Object-Oriented Analysis & Java

Lecture 4

Florentin Bekier

Packages

Packages & API

- In Java, a package is used to group related classes
  - Like a folder
- Used to avoid name conflicts
- Write a better maintainable code
- 2 categories:
  - Built-in packages (Java API)
  - User-defined packages

Using a class from a package

- 3 possibilities:
  - Make a reference to the class with the package name:
    ```
    java.util.Date date = new java.util.Date();
    ```
  - Import the class:
    ```
    import java.util.Date;
    ```
  - Import the whole package:
    ```
    import java.util.*;
    ```

Static import

- You can use the keyword `static` to import a package
- Allows to reduce the code when using static methods or attributes
- Needs to be used with care because it may be confusing

```
import static java.lang.Math.*;
int x = max(sqrt(abs(y)), sin(y)); // Instead of
Math.max...
```

User-defined packages

- Packages are stored like folders:

```
root
└── mypack
    └── MyPackageClass.java
```

- To create a package, use the keyword `package`

```
package mypack;

class MyPackageClass {
}
```

Input/output

Output

- `System.out` is the default output stream (terminal)
- 3 functions:
  - `System.out.print()`: print the string passed as argument
  - `System.out.println()`: print the string passed as argument and move to the next line
  - `System.out.printf()`: provide string formatting (like C/C++)

String formatting

```
System.out.printf("format-string" [arg1, arg2, …]);
String.format("format-string" [arg1, arg2, …]);
```

- Format specifiers: % [flags] [width] [.precision] conversion-character

String formatting

Flags:

- `-` : left-justify (default is to right-justify)
- `+` : output a plus or minus sign for a numerical value
- `0` : forces numerical values to be zero-padded (default is blank padding)
- `,` : comma grouping separator (for numbers >1000)
- ` ` : space will display a minus sign if the number is negative or a space if it is positive

String formatting

Width:

- Specifies the field width for outputting the argument and represents the minimum number of characters to be written to the output

Precision:

- Specifies the number of digits of precision when outputting floating-point values or the length of a substring to extract from a String

String formatting

Conversion-Characters:

- d : Decimal integer (byte, short, int or long)
- f : Floating-point number (float or double)
- c : Character (capital C will uppercase the letter)
- s : String (capital S will uppercase all the letters in the string)
- h : Hashcode (this is useful for printing a reference)
- n : Newline (platform specific)

String formatting (example)

```
System.out.printf("Total is: $%,.2f%n", dblTotal);
System.out.printf("Total: %-10.2f: ", dblTotal);
System.out.printf("% 4d", intValue);
System.out.printf("%20.10s\n", stringVal);
String s = "Hello World";
System.out.printf("The String object %s is at hash
code %h%n", s, s);
```

Input

- `System.in` is the default input stream (keyboard)
- You can get the value of the input with the class `Scanner` (package `java.util`)
- Create with: `Scanner sc = new Scanner(System.in);`
- Get the input via the methods: `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `next()`, ...

Exceptions

Error handling

- Several errors can occur when your program is running: coding errors, errors due to wrong input, etc.
- To handle those errors, Java will throw an **exception** (throw an error)

Try/catch

- The `try` statement allows you to define a block of code to be tested for errors while it is being executed
- The `catch` statement allows you to define a block of code to be executed, if an error occurs in the `try` block
- The `try` and `catch` keywords come in pairs and are used to catch and handle errors

Try/catch (example)

```
try {
    //  Block of code to try
}
catch (Exception e) {
    //  Block of code to handle errors
}
```

Finally

- The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

```
try {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // Error!
} catch (Exception e) {
    System.out.println("Something went wrong.");
} finally {
    System.out.println("The 'try catch' is finished.");
}
```

Throwing exceptions

- The `throw` statement allows you to create a custom error
- Used together with an **exception type** (*e.g.*, `ArithmeticException`, `FileNotFoundException`, `SecurityException`, `ArrayIndexOutOfBoundsException`, etc.)
- A method has to declare the type of exception it throws with the keyword `throws`

Throwing exceptions (example)

```
public void checkAge(int age) throws ArithmeticException {
    if (age < 0) {
        throw new ArithmeticException("Your age has to be a
    positive number.");
    }
}
```

Checked and unchecked exceptions

- Checked exceptions are exceptions that need to be treated explicitly (they extend `Exception`)
- Unchecked exceptions don’t need to be treated explicitly (they extend `RuntimeException`)
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - …

Custom exceptions

- To create a custom exception, we create a class that extends the `Exception` or `RuntimeException` class

```
public class OutOfFuelException extends Exception {
    public OutOfFuelException(String errorMessage) {
        super(errorMessage);
    }
}
```