Object Oriented Programming in Java

7: Exceptions

Licence

Traditional approach to error handling

- What to do when an error or unexpected situation occurs?
  - stop the program?
  - return (if possible) wrong or error value or set error status?
- What older languages (e.g. C) have done when error had occurred?
  - e.g. C function to read a character from standard input

    ```
    int getchar(void);
    ```

    - In case of error the method returns EOF (constant defined as -1)
    - If the result is not EOF, then it is valid, and it should be casted to (unsigned) char
  - This is a typical example of misuse (or accommodation) of return value and leads to a code with many *if* statements tangling normal code and code for error-handling

Modern approach - exception

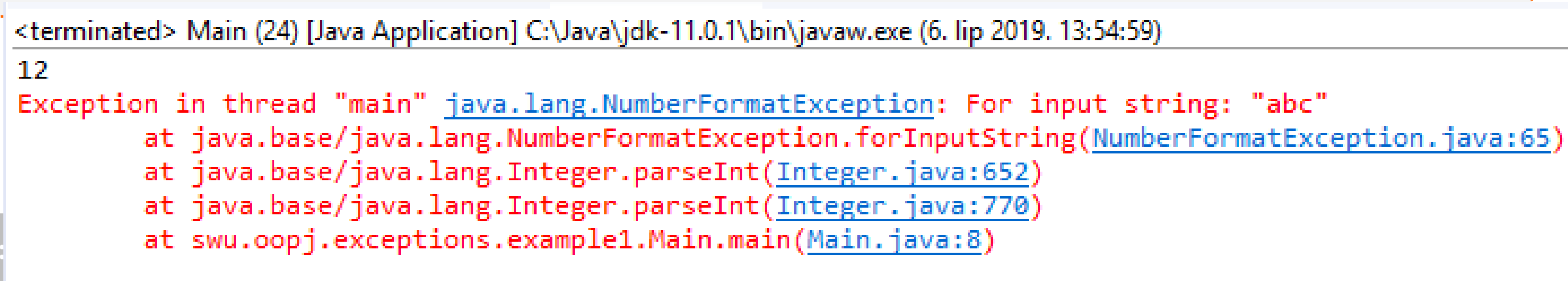
- No reason to misuse return value
  - If the method executes successfully that return value is always valid
  - *Exception* for exceptional situations that stops the normal execution of a program
- Exception is an object that holds the information why and where normal method execution had been stopped
  - The exception object is *thrown* from a method and it can be *caught later*
  - Further execution starts from the point where the exception had been caught (if ever…)

An example of an exception

- Parsing string in order to extract an integer succeeds only if the number is stored inside the string
  - If not, *parseInt* method cannot continue it execution
    - returning zero, -1, or any magic number is not an option. How to e.g. distinguish error from string containing exactly that number (i.e. "-1")
    - *parseInt* stops its further execution and throws an object (exception) of type *NumberFormatException*

```java
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<arr.length; i++) {
	int num = Integer.parseInt(arr[i]);
	System.out.println(num);
}
System.out.println("Done");
```

