
Exceptions



|



Introduction

- ▶ When executing, abnormal situations (or exceptions) may happen, due to:
 - ▶ Coding errors made by the programmer:
 - ▶ Errors caused by wrong input
 - ▶ Unforeseen external conditions (network disconnected, data file deleted,...)
- ▶ On these cases, Java by default will stop and generate an error message
 - ▶ `int[] values = {1, 2, 4};`
`System.out.println(values[10]);`
`System.out.println("Task done!");` `// unreachable`
 - ▶ Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 10 out of bounds for length 3

Exception Handling

- ▶ Technically said: Java will throw an exception
 - ▶ One can handle these situations to change the default behavior by catching the exception

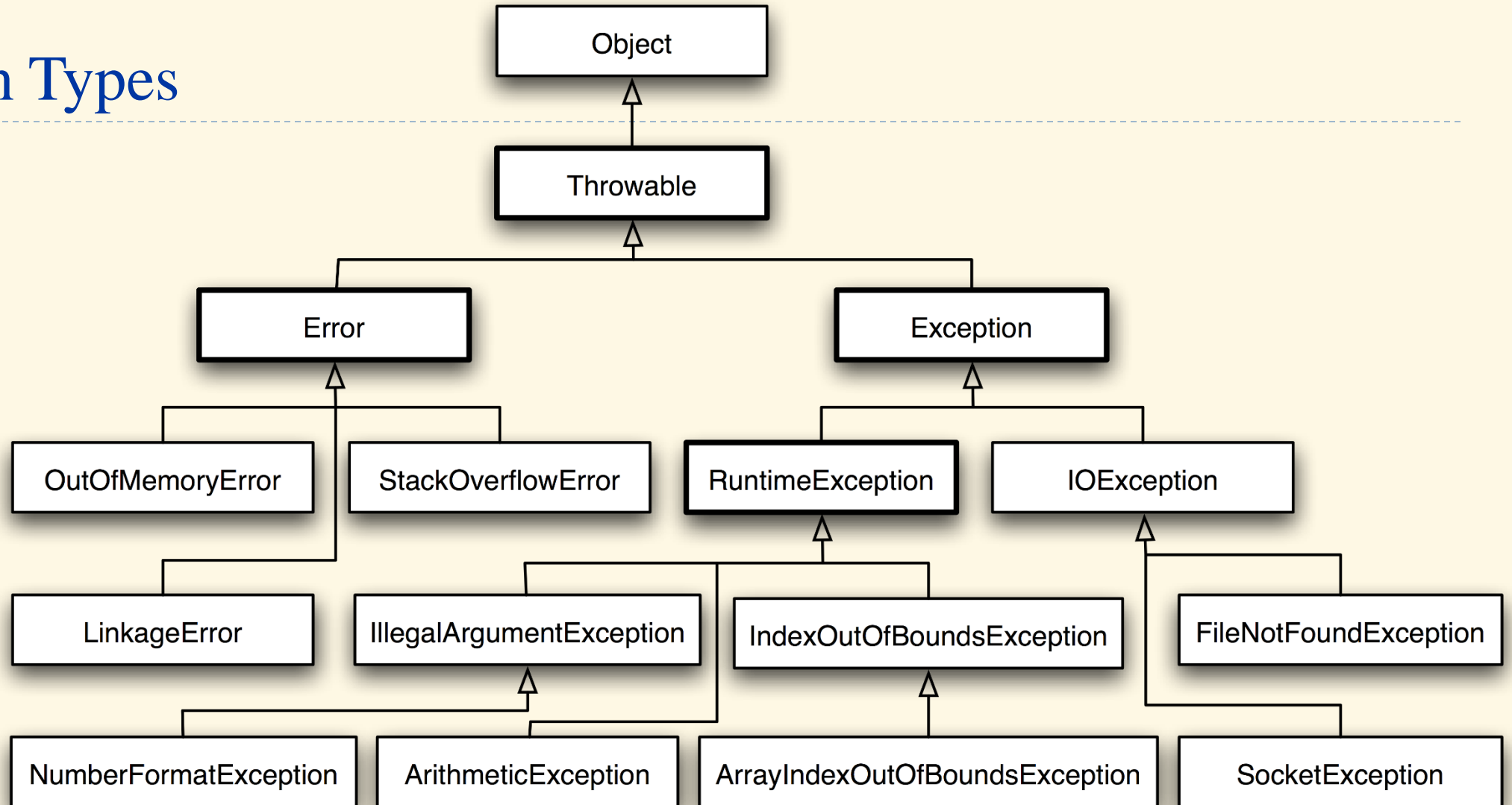
- ▶ Example:

```
▶ try {  
    int[] values = {1, 2, 4};  
    System.out.println(values[10]);  
    System.out.println("Success!"); // unreachable  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.err.println("Error: " + e.getMessage());  
    System.out.println("Error recovered");  
}
```

```
System.out.println("Task done!"); // reachable
```

- ▶ Error: Index 10 out of bounds for length 3
Error recovered
Task done!

