Exceptions

Exception

Exceptions in Java are usually misunderstood by developers (at least beginners)

Probably because exception mechanism is used for several distinct objectives

Why exceptions ?

Mechanism invented to represent an **abnormal** mode of operation

Some programming languages have no exceptions

- In C, we frequently use the return value to signal an error… but what if all return values are correct? => modification of argument, passed by address :-(

C++, Java, Python, Ruby, etc. do have exceptions

There are different **kinds** of exceptions

Exceptions hold for 3 different things

- Signal **programming errors**
  - Developer did not read the doc
  - Developer has trouble with null, bounds of array...
- Signal **fatal errors**
  - StackOverflowError, OutOfMemoryError, InternalError
- Signal errors that **depend on external conditions**
  - Typically Input/Output errors

Error handling is different

- **programming errors**
  - Should only happen in dev phase, but not in production
  - You should not try to pick up on the error, but rather fix the bug that causes the exception to be launched.
- **fatal errors**
  - Should happen because of a error of the dev or of the ops
  - You should not try to pick up on the error
- **external errors**
  - Independent of the state of the program
  - You could pick up on the error, if a treatment allows the program to continue normally, or at least to display a nice error message for the user, stopping the application

In Java

These 3 types of errors have distinct types

- All sub-types of the general Throwable type

RuntimeException (you don't have to treat them)

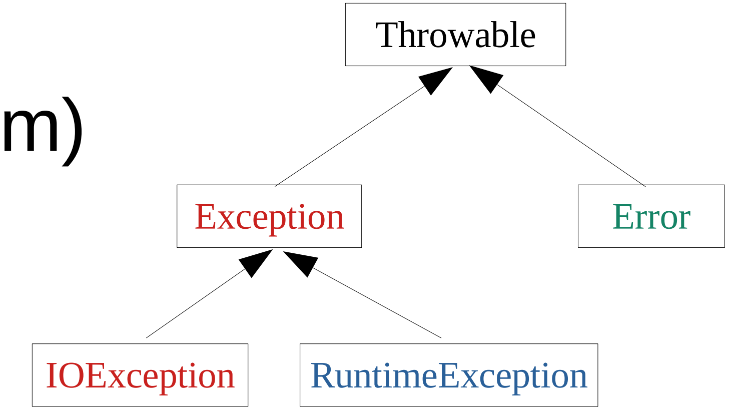
- Programming errors

Error (you don't have to treat them)

- Fatal errors

Exception (you have to treat them)

- External error



There is two main problems

- Exception does not hold for all exceptions
- RuntimeException extends Exception

Raise/throw an exception

JVM can raise exceptions by itself:
NPE, AIOOBE, CCE, OutOfMemoryError, etc.

Syntax **throw** allows us to raise an exception

```
throw new IllegalArgumentException("invalid value");
```

Exception go back in the execution stack until been « catch » by a method, or by the JVM if exception reach method main().

StackTrace

When **created** an exception (when its constructor is called), JVM saves the stack of method calls until the call to “new” ot this exception

Exception creation has a cost in execution time in order to create this « stack trace »

But throwing or catching an exception is cheap

- a small cost enterring a “try” and one “instanceof” for each “catch”

Checked Exception

Compiler requires to deal with all sub-types of Exception that are not of type RuntimeException

- These exceptions are called **checked** exceptions

There is two way do deal with a **checked exception**

- Either declare that the method can raise such exception with the key-word **throws**
- Or handling the exception with **try-catch** syntax

Key-word throws

Signal that the method can raise a checked-exception

```
public static void sayHello(Writer writer)
                                throws IOException {
  writer.write("hello");
}
public static void main(String[] args)
                                throws IOException {
  sayHello(System.console().writer());
}
```

Compiler ignores throws on unchecked-exceptions

try/catch syntax

Allow to specify a code to pick up on the error

```
public static void sayHello() throws IOException {
  writer.write("hello");
}

public static void main(String[] args) {
  try {
    sayHello(System.console().writer());
  } catch(IOException e) {
    System.err.println("can't write on console\n" +
          e.getMessage());
    System.exit(1);
  }
}
```

Multiple catch

You could write several catch blocks

```
try {
  writeOnHDOrNetwork();
} catch(IOException e) {
  // ...
} catch(NetworkException e) {
  // ...
}
```

If the same code fits both catch, both exceptions can be gathered (with a '|')

```
try {
  writeOnHDOrNetwork();
} catch(IOException | NetworkException e) {
  // A single common code
}
```

Finally

A optional “finally” clause is possible

```
try {
  foo();
} catch(IOException e) {
  // executed if IOException is raised in try
} finally {
  // finally executed
}
```

Useful to perform some mandatory treatments, like freeing resources

Throws or try/catch ?

When should we use throws and when should we use try/catch ?