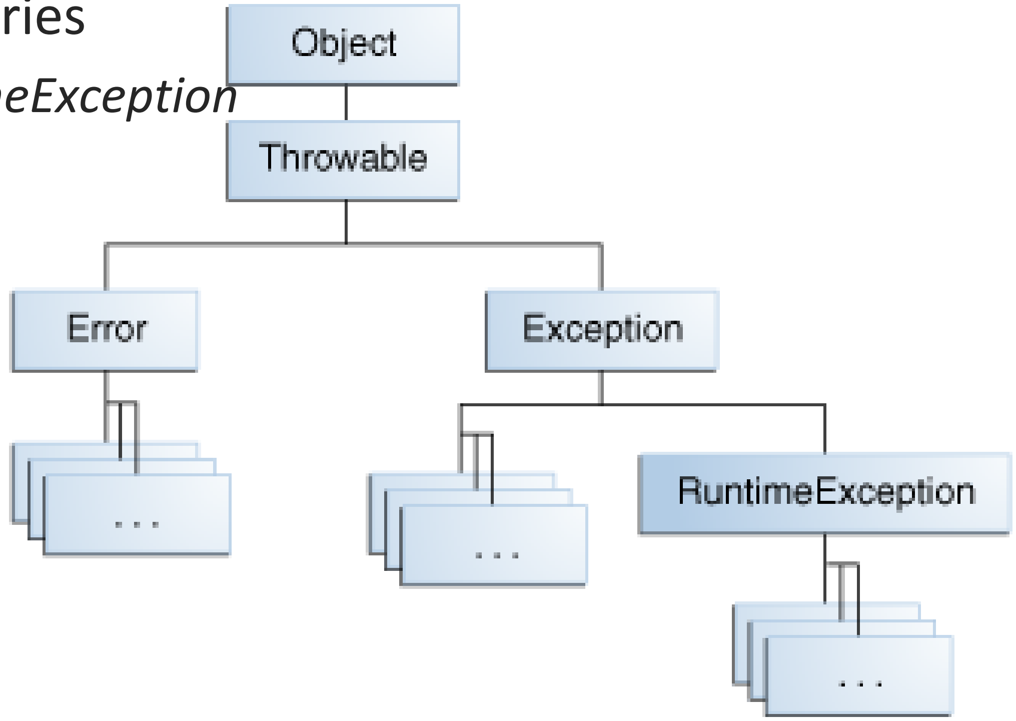
07_Exceptions/.../example1/Main.java



```
<terminated> Main (24) [Java Application] C:\Java\jdk-11.0.1\bin\javaw.exe (6. lip 2019. 13:54:59)
12
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
	at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
	at java.base/java.lang.Integer.parseInt(Integer.java:652)
	at java.base/java.lang.Integer.parseInt(Integer.java:770)
	at swu.oopj.exceptions.example1.Main.main(Main.java:8)
```

Types of objects used for exceptions

- An object thrown as an exception is of Throwable type or some of its subclasses
  - Divided in three main categories
    - *Error*, *Exception* and *RuntimeException* (discussed later)
- Some of commonly used exception types are
  - *NullPointException*
  - *ClassCastException*
  - *ArithmeticException*
  - *IllegalArgumentException*
  - *IndexOutOfBoundException* with two sublasses:
    - *ArrayIndexOutOfBoundException*
    - *StringIndexOutOfBoundException*



Throwable class

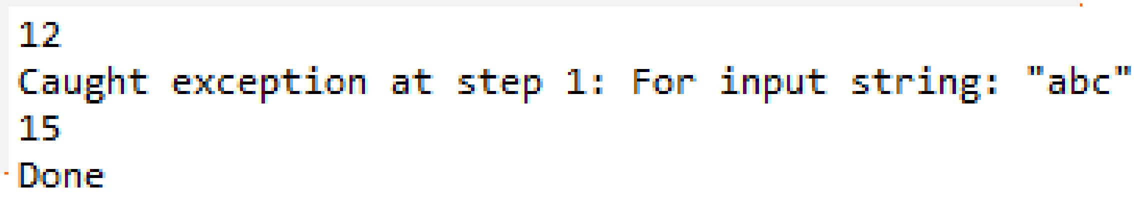
- *Throwable* class contains (among others) methods for:
  - Descriptive exception message
  - Stack trace (order of method calls preceding the exception)
    - E.g. main() → m1() → m2() → m3() → exception!
    - Enables locating exact position of an exception (source filename, and in some cases line number)
    - Stack trace can be printed to some output or get as an array of stack trace elements
- *Throwable's* subclasses can contain addition useful information depending on exception type

Catching an exception

- We expect an exception while parsing string, and know how to recover after it happens
- Code that could cause the exception is wrapped inside try-catch block