Exception Types



- ▶ This are just some most used ones of the built-in exception types!

Exception Objects

- ▶ An exception object is an instance of an exception class
 - ▶ Can be thrown with the `throw` keyword
 - ▶ Any uncaught exception will cause the program to stop
- ▶ Inside the `catch` block, one may re-throw the caught exception object, or throw another new exception if necessary

- ▶ Example:

```
static String date2String(int day, int month, int year) {  
    if (day <= 0 || day > 31)  
        throw new IllegalArgumentException("Invalid day value");  
    if (month <= 0 || month > 12)  
        throw new IllegalArgumentException("Invalid month value");  
    if (List.of(2, 4, 6, 9, 11).indexOf(month) >= 0 && day == 31)  
        throw new IllegalArgumentException("Invalid day value for the  
given month");  
  
    return String.format("%02d/%02d/%04d", day, month, year);  
}
```

Handling Multiple Exceptions

- ▶ One code may throw different types of exceptions. There are several ways to handle (choose one to use depending on situation):
 - ▶ Multiple nested `try ... catch`
 - ▶ Multiple sequential `try ... catch`
 - ▶ A single `try` with multiple `catch`
 - ▶

```
try {  
    doSomething1(); // may throw FileNotFoundException  
    doSomething2(); // may throw FileAlreadyExistsException  
}  
catch(FileNotFoundException e) { //... }  
catch(FileAlreadyExistsException e) { //... }
```
 - ▶ A single `try ... catch` with piped classes
 - ▶

```
try { // ... }  
catch(FileNotFoundException | FileAlreadyExistsException e) { //... }
```
 - ▶ A single `try ... catch` with a super exception class
 - ▶

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

Checked and Unchecked Exceptions

- ▶ Checked exceptions are the ones that are checked at compile-time, and need to be explicitly handled in code
 - ▶ Examples: `IOException`, `FileNotFoundException`, `ClassNotFoundException`,...
 - ▶ A method may leave the handling of some of its checked exceptions to the callers with the `throws` keyword
 - ▶

```
MyData loadData(String filepath)
    throws FileNotFoundException, FileAlreadyExistsException
{ // ... }
```
- ▶ Unchecked exceptions are the other ones
 - ▶ Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`,...

User-defined Exceptions

- ▶ One may create his own types of exceptions for capturing additional information by extending the `Exception` class

- ▶ Example:

```
class DateException extends Exception {  
    private int day, month, year;  
  
    DateException(String message, int day, int month, int year) {  
        super(message);  
        this.day = day;  
        this.month = month;  
        this.year = year;  
    }  
  
    // ...  
}  
  
throw new DateException("Invalid day value", day, month, year);
```


finally Block

- ▶ The **finally** block always executes when the **try** block exits, even when an exception occurs
 - ▶ It is useful for cleanup code to avoid being accidentally bypassed by a **return**, **continue**, **break** or an exception
 - ▶ Good practice to put cleanup code in a **finally** block, even when no exceptions are anticipated.
- ▶ Example:

```
int safeFunction() {  
    SomeResource rc = acquireResource();  
    try {  
        doGoodThing(); // may throw GoodException  
        doBadThing();  // may throw BadException  
        return 0;  
    } catch(GoodException e) {  
        return 1;  
    } catch(BadException e) {  
        return 2;  
    } finally {  
        rc.release();  
    }  
}
```

Assertions

- ▶ An `assert` statement is used to declare an expected `boolean` condition in a program, which is checked at runtime (the program must be running with assertions enabled)
 - ▶ If the condition is false, the Java will throw an `AssertionError` exception
 - ▶ To run a program with assertions enabled:
 - ▶ `java -ea <Program>`
- ▶ Programmers usually use `assert` for program testing and verification purposes, and try to ensure that assertions never fail in production programs
 - ▶ Enable assertions only in development time
- ▶ Example:
 - ▶

```
MyData data = loadData(dataFilePath);  
assert data != null;
```

Exercise

- ▶ Write a program to repeatedly ask the user for a numerator and a divisor, and print the division result, but the invalid-user-input and division-by-zero situations must be handled correctly by your own exception class(es)