Lecture and Tutorial 17

Exception handling

- Typical problems
- try/catch; Catching Exceptions
- Checked and unchecked Exceptions
- throws, throw; Declaring and Throwing Exceptions
- (Writing own exception classes)
- Do’s and don’ts
- (Exercises next Tuesday)

A first example

```
public class Example1{
 public static void main(String[] args){

  int a = Integer.parseInt(args[0]);
  int b = Integer.parseInt(args[1]);

  int c = a / b;
  System.out.println("The quotient is "+c);
 }
}
```

What can go wrong?

Exceptions

When a Java program performs an illegal operation (division by zero, access an array at a position which does not exist, etc) an event known as exception happens. When an exception occurs, we say an exception is thrown.

Examples:

- ArithmeticException, ArrayIndexOutOfBoundsException, NumberFormatException
- IOException, FileNotFoundException, etc

Exceptions

When a Java program performs an illegal operation (division by zero, access an array at a position which does not exist, etc) an event known as exception happens. When an exception occurs, we say an exception is thrown.

Examples:

- ArithmeticException, ArrayIndexOutOfBoundsException, NumberFormatException
- IOException, FileNotFoundException, etc

Usually, when an exception occurs, the program will terminate immediately. However, Java provides ways to detect that an exception has occurred. This process is called exception handling.

Catching Exceptions

```
public class Example2 {
 public static void main(String[] args) {
  int a = Integer.parseInt(args[0]);
  int b = Integer.parseInt(args[1]);
  try {
    int quot = a / b;
    System.out.println("The quotient is "+ quot);
    }
  catch (ArithmeticException e){
    System.out.println("0 not allowed as 2nd input");
  }
 }
}
```

Catching Exceptions

- `try` block: Method calls which may cause an exception can/must be put in a `try` block.
- `catch` block: the `catch` block indicates what should happen in case an exception occurred.

Remarks:

- Variables defined in the `try` block are only local.

Better: Define and initialise the variables outside the `try` block.

- If you want that the program terminates, use `System.exit(-1)` in the `catch` block. The nonzero value -1 indicates that the program terminates abnormally.

And what is e? Or about accessing info on an exception

- When an exception occurs, Java creates an exception object which (was called e in the last example and) contains information about the error.
- Every exception object contains a string message, which can be used rather then printing your own message.

```
try {
  quot = a / b;
}
catch (ArithmeticException e){
  System.out.println(e.getMessage())
}
```

Valid inputs

Note that in the preceeding example, the Arithmetic Exception can be avoided by using a case distinction.

More difficult it will be when we deal with user input and the question whether a valid number has been entered. Here, definitely exception handling is appropriate.

```
try {
  a = Integer.parseInt(str)
}
catch (NumberFormatException e){
  // Handle exception
}
```

```
public class Example3 {
   public static void main(String[] args) {
     try {
     int a = Integer.parseInt(args[0]);
     int b = Integer.parseInt(args[1]);
     }
     catch (NumberFormatException e){
       System.out.print("NumberFormatException: ");
       System.out.println(e.getMessage());
     }
     catch (ArrayIndexOutOfBoundsException e2){
       System.out.println("Give enough arguments");
     }
   }
}
```

Or for the user’s convenience

```
public class Example4 {
 public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  while (true){
   try {
     System.out.println("Please enter number");
     String str = in.next();
     int a = Integer.parseInt(str);
     break;
   }
   catch (NumberFormatException e){
     System.out.println("Not an integer, try again");
   } ...
```

Unchecked and checked Exceptions

- By now we looked only at so-called unchecked exceptions,i.e. exceptions which can be ignored by the programmer (remember: our first program Example1 compiled with no problems). Unchecked exceptions extend the classes `RuntimeException` and `Error`.

Unchecked and checked Exceptions

- By now we looked only at so-called unchecked exceptions,i.e. exceptions which can be ignored by the programmer (remember: our first program Example1 compiled with no problems). Unchecked exceptions extend the classes `RuntimeException` and `Error`.

- Checked Exceptions are exceptions which we can't ignore; the compiler will produce an error if we fail to use a `try` block and a `catch` block to handle the exception. In other words whenever you call a method that 'throws' a checked exception, then you must tell the compiler what to do, if it ever is thrown. Typical example of a checked exception: FileNotFoundException.

Declaring exceptions: The `throws` clause

We said that checked exceptions must be caught. But that was not the full story. When ever an exception can occur inside a method we have two choices:

- Handle the exception within the method (`catch`)
- Declare that the method throws the exception.

The latter means that the method calling our method is now responsible for handling the exception.

Which technique is better? It depends on whether we can deal with an exception within a method in a meaningful way or not.

Note, in general, each method must state the types of checked exceptions it might throw.

Throwing exceptions: the `throw` statement

Finally, how to throw exceptions? How to create an exception object? Have a look at the following example in which we want to withdraw some money, however only if there is enough of it.

```
public void withdraw(int amount){
    if (amount > balance){
          IllegalArgumentException exception =
            new IllegalArgumentException
                               ("Amount exceeds balance");
          throw exception;
    }
```

or in short:

```
  throw new IllegalArgumentException(<your message>);
```

finally

Occasionally you need to take some action whether or not an exception is thrown. E.g. you have opened a file, an exception occurs and you want to close the file first before dealing with the exception.

```
FileReader reader = new FileReader(filename);
 try{
   Scanner in = new Scanner(reader);
   readData(in);
 }
 finally {
   reader.close();
 }
```

When the try block is completed, then the `finally` block will be executed. If an error occurs while reading the data, then first the `finally` block will be executed, i.e. the file is closed and then the exception is passed to its handler.

Similarly, we can catch an exception, do some action, then then ‘rethrow’ the exception.

```
try{
  <statements>
}
catch (AnException e){
  <do some actions>
  throw e;
}
```

Summary: Do’s and Don’ts

- Throwing exceptions is better than just passing a boolean flag whether an operation was successful. (The calling method simply might forget to check the flag.)

Summary: Do’s and Don’ts

- Throwing exceptions is better than just passing a boolean flag whether an operation was successful. (The calling method simply might forget to check the flag.)
- Do throw specific Exceptions (and not just a RunTimeException if you mean something more specific).

Summary: Do's and Don'ts

- Throwing exceptions is better than just passing a boolean flag whether an operation was successful. (The calling method simply might forget to check the flag.)
- Do throw specific Exceptions (and not just a RunTimeException if you mean something more specific).
- Throw early, Catch late. (It is better to declare that a method throws a checked exception than to handle the exception poorly.)

Summary: Do’s and Don’ts

- Throwing exceptions is better than just passing a boolean flag whether an operation was successful. (The calling method simply might forget to check the flag.)
- Do throw specific Exceptions (and not just a RunTimeException if you mean something more specific).
- Throw early, Catch late. (It is better to declare that a method throws a checked exception than to handle the exception poorly.)

- Don’t use Error handling instead of ordinary control structures (for cases which can be forseen).

Summary: Do’s and Don’ts

- Throwing exceptions is better than just passing a boolean flag whether an operation was successful. (The calling method simply might forget to check the flag.)
- Do throw specific Exceptions (and not just a RunTimeException if you mean something more specific).
- Throw early, Catch late. (It is better to declare that a method throws a checked exception than to handle the exception poorly.)

- Don’t use Error handling instead of ordinary control structures (for cases which can be forseen).
- And of course, do not forget to handle the thrown exceptions.