- If you can write something in the catch clause to be able to pick up on the error, you could use try/catch to apply corrective treatment
  => the program will continue as if no problem happened
- If not, use throws, to signal the problem

statistically there is much more throws than try/catch !

and **only** for checked-exceptions !

try/catch of the death

In Javan exceptions type hierarchy sucks

- RuntimeException extends Exception :-(
- Then, writing a catch(Exception) is not a good idea => it will not be easy to write a code that picks up on error, because you don’t know if the error is a programming one, or an external one...
  - You will just pretend that everything is fine:-/
- Same problem with catch(Throwable)

Checked-exception and overriding

Compiler checks that an overriding method cannot raise some checked-exception that are not expected in the overridden method

```
public interface Runnable {
  public void run();
}

public class HelloRunnable implements Runnable {
  public void run() throws IOException {
    // does not compile !
  }
}
```

Exception tunneling

Sometimes, you can « wrap » a checked exception in an unchecked exception, and then get it out with getCause()

```
public class HelloRunnable implements Runnable {
  public void run() {
    try {
      foo() ; // can raise IOException
    } catch(IOException e) {
      throw new UncheckedIOException(e);    // wrap
    }
  }
}
```

-
  ```
  public static void main(String[] args) throws IOException {
    Runnable runnable = new HelloRunnable();
    try {
      runnable.run();
    } catch(UncheckedIOException e) {    // get it out
     throw e.getCause();                 // and raise it!
    }
  }
  ```

Defensive Programming &
Contract programming
(design-by-contract programming)

Bug fixing

The latter a bug is discovered in the softwre life cycle, the more expensive it is to fix.

=> defensive programming

All argument received by a public method must be verified before to be used

=> contract programming

The job of constructor

Don’t trust argument values

- A field of an object is more often read than wrote
- OOP: the state of an object should always be valid

The, a constructor has to verify its arguments before assigning their values into fields

Example

Awful code :(

```java
public class Author {
  private final /*maybe null*/ String firstName;
  private final /*maybe null*/ String lastName;

  public Author(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public boolean equals(Object o) {
    if (!(o instanceof Author)) {
      return false;
    }
    Author author = (Author)o;
    return ((author.firstName == null && firstName == null) ||
                 author.firstName.equals(firstName)) &&
            (author.lastName == null && lastName == null) ||
                 author.lastName.equals(lastName)));
  }
}
```

Must check if null
at each read :((

Example

Better version of same code

```
public class Author {
  private final String firstName;
  private final String lastName;

     public Author(String firstName, String lastName) {
       this.firstName = Objects.requireNonNull(firstName);
       this.lastName = Objects.requireNonNull(lastName);
     }

     public boolean equals(Object o) {
       if (!(o instanceof Author)) {
         return false;
       }
       Author author = (Author)o;
       return author.firstName.equals(firstName) &&
                  author.lastName.equals(lastName);
     }
   }
```

Single check if null
at assignment **write**

Contract programming

All public method must document their contract

- What it does
- What are expected arguments
- What are possible return values
- What exceptions are raised and why ?

- Normally, dev should read this doc ! Practically, the doc is only read by the dev when the behavior is not the one he expected :(

Javadoc

Java provides a documentation syntax and format allowing to localize it directly in the source code

- Eases a doc up to date with respect to the code

Do not confuse « code comment » and « documentation comment »

- Documentation comment shows how to use the method from a user's point of view.
- Code comment shows there is something unusual in the code (it is intended for the developer)

Example

Source code to implement a stack...

```
public class IntStack {
  private final int[] array;
  private int top;

  public IntStack(int capacity) {
    array = new int[capacity];
  }

  public void push(int value) {
    array[top++] = value;
  }

  public int pop() {
    return array[--top];
  }
}
```

Defensive programming

```
public class IntStack {
  private final int[] array;
  private int top;

  public IntStack(int capacity) {
    if (capacity < 0) {
      throw new IllegalArgumentException("capacity < 0");
    }
    array = new int[capacity];
  }

  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
  }

  public int pop() {
    if (top == 0) {
       throw new IllegalStateException("stack is empty");
    }
    return array[--top];
  }
}
```

Contract programming

```java
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Create an integer stack with a maximum capacity.
   * @param capacity the capacity of the stack
   * @throws IllegalArgumentException if the capacity
   * is less than 0
   */
  public IntStack(int capacity) {
    if (capacity < 0) {
      throw new IllegalArgumentException("capacity < 0");
    }
    array = new int[capacity];
  }

  ...
}
```

Contract programming

```
public class IntStack {
  ...

  /**
   * Put the value on top of the stack.
   * @param value the value to push on the stack.
   * @throws IllegalStateException if the stack is full
   */
  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
  }

  ...
}
```

Contract programming

```
public class IntStack {
  ...
  /**
   * Extract and return the value on top of the stack.
   * @return the value on top of the stack.
   * @throws IllegalStateException if the stack is empty
   */
  public int pop() {
    if (top == 0) {
       throw new IllegalStateException("stack is empty");
    }
    return array[--top];
  }

  ...
}
```

Contract programming

Whereas defensive programming consists in testing pre-conditions...

One may also want to test if the code has done its job by testing post-conditions and invariants.

- Post-condition: output state of an algorithm
- Invariant: always true condition for the implementation of the class.

assert syntax

assert syntax allows some code to be tested while its execution

- assert i == j;
- assert i == j: "error message";

in Java, **assert** are only executed if JVM is run with

```
java -ea Prog
```

(ea = enable assert) : then, on while dev phase… and off in production

Invariants & post-condition

```java
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Put the value on top of the stack.
   * @param value the value to push on the stack.
   * @throws IllegalStateException if the stack is full
   */
  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
    assert array[top - 1] == value;
    assert top >= 0 && top <= array.length;
  }

  ...
}
```

post-condition

invariant

Contract programming and unit testing

Unit testing like post-conditions and invariants are also testing program execution

Post-condition and invariant

- Tests with any real data, inside the code

Unit testing

- Tests with data at boundaries, outside the code

Then, both are required !