6. Object Oriented Programming (OOP)

Prof. Dr. Charles Wüthrich
B.Sc. Francesco Andreussi,
CoGVis/MMC, Faculty of Media
Bauhaus-University Weimar

Introduction

- Modern programming lang. allow programmers to
  - organize their data
  - Organize the way to manipulate it in a structured way.
- Like the name says, Object Orientation centers programming on *objects*.
- Objects in Java are instances of *classes*:
  - A class is the blueprint for the objects
  - Objects are the actual things built through these blueprints.
- Sometimes terminology is a little different, but the concept is the same

Data Types

- In OOP, a Data Type is a collection of
  - Data (variables and the like) → *attributes*
  - Procedures to manipulate the data → *methods*
- In Java, these are declared in a *class*.
- Example: The `String` class (extract of real class)

public class String		
	String(String s)	*constructor*
int	length()	*n^o of characters*
char	chatAt(int i)	*character at index i*
String	substring(int i, int j)	*substring from indices i to j*
boolean	contains(String substring)	*is substring contained in string?*
boolean	startsWith(String pre)	*does the string start with pre?*
boolean	endsWith(String post)	*does the string end with post?*
String	concat(String t)	*Appends string t at end of string*
int	compareTo(String t)	*compares string t to string*
String	toLowerCase()	*Converts string to lower case*
String	toUpperCase()	*Converts string to all caps*

Data Types

public class String

	String(String s)	*constructor*
int	length()	*n° of characters*
char	chatAt(int i)	*character at index i*
String	substring(int i, int j)	*substring from indices i to j*
boolean	contains(String substring)	*is substring contained in string?*
boolean	startsWith(String pre)	*does the string start with pre?*
boolean	endsWith(String post)	*does the string end with post?*
String	concat(String t)	*Appends string t at end of string*
int	compareTo(String t)	*compares string t to string*
String	toLowerCase()	*Converts string to lower case*
String	toUpperCase()	*Converts string to all caps*

- 1st element: *constructor*
  - Has same name of the class and no data type
  - There can be more than one
  - When one declares an object, with `String astring;`, Java just notices the blueprint it has to use for creating the object
- The constructor is used when `new` statement appears:

```
astring = new String("Artists love Java!);
```

  - Only then the instance of object is created: `new` calls constructor

Data Types

public class String

	String(String s)	*constructor*
int	length()	*n° of characters*
char	chatAt(int i)	*character at index i*
String	substring(int i, int j)	*substring from indices i to j*
boolean	contains(String substring)	*is substring contained in string?*
boolean	startsWith(String pre)	*does the string start with pre?*
boolean	endsWith(String post)	*does the string end with post?*
String	concat(String t)	*Appends string t at end of string*
int	compareTo(String t)	*compares string t to string*
String	toLowerCase()	*Converts string to lower case*
String	toUpperCase()	*Converts string to all caps*

- Other elements: *methods* to manipulate the object
  - Once an instance of object is created, one can use the methods in the API described above
- Example:

```
astring = astring.toLowerCase();
```

  will convert all caps letters of `astring` to lower case converting the string to the string "`artists love java`"
- Java has other built in types, for example `Color`

```
red = new Color(255,0,0);
```

Object characteristics

- Some characteristics of Data Types:
  - *Aliasing*: when `object1` is assigned to `object2` by writing
    ```
    object1=object2;
    ```
    Java just copies the reference, not the object.
    → modifying things in `object2` is the same as in `object1` because the references are the same.

  - *Pass by value*: If a method is called with arguments, Java makes a copy of the arguments.
    → Modifying them does not modify anything in the calling class
  - *Arrays*: Arrays are normal objects + special notation to facilitate index computations, like the use of `[]`
    → when you pass an array, you pass its reference!
    → your can modify array elements in a method and see changes when the method execution ends

Object characteristics

- Other characteristics of Data Types:
  - *Arrays of objects*: Creating an array of objects is a 2 step process
    - Array of references to the single objects
    - Initialize each object through `new`.
    - For example:
      ```
      colorvec = new Color[2];
      colorvec[0] = new Color(0,0,255);
      colorvec[1] = new Color(0,255,0);
      ```
      creates an array of 2 `Color` objects.
      The array is just references
  - *Safe pointers*: Java references are called safe because there is only one way to create them, through new.
    → you cannot manipulate them except through assignments
    → no operations on references!

