MSc/ICY Software Workshop
Exception Handling, Assertions
Scanner, Patterns
File Input/Output

Manfred Kerber www.cs.bham.ac.uk/~mmk

Tuesday 22 October 2019

Classes and Objects

The information we have about a particular object is encapsulated in so-called field variables. First, we have to clarify which ones that should be.
In order to create and manipulate objects we always have:

- At least one constructor (for the creation of objects)
- getters are methods to get the components of objects back.
- setters are methods to change components of objects.
- The `toString()` method is used when the object is to be printed. Without it, an object is not printed in a human readable way.
- In order to check two objects for equality we can write a method `equals`.

Multiple Constructors

You may construct objects (as characterized by the field variables) using constructors with different number of arguments (or different types in the arguments).
E.g.,

```
public BankAccount(int accountNumber, String  accountName)
    this.accountNumber      = accountNumber;
    this.accountName        = accountName;
    this.balance            = 0;
}

public BankAccount(int accountNumber,
                   String  accountName,
                   int balance) {
    this.accountNumber      = accountNumber;
    this.accountName        = accountName;
    this.balance            = balance;
}
```


Problems with User Input

How to deal with problems of input?

Not under control of the programmer

```
System.our.println("Provide n, m with m != 0");
n = Integer.parseInt(args[0]);
m = Integer.parseInt(args[1]);
System.out.println("n/m: " + (n/m));
```


Exceptions

Exceptions are used to deal with errors

```
System.our.println("Provide n, m with m != 0");
try {
    n = Integer.parseInt(args[0]);
    m = Integer.parseInt(args[1]);
    System.out.println("n/m: " + (n/m));
}
catch (IllegalArgumentException e) {
    // By "catch" we say what should happen
    // if the error occurs.
    System.out.println("Oops. Do not divide by zero");
}
```


Exceptions (Cont'd)

```
System.out.println("Provide n, m with m != 0");
try {
    n = Integer.parseInt(args[0]);
    m = Integer.parseInt(args[1]);
    System.out.println("n/m: " + (n/m));
}
catch (NumberFormatException e) {
    System.out.println("Oops. Numbers of type int expected!");
catch (IllegalArgumentException e) {
    System.out.println("Oops. Do not divide by zero!");
}
```


Exceptions in general

```
try { some code which may throw an exception
      of type ExceptionType
}
catch (ExceptionType e) {
   code executed if exception e of ExceptionType occured
}
```


Exceptions in general (Cont'd)

```
try {some code which may throw an exception e of type
     ExceptionType1 or ExceptionType2 or ExceptionType
}
catch (ExceptionType1 e) {
   code executed if exception e of ExceptionType1 occured
}
catch (ExceptionType2 e) {
   code executed if exception e of ExceptionType2 occured
}
catch (ExceptionType3 e) {
   code executed if exception e of ExceptionType3 occured
}
```

Exceptions and finally

```
try {some code which may throw an exception
   of type ExceptionType
}
catch (ExceptionType e) {
   code executed if exception e of ExceptionType occured
}
finally {
   some more code executed of whether the try or the
   catch part is executed.
}
```

Make sure that code in catch and finally never crashes!

Checked vs Unchecked Exceptions

- Unchecked Exceptions may or may not be caught by the program.
  They deal typically with problems that are under control of the programmer (e.g., an `ArrayIndexOutOfBoundsException`)
- Checked Exceptions must be caught by the program. These deal typically with problems that are NOT under control of the programmer (e.g. whether a file exists or is accessible, `FileNotFoundException` or `AccessDeniedException`).
  The Java compiler enforces a catch statement for a checked exception.

Scanner for Input

```
String str;
int n;
double d;
// creates a new scanner object, reads from the terminal
Scanner s = new Scanner(System.in);

//  reads next word of input (delimited by white spaces).
str = s.next();

// reads next integer. Exception if next word not int
n = s.nextInt();

// reads next double. Exception if next word not double
d = s.nextDouble();
```


Patterns

```
// any number of a followed by a single b
Pattern p1 = Pattern.compile("a*b");

// any number of a,b,c in any order
Pattern p2 = Pattern.compile("[abc]*");

// any number of letters
Pattern p3 = Pattern.compile("[a-zA-Z0-9]*");

// any number of letters followed by a single @,
// followed by any number of letters.
Pattern p4 = Pattern.compile("[a-zA-Z.]*@[a-zA-Z.]*");
```

For a full description see java/util/regex/Pattern.html.

Pattern to Restrict Input for Scanner

```
// either 1, or 2, or 3.
Pattern p = Pattern.compile("[1-3]");

int n;
Scanner s = new Scanner(System.in);

/*  reads next word which must correspond
 *  to either 1, or 2, or 3.
 */
n = s.nextInt(p);
```


Reading from/Writing to File

```
import java.io.*;

public static void main(String[] args) {
    try {
        String readString,writeString;
        BufferedReader in =
            new BufferedReader(new FileReader("test1.in"));
        BufferedWriter out =
            new BufferedWriter(new FileWriter("test1.out"));
        int counter = 0;
        while ((readString = in.readLine()) != null) {
            System.out.println(readString);
            counter++;
            out.write(counter + " " + readString + "\n");
        }
        // in.close();
        // out.close();
    }
    catch (IOException e) {
        System.out.println("File not found.");
    }
}
```


Reading from a Web page

```
import java.io.*;
import java.net.URL;
String s = "https://birmingham.instructure.com/courses/38428";
try {
    URL url = new URL(s);
    BufferedReader in =
        new BufferedReader(new
            InputStreamReader(url.openStream()));
    BufferedWriter out =
        new BufferedWriter(new FileWriter("test1.html"));
...
}
catch (IOException e) {
    System.out.println("no access to URL: " + s);
}
```


Throwing Exceptions

```
public static boolean estimateInBounds(double actual,
                                       double nominal) {

...
    if (nominal < 5 || nominal > 10000) {
        throw new IllegalArgumentException();
    } else {
        return
            (absShortFall <= 0                          ||
            (5 < nominal    && nominal <= 50
                            && relShortFall <= 0.09)    ||
             ...) ;

    }
}
```


Class Invariants

In classes the implementer may want to enforce that certain field variables can take values only in a restricted form, e.g., for a variable `private String months` not every value may be allowed, but only one of `"January"`, ..., `"December"`.
Likewise that a variable `private String gender` takes only the values `"m"`, `"f"`, or `"x"`.
If this is always the case then this is called a Class Invariant. The program and the programmer can rely on the fact that a month is always one of twelf given strings.
This can be achieved by throwing an exception whenever with a constructor or a setter it is tried to give the variable a value that is not allowed.

Assertions

Assertions are used to establish that properties we are certain that they hold at a particular point actually do hold. If not an exception will be raised – assumed the compiler is correspondingly configured (by -ea option in 'Run Configurations' and '(x)= Arguments' under 'VM Arguments' in Eclipse). Good for debugging.

```
public class AssertExample {
    public static void main(String[] args) {
        int x = -5;
        x = Math.abs(x);
        assert x >= 0;
        System.out.println(Math.sqrt(x));
    }
}
```