07_Exceptions/…/example2/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<arr.length; i++) {
  try{
       int num = Integer.parseInt(arr[i]);
       System.out.println(num);
  }
  catch(NumberFormatException exc){
       System.out.format("Caught exception at step %d: %s%n",
              i, exc.getMessage());
  }
}
System.out.println("Done");
```



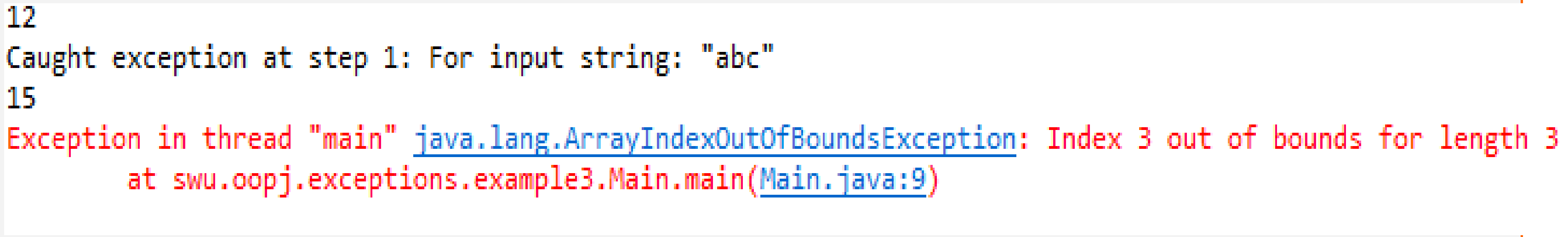
```
12
Caught exception at step 1: For input string: "abc"
15
Done
```

Different exception types

- Small change of the previous program causes another exception
  - Trying to access element at the index out of array range
    - this exception is not caught

07_Exceptions/…/example3/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
  }
  catch(NumberFormatException exc){
        System.out.format("Caught exception at step %d: %s%n",
              i, exc.getMessage());
  }
}
System.out.println("Done");
```



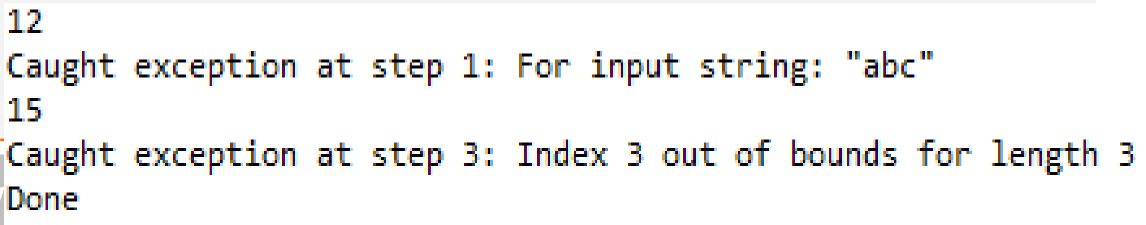
```
12
Caught exception at step 1: For input string: "abc"
15
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
	at swu.oopj.exceptions.example3.Main.main(Main.java:9)
```

Try-catch blocks with more that one catch part

- We can have more that one catch part of *try-catch* block

07_Exceptions/…/example4/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
       int num = Integer.parseInt(arr[i]);
       System.out.println(num);
  }
  catch(NumberFormatException exc){
       System.out.format("Caught exception at step %d: %s%n",
              i, exc.getMessage());
  }
  catch(ArrayIndexOutOfBoundsException exc){
       System.out.format("Caught exception at step %d: %s%n",
              i, exc.getMessage());
  }
}
System.out.println("Done");
```



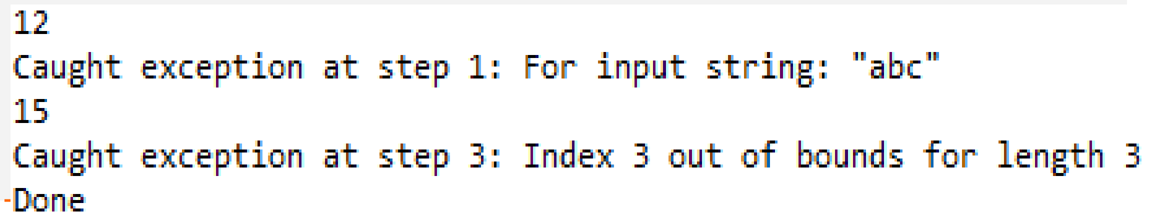
```
12
Caught exception at step 1: For input string: "abc"
15
Caught exception at step 3: Index 3 out of bounds for length 3
Done
```

Multi-catch

- In case two catch parts uses the same code, they can be joined to one multi-catch part using operator |

07_Exceptions/…/example4/MainMultiCatch.java

