clause, so not in catch or finally
- You can get suppressed exceptions in the catch block with e.getSuppressed(). This gives back an iterable which you can loop through

EXERCISE: SUPPRESSED EXCEPTIONS

- Change the try-with-resources from the previous assignment to throw two exceptions in the catch block
- Print caught and the suppressed exception

EXCEPTION HANDLING DO’S

- Clean up resources / use try-with-resources
- No empty catch blocks
- Catch and throw specific exceptions
- Add clear descriptions when throwing messages
- Don’t ever catch errors (so don’t catch Throwable, becauser errors are a subclass)
- Describe usage of exceptions in Javadoc

- Additions..?

ASSERTIONS

- Boolean expression that is used that should be true
- Used in non-production code to detect serious problems
- Don’t believe it until you see it? >> use an assertion, and then get rid of it again

- If an assertion is false, it throws an AssertionError and kills the program

- Assert statement: assert *boolean-expression: “optional error message*”;

- Enable assertions, else they’re ignored at runtime: use –ea or –enableassertions on cmd:
    - Java –ea SomeJavaClassName

EXAMPLE: ASSERTIONS

```
if(true) {
} else {
        assert false: “this cannot be reached”;
}
```

EXERCISE: ASSERTIONS

- Add an assert statement in your application that returns true and that the application will encounter
- Run the application with assertions enabled

- Bonus: what would have been a better way to do this than with assert?