Object characteristics

- Other characteristics of Data Types:
  - *Orphaned objects*: Objects might get lost!
    - For example:
      ```
      color1 = new Color(0,0,255);
      color2 = new Color(0,255,0);
      color2 = color1;
      ```
      After last statement, the reference to `color2` is LOST
      → the data with no reference is called *orphaned object*
  - *Garbage collection*: Java is capable of tracking orphaned memory, collect it and make it available again.
    This operation is called garbage collection

Creating Data Types

- What are the issues when creating Data Types?
  - Lots to be considered
  - Not simple: lots of planning needed
- Suppose we want to create a class representing a light we want to turn on and off:
- Looks like a simple task…..
- What do we need for it?

Creating Data Types

- First place to start from:
  Create an API for the class!
- What do we need?
  - Constructor
  - Lamps are either ON or OFF:
    In the class I need a boolean variable storing this → true=ON, false=OFF
  - I need to turn on the light → method! Returns nothing!
  - I need to turn off the light → method! Returns nothing!
- The partial API will then look like this:

public class Light		
	Light()	*constructor*
void	switchOn()	*switch on light*
void	switchOff()	*switch off light*

Creating Data Types

- Is this all?
  - Of course NOT: how do I find out if the light is ON or OFF?
  - Obviously we must enquire its state:
    we need an isOn method which returns a boolean:
    - `true` if ON
    - `false` if OFF
- So the API looks like:

`public class Light`		
	`Light()`	*constructor*
`void`	`switchOn()`	*switch on light*
`void`	`switchOff()`	*switch off light*
`bool`	`isOn()`	*am I on?*

- We can start from here…

Creating Data Types

- Now we need to ask ourself a few questions:
  - Do we allow other programmers to access our variables?
    - *Public* members are accessible to everyone.
    - Private members are accessible only to class members, but NOT to external clients.

	class	package	subclass same pkg	subclass diff pkg	world
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

- Table show member access rules: relevant for now
  - class and world
  - public and private
- One more attibute of variable is important: `final`
  A `final` variable cannot be modified, but it can assigned only ONCE

Creating Data Types

- *Instance variables* are variables that are declared right after the class declaration, not in `main`.
- They are accessible to all members of the class.
- For our `Light` class, we need only one boolean variable: on.
  - We protect this variable so that no one is allowed to access it except through the methods of the class `Light`:

```
public class Light{
    private boolean on;

    ... methods
    }
```

Creating Data Types

- Once the variables are defined, we can implement our methods to manipulate classes:
- We need a *constructor*, which is called when `new` creates a new object:

```
public Light() {
    on=false;
    }
```

- Notice that it sets the `on` variable to `false`: lamp is switched off when it is created.

Creating Data Types

- Then we need two methods for turning on and off the light.
- Moreover, we need a method to check if the light is on or off:

```
public void switchOn() {
    on=true;
    }

public void switchOff() {
    on=false;
    }

public boolean isOn() {
    return on;
    }
```

- Notice that these methods are `public`, so they provide the means for external clients to work with the class variable.

Creating Data Types

- To use the methods in the class on an object, once an object is instantiated through

```
Light light1 = new Light();
```

  we can address its methods like this:

```
light1.switchOn();
light1.isOn();
```

- Just remember what types these methods return, which is defined in the API
- Up to now, we saw different types of variables:
  - *parameter values,* used to pass parameters to a method
  - *local variables* used within a method
  - *instance variables*, which hold data type values for a class.
- So `light1.isOn()` queries the `on` variable of the instance named `light1`

Creating Data Types

- Finally we need a class to test our LightClass:

```
class LampTest() {
    public void main(String[] args){
      Light light1, light2;

      light1= new Light();
      light2= new Light();
      light1.switchOn;
      light2.switchOff;

      System.out.println("light1 is on?" + light1.isOn() +
                         " light2 is on?" + light2.isOn());
      }
    }
```