```java
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
      int num = Integer.parseInt(arr[i]);
      System.out.println(num);
  }
  catch(NumberFormatException | ArrayIndexOutOfBoundsException exc) {
      System.out.format("Caught exception at step %d: %s%n",
            i, exc.getMessage());
  }
}
System.out.println("Done");
```



```
12
Caught exception at step 1: For input string: "abc"
15
Caught exception at step 3: Index 3 out of bounds for length 3
Done
```

Exception handling with *try-catch* blocks

- Code that can throws an exception is written inside try block followed by one or more catch blocks
- Order of catch blocks is important
  - When exception occurs, execution continues from the first catch block that has exception type that is of thrown type or its superclass
  - Other catch blocks are ignored
- If there is no appropriate catch block exception is thrown further
  - Uncaught exception causes JVM to terminate
  - See **07_Exceptions/…/example5/ExampleStackStrace.java** for an example
- Reminder: <u>throwing exception is not like calling a method</u>, there is no return to the line followed by the line that caused the exception
  - Note: In the previous example(s) try-catch is inside for loop. What would happen it was vice-versa (i.e. for inside try-catch)

Throwing an exception

- A programmer can also throw an exception with *throw someobject* where the object is subclass of *Exception* class
  - e.g. calculating perimeter of something that is not triangle could be misleading and cause (logical) errors somewhere else

```
public static void main(String[] args) {
  try{
       if (perimiter(5, 4, 3) > perimiter(3, 2, 1))
              //do something...
  }
  catch(Exception exc){ System.out.println(exc); }
}
public static int perimiter(int a, int b, int c){
  if (!(a + b > c && a + c > b && b + c > a))
     throw new IllegalArgumentException(
       String.format("%d %d and %d cannot make triangle", a, b, c));
  return a + b + c;
}
```

07_Exceptions/…/example6/ThrowingAnException.java

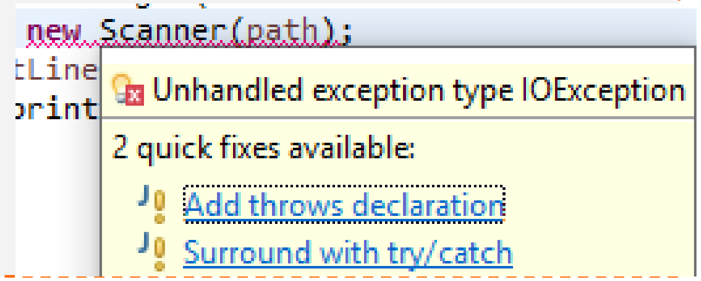
Unchecked exceptions

- Exceptions that had been used in previous examples were **unchecked** exception
  - Exception that can occur at any time, e.g.
    - Converting string to number → *NumberFormatException*
    - Using wrong array index → IndexOutOfBoundsException
    - Dereferencing null references (invoking method or accessing variable using reference that is null) → *NullPointerException*
    - Wrong downcasting, e.g. trying to downcast *Item* reference as *Food* reference, when reference hold an address of *Beverage* object → *ClassCastException*
- We may have try-catch blocks for code that could throw such exceptions, but we don’t have to
- All unchecked exception are derived from *RuntimeException*

Checked exceptions (1/4)

- Suppose that we would like to read some lines from a file
    - The easiest way to do that is using Scanner
- Such code would look like the code below, but it has compilation errors
    - There is nothing wrong with constructor arguments

```
public static void main(String[] args) {
        Path path = Paths.get("src/main/resources/dates.txt");
        Scanner s = new Scanner(path);
        String firstLine = s.nextLine();
        System.out.println(firstLine);
        s.close();
}
```


