Exceptions

Try-with-resources

Resource Management

- any external data sources (files, databases, etc.) are referred to as *resources*
- dealing with resource almost always requires three steps:
    1. opening the resource
    2. dealing with the resource (e.g. read/write)
    3. closing the resource
- forgetting to closing the resource can cause many bad things
    - *resource leak* results in resource becoming inaccessible
    - e.g. inability to connect to database by your or other programs, etc.

```
// example: method that opens a file, reads the data and closes the file
public void readFile(String file) {
  FileInputStream is = null;
  try {
     is = new FileInputStream("myfile.txt");
     // read file data
  } catch (IOException e) {
     e.printStackTrace();
  } finally {
     if (is != null) {
       try {
          is.close();
       } catch (IOException e2) {
          e2.printStackTrace();
       }
     }
  }
}
```

opening the resource

closing the resource

```
// same thing, but using try-with-resources block
// (also known as automatic resource management)


public void readFile(String file) {
  try (FileInputStream is = new FileInputStream("myfile.txt")) {
     // read file data
  } catch (IOException e) {
     e.printStackTrace();
  }
}
```

opening the resource

the resource is closed automatically

(implicit `finally` block)

```
// creating custom resource
public class MyFileClass implements AutoClosable {
  private final int num;
  public MyFileClass(int num) {
    this.num = num;
  }
  @Override
  public void close() {
    System.out.println("closing MyFileClass #" + num");
  }
}
```

interface with abstract method `close()` which has to be implemented

implementation of method `close()`

```
// using MyFileClass resource with explicit finally block
try (MyFileClass bookReader = new MyFileClass(1);
     MyFileClass movieReader = new MyFileClass(2)) {
    System.out.println("try block");
    throw new RuntimeException();
} catch (Exception e) {
    System.out.println("catch block");
} finally {
    System.out.println("finally block");
}
```

```
try block
closing MyFileClass #2
closing MyFileClass #1
catch block
finally block
```

when the exception occurs, first all the resources are closed (starting with the latest),

and only then the program continues to be executed in a regular order

Suppressed exceptions

- suppose `close()` can throw an exception, e.g.

```
public void close() throws IllegalStateExpression {
  throw new IllegalStateException("The door does not close")
}
```

- if try-with-resources block also throws an exception in catch block then
  - only the first exception will be caught
  - other exceptions will be suppressed

```
public class Door implements AutoCloseable {
  public void close() throws IllegalStateExpression {
    throw new IllegalStateException("The door does not close");
  }
}
// in main method
try (Door d = new Door()) {
   throw new IllegalStateException("Something is wrong");
} catch (IllegalStateExpression e)
    System.out.println("caught: " + e.getMessage());
     for (Throwable t : e.getSupressed())
       System.out.println("suppressed: " + t.getMessage());
}
```

second exception => suppressed

first exception => caught

this is how we can print out suppressed exceptions

```
caught: Something is wrong
suppressed: The door does not close
```