Creating Data Types

- Notice that the test class is defined without attributes:
  - This because we put it in the same file like this:

```
public class Light{
   private boolean on;

   public Light() {
      on=false;
   } //constructor

   public void switchOn() {
      on=true;
   }

   public void switchOff() {
      on=false;
   }

   public boolean isOn(){
      return on;
   }
}
```

```
class LampTest{
    public static void main(String[] args){
      Light light1, light2;

      light1= new Light();
      light2= new Light();
      light1.switchOn();
      light2.switchOff();

      System.out.println("light1 is on? " + light1.isOn()
                  + " -- light2 is on? " + light2.isOn());
    }
}
```

Creating Data Types

- If we want to make `LampTest` externally available, we need to put it in a separate file: two classes can't be defined in a single file

```
public class Light2{
   private boolean on;

   public Light2() {
      on=false;
   } //constructor

   public void switchOn() {
      on=true;
   }

   public void switchOff() {
      on=false;
   }

   public boolean isOn(){
      return on;
   }
}
```

File `Light2.java`

```
public class LampTest2{
    public static void main(String[] args){
      Light light1, light2;

      light1= new Light();
      light2= new Light();
      light1.switchOn();
      light2.switchOff();

      System.out.println("light1 is on? " + light1.isOn() +
                         " -- light2 is on? " + light2.isOn());
    }
}
```

File `LampTest2.java`

Creating Data Types

- Running our LightTest program results in the following output:

```
> run LampTest
light1 is on? true - light2 is on? false
```

- We have implemented our first ***thought through*** class!

Designing Data Types

- Data and methods are the *members* of a class
- Before starting to program, one needs to know
  - What should the class do
  - Which of its members are available to clients: this is called *encapsulation*, and ruled through private and public attributes

- Final variables, for example:

```
public final double PI = 3.141592653589793;
```

  are assigned only ONCE and then are fixed
- `static` members are associated to the class, not its instances: if I create a new object, they are not duplicated

```
public static final double PI=3.141592653589793;
```

Designing Data Types

- `this:`
  within an instance method refers to the current object
- Example:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Designing Data Types

- `interface:`
  Allows to set logical relations to things which are otherwise unrelated, and provides templates for similar methods:
- Example: Suppose you have classes representing geometric entities, like circles, squares, lines and points
- They are similar, in that you can operate same things on them, like dragging them around, or moving them

Designing Data Types

- Let us define a common interface `Draggable` for them:

```
public interface Draggable{
  public abstract void moveTo (int x, int y);
  ...
}
```

- You can explicitly say your class implements the interface like this:

```
public class Circle implements Draggable{
  // whatever a Circle is made of
  void moveTo (int x, int y){
    ...
  }
  ...
}
```

Designing Data Types

- You can derive classes from other classes, creating subclasses that inherit the variable and methods of their superclass
- For example, we can extend the Light class by adding its position:

```
public class LightPositioned extends Light {
  public int x_position,y_position;
  ....
}
```

- The subclass inherits both *public* and *protected* members of its parent class.

Designing Data Types

- In theory, one could implement a whole hierarchy of classes, the structure of which looks like a genealogic tree, with *super*- and *sub*-classes.
- In fact, in Java every class is a subclass of Java.lang.Object, which is basically the root of the genealogic tree of classes.

Designing Data Types

- A *Java package* is a collection of classes, interfaces and subpackages.
- You can define your own packages and make them available, by compiling them as

```
javac -d DestinationFolder packagename.java
```

- This creates a new folder called DestinationFolder and will put the compiled class in this folder.

Designing Data Types

- From packages, you can import classes and methods through the import statement, which you have to put at the beginning:

```
package letmecalculate;

public class Calculator {
  public int add(int a, int b){
  return a+b;
  }
  public static void main(String args[]){
  Calculator obj = new Calculator();
  System.out.println(obj.add(10, 20));
  }
}
```

```
import letmecalculate.Calculator;

public class Demo{
  public static void main(String args[]){
  Calculator obj = new Calculator();
  System.out.println(obj.add(100, 200));
  }
}
```

- If a package is not declared in a class, then it will belong to the current package (current directory)

End

+++ Ende - The end - Finis - Fin - Fine +++ Ende - The end - Finis - Fin - Fine +++