```
new Scanner(path);
```
Unhandled exception type IOException

2 quick fixes available:

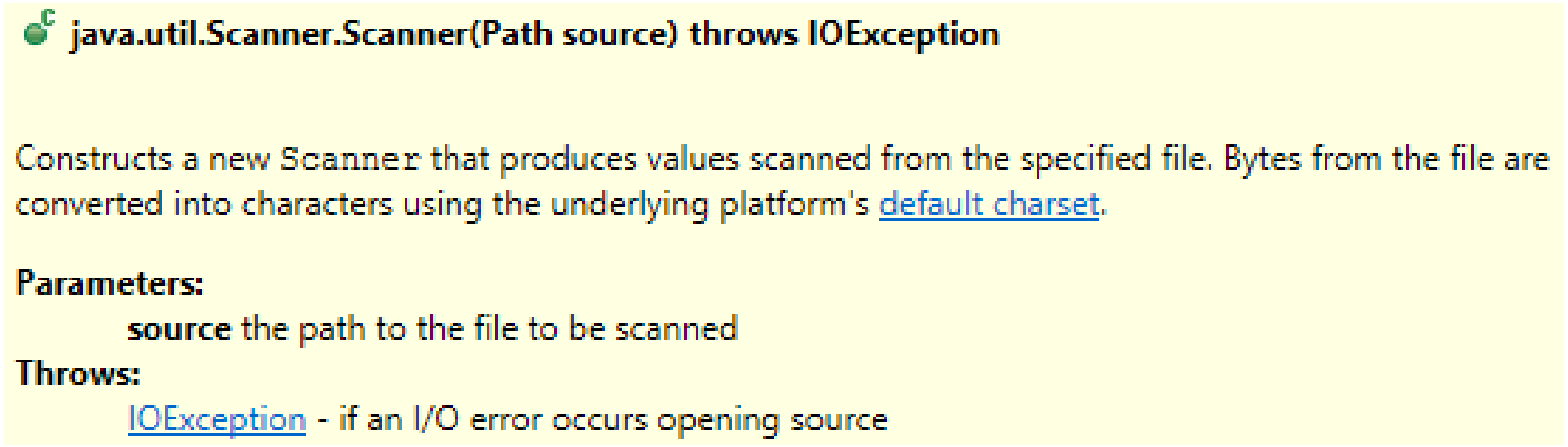
Add throws declaration

Surround with try/catch

- The code does not compile because it does not handle the exception that can happen in Scanner's constructor

Checked exceptions (2/4)

- What is different in Scanner's constructor?



java.util.Scanner.Scanner(Path source) throws IOException

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:
    source the path to the file to be scanned
Throws:
    IOException - if an I/O error occurs opening source

- It explicitly states that it throws (i.e. could throw) an exception of type *IOException* which has Exception as superclass and not *RuntimeException*
  - Such exceptions are called **checked** exceptions
- Exception handling must be done for a code using methods that throws checked exception!

Checked exceptions (3/4)

- A possible solution is to wrap the code inside try-catch block

```
public static void main(String[] args) {
  Path path = Paths.get("src/main/resources/dates.txt");
  try {
        Scanner s = new Scanner(path);
        String firstLine = s.nextLine();
        System.out.println(firstLine);
        s.close();
  catch (IOException e) {
        e.printStackTrace();
  }
}
```

07_Exceptions/…/example7/Main.java

Checked exceptions (4/4)

- Another is not to handle an exception, but to declare that it could be thrown from the method (in this case it is main)
  - … and the “problem goes” to a method that invokes such method

```
public static void main(String[] args) throws IOException {
       Path path = Paths.get("src/main/resources/dates.txt");
       Scanner s = new Scanner(path);
       String firstLine = s.nextLine();
       System.out.println(firstLine);
       s.close();
}
```

07_Exceptions/…/example7/MainWithThrows.java

- Besides checked and unchecked exception, third class of exceptions exists with class *Error* as superclass

Note about throws and unchecked exceptions

- When writing custom methods we can state that it can throws any kind of exception, even unchecked
- However, exception handling is required only if it throws checked exception

Code that must be run regardless of exception occurrence

- In some cases, part of code must be run regardless of exception occurrence and regardless if the exception is caught or not
  - E.g. we have to close Scanner object used to read from a file
  - Naïve approach would be to put close after the *try-catch* block, but what happens if some other exception (except IOException) occurs

```
Path path = Paths.get("src/main/resources/dates.txt");
Scanner s = null;
try {
    s = new Scanner(path);
    String firstLine = s.nextLine();
    LocalDate date = LocalDate.parse(line); //DateTimeParseException?
}
catch (IOException e) {
   e.printStackTrace();
}
s.close();
```

Code that must be run regardless of exception occurrence – *finally* block

- Code in finally block is always executed regardless if exception occurs or not and even if there is uncaught exception
  - Note: we have to declare s outside the try block and check for null in case that exception occurs in Scanner's constructor

```
Path path = Paths.get("src/main/resources/dates.txt");
Scanner s = null;
try {
	s = new Scanner(path);
	String firstLine = s.nextLine();
	LocalDate date = LocalDate.parse(firstLine);
	System.out.format("Day in year: %d%n", date.getDayOfYear());
}
catch (IOException e) { e.printStackTrace(); }
finally {
	System.out.println("This code is always run");
	if (s != null) s.close();
}
```

07_Exceptions/…/example8/ScannerTryCatchFinally.java

Finally block

- It can exist try-finally without catch
  - See **07_Exceptions/…/example9/TryFinallyWithoutCatch.java** for details
- Reminder: Exception does have to be caught in order to execute finally block
  - See **07_Exceptions/…/example9/ExampleWithFinally.java**
    for details and try to experiment with various catch blocks

Try-with-resources (1/2)

- If a class implements *Closeable* or *AutoCloseable* (both defines method close) then a special variant of try block can be used
- For every non-null object given inside parenthesis close is automatically called when execution leaves try block for any reason
  - Compiler generates another try-finally block

```
Path path = Paths.get("src/main/resources/dates.txt");
try (Scanner  s =  new Scanner(path)) {
       String firstLine = s.nextLine();
       LocalDate date = LocalDate.parse(firstLine);
       System.out.format("Day in year: %d%n", date.getDayOfYear());
}
catch (IOException e) {
       e.printStackTrace();
}
```

07_Exceptions/…/closeable/ScannerTryWithResources.java

Try-with-resources (2/2)

- Demonstration of try-with-resource is made using custom class
  - See the main program for details **07_Exceptions/…/closeable/Main.java**

```
package swu.oopj.exceptions.closeable;
public class Resource implements AutoCloseable {
	private int i;
	public Resource(int n){
		System.out.println("Creating #" + n);
		i = n;
	}
	@Override
	public void close()  {
		System.out.println("Closing #" + i);
	}
}
```

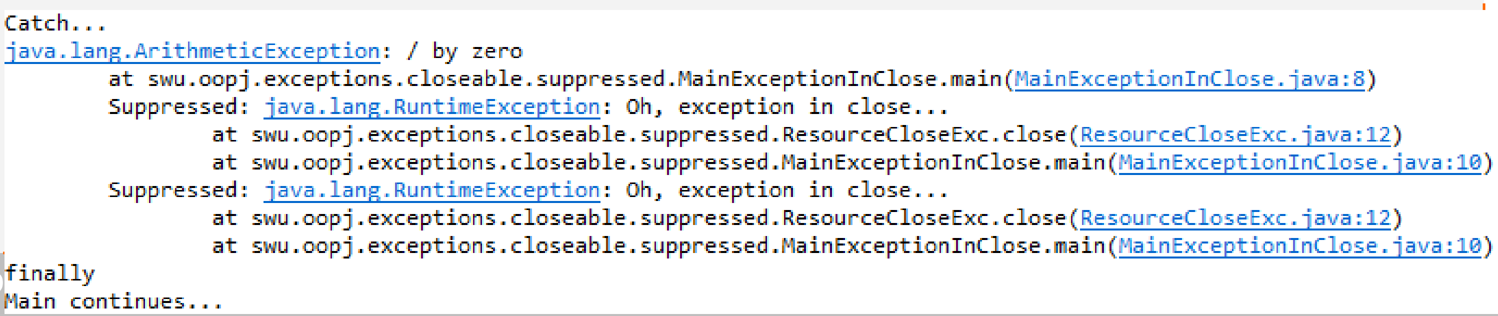
07_Exceptions/…/closeable/Resource.java

Try-with-resources – exceptions in close block

- If an exception occurs in *close* method, then
  - If there was no exception before, the exception from *close* is thrown
  - If there is a thrown exception, then the exception from *close* is suppressed (can be accessed with *getSuppressed* method)

```
public static void main(String[] args) {
  try(ResourceCloseExc r1 = new ResourceCloseExc(1);
      ResourceCloseExc r2 = new ResourceCloseExc(2)){
          int a = 5, b = 0; a = a / b;
          System.out.println("Try block ends.");
   }
   catch (Exception e) {
        System.out.println("Catch..."); e.printStackTrace(System.out);
   }
   finally{ System.out.println("finally"); }
   System.out.println("Main continues...");
```

07_Exceptions/…/closeable/suppressed/*.java



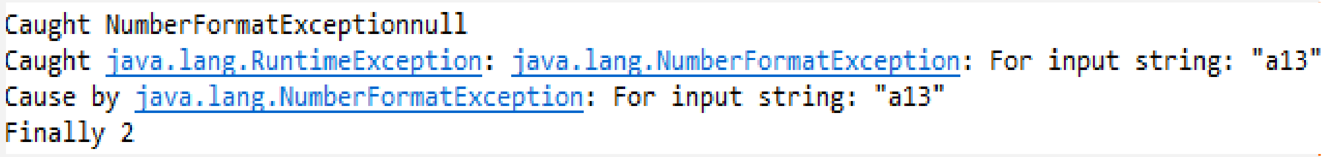
```
Catch...
java.lang.ArithmeticException: / by zero
	at swu.oopj.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:8)
	Suppressed: java.lang.RuntimeException: Oh, exception in close...
		at swu.oopj.exceptions.closeable.suppressed.ResourceCloseExc.close(ResourceCloseExc.java:12)
		at swu.oopj.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:10)
	Suppressed: java.lang.RuntimeException: Oh, exception in close...
		at swu.oopj.exceptions.closeable.suppressed.ResourceCloseExc.close(ResourceCloseExc.java:12)
		at swu.oopj.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:10)
finally
Main continues...
```

Wrapping an exception

- Exceptions can be wrapped into another exception.
  - Wrapped exceptions can be retrieved using *getCause* method

07_Exceptions/…/wrap/WrapException.java

```
String s = "a13";
try {
        try  {
                int i = Integer.parseInt(s);
        }
        catch (NumberFormatException exc) {
                System.out.println("Caught NumberFormatException");
                throw new RuntimeException(exc);
        }
}
catch (Exception e) {
        System.out.println("Caught " + e);
        System.out.println("Cause by " + e.getCause());
}
finally {
        System.out.println("Finally 2");
}
```



```
Caught NumberFormatExceptionnull
Caught java.lang.RuntimeException: java.lang.NumberFormatException: For input string: "a13"
Cause by java.lang.NumberFormatException: For input string: "a13"
Finally 2
```

Custom exceptions

- In most cases built-in exception type are enough. However, custom exception types can be created
- E.g. let’s define few custom exception types for our simple library that works with matrices
  - *MatrixException* as a root from with all custom exceptions for matrices are derived
  - *IncompatibleMatrixException* that should be thrown if adding two matrices of different dimension
  - *SingularMatrixException* to be thrown when inverting a matrix that is not invertible

What to choose as a base class for custom exceptions? (1/2)

- All exceptions are directly or indirectly inherited from *Throwable*
  - However, Throwable is too general to use it as a base class
- Class *Error* is used for group of exceptions for which is expected that program cannot recover and should not try to catch them

→ *RuntimeException* or *Exception* (unchecked or checked)

  - Historical controversy:
    - Unchecked Exceptions — The Controversy http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
    - Java theory and practice: The exceptions debate http://www.ibm.com/developerworks/library/j-jtp05254/
    - The Trouble with Checked Exceptions http://www.artima.com/intv/handcuffs2.html
    - Java's checked exceptions were a mistake http://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html

What to choose as a base class for custom exceptions? (2/2)

- Some general advices:
  - If a client can reasonably be expected to recover from an exception, make it a checked exception.
  - If a client cannot do anything to recover from the exception, make it an unchecked exception.
- Ask yourself:
  - How would the code for handling such exception look like?
- In the example we have chosen to use unchecked exceptions (*RuntimeException* as a base class)
  - See **07_Exceptions/…/swu/oopj/exceptions/custom/*** for details