Objects and Classes

Java version

Objectives

To learn about Java classes and objects.

Main concepts discussed in this chapter

- objects
- classes
- methods
- parameters
- packages

Background

Resources

- Classes needed for this lab – in the Java archive file *chapter01.jar*.
- Slides from the course.
- Material covered: Objects and Classes.

To do

Getting the resources

- Download *chapter01.jar* into a lab-specific folder

  ```
  path/to/courses
    +--oop
         +--objects_and_classes
              +--chapter01.jar
  ```
  where **`path/to/courses`** just indicates where you are putting your work (*oop* is your folder for this course, *objects_and_classes* is your folder for this lab).

- Go to the lab folder

  ```
  cd path/to/courses/oop/objects_and_classes
  ```

- Unarchive the code

  ```
  jar -xvf chapter01.jar
  ```

- Open your favourite IDE on the lab folder by creating a new project or whatever you have to do. The code should be visible.

First lab session

Well ok, this is the obligatory first application, and yes it's pretty trivial. The code is in the package *nihaoshijie*.

Open *Main.java* in your favourite editor / development tool to display the source code for the application. This should look something like:

```
package nihaoshijie;
/**
 * The traditional start point for learning any (computer) language. Note that
 * this class does not have to be named Main. But to be executable by the
 * JVM it must have a method main, as below.
 *
 * The follwing tag is obligatory for all your classes to indicate who owns
 * the code.
 * @author <a href="mailto:sander@univ-cotedazur.fr">Peter Sander</a>
 * If you modify existing code, add another @author tag with your name, eg,
 * @author Mickey Moose
 */
class Main {
    /**
     * Entry point for the JVM to begin executing code. Note that this method
     * must have exactly this signature.
     *
     * @param args Unused.
     */
    public static void main(String[] args) {
        System.out.println("Ni hao shijie");
    }
}
```

Congratulations – your first Java application! (Well, Ok, you didn't write it...)

Unfortunately, it doesn't really do very much: it just *invokes a method* **`println`** *on an object* **`System.out`**, and passes the method a character string argument **`"Ni hao shijie"`**. To advance a bit, we'll introduce another class, called **`NiHaoShiejie`** into the same folder as **`Main`**:

```
package nihaoshijie;

/**
 * Class to create an object to print a message.
 *
 * @author <a href="mailto:sander@univ-cotedazur.fr">Peter Sander</a>
 */
class NiHaoShijie {
    private String message = "Ni hao shijie";

    /**
     * Must use the variable message.
     * @return the character string: "from NiHaoShijie: Ni hao shijie"
     */
    public String toString() {
        return "from NiHaoShijie: " + message;
    }

    /**
     * Prints the character string returned by the method toString
     */
    void print() {
        System.out.println(toString());
    }
}
```

Exercises

I. Try to figure out what is happening inside the class **`NiHaoShijie`**: what are the attributes? what are the methods?

II. Can you execute this class?

III. Create a new object inside **`Main`** by replacing the code in the **`main`** method with the following:

```
NiHaoShijie messageObject = new NiHaoShijie();
messageObject.print();
```

IV. What happens when you now execute the **`Main`** class? Can you explain what happens?

V. Can you run the code both from the command line and from whatever tool you're using?

Going full-paranoid

A very important bit of Java orthodoxy is *need to know* - give away as little information as possible (this is also known as *encapsulation* or *information-hiding*). In particular, we'll make a difference between *us* and *them*.

Us means members of my development group. These are the good guys, they can generally be trusted. My development group participates in development of our product code. Normally the development group works on code in specific packages to which they need privileged access.

In the first example above, both classes **`NiHaoShijie`** and **`Main`** were in the package **`nihaoshijie`**, ie, the declaration for **`NiHaoShijie`** was

```
package nihaoshijie;

class NiHaoShijie
```

The class declaration has no preceding access keyword because it has *default*-level access. This is also known as *package-private* (or sometimes just package) access. This means that the code in **`NiHaoShijie`** is only accessible from code also in the **`nihaoshijie`** package, like the class **`Main`**.

Them refers to everybody else. These are people we cannot trust because they can break our code, either through incompetence, evil intentions, whatever. The less of our code they have access to, the less damage they can cause. They should never have access to package-private code because they should never be developing anything in our packages. They are users of our packages, not developers in our packages.

However, they do need to be able to use our code. To illustrate this, we'll make some changes to our example.

Exercises

VI. Create a new folder *src/nihaoshijie/client* and move the file *Main.java* into that folder. Replace the package statement in the class **`Main`** by **`nihaoshijie.client`**. :

```
package nihaoshijie.client;

class Main {
    public static void main(String[] args {
        NiHaoShijie messageObject = new NiHaoShijie();
             messageObject.print();
    }
}
```

This now represents client code which is going to use our developer code in the class **`NiHaoShijie`**, which remains in the package **`nihaoshijie`**. Does the **`Main`** class still execute? Does it even compile?

The first problem is that **`Main`** uses a class **`NiHaoShijie`** which is inside a different package. We have to give the compiler some help in finding **`NiHaoShijie`** – modify the declaration of **`Main`** as follows:

```
package nihaoshijie.client;

import nihaoshijie.NiHaoShijie;  // look for NiHaoShijie in package
                                 //  nihaoshijie

class Main {
    // no other changes
}
```

VII. Oops again – since we've moved the **`Main`** class into another package, its code has lost access to the package-private code in the original package. If we want our client to be able to use our awesome code in **`NiHaoShiJie`**, we have to open the access to allow them to do so.

Fix the classes so they compile and run by changing the class declaration to

```
package nihaoshijie;

public class NiHaoShijie {
    // no other changes
}
```

Does the **`Main`** class execute now? Does it compile now? Looks like we still have a problem.

We have seen that we can limit access to a class, but we can be more selective. Inside **`NiHaoShijie`**, we declared

```
void print() {
    System.out.println(toString());
}
```

There is no explicit keyword for the **`print`** method so it gets default access, which is again package-private. It can only be used within the **`nihaoshijie`** package. Thus when **`Main`**, which is now in a different package, tries to access it with **`messageObject.print`**, the compiler says no.

Exercise

VIII. Change the access level for the method

```
    public void print() {
        System.out.println(toString());
    }
```

and try to compile and execute again.

The following table (from the Java documentation[1]) summarizes the access control keywords for attributes and methods:

Access Levels

Modifier	Class	Package	World
`public`	Y	Y	Y
default	Y	Y	N
`private`	Y	N	N

`private` access is even more restrictive – it is accessible only from within the same class. Sometimes we don't even trust our fellow-developers!

Accessibility guidelines

- Declare classes to have package-private access (default - no keyword).
    - Only if necessary for use outside the dev group increase access to **`public`**.
- Declare attributes and methods to have **`private`** access.
    - Only if necessary for use by the dev group increase access to package-private (default – no keyword).
    - Only if necessary for use outside of the dev group increase access to **`public`**.

From now on we're going to separate code intended for *us* (developers) from code intended for *them* (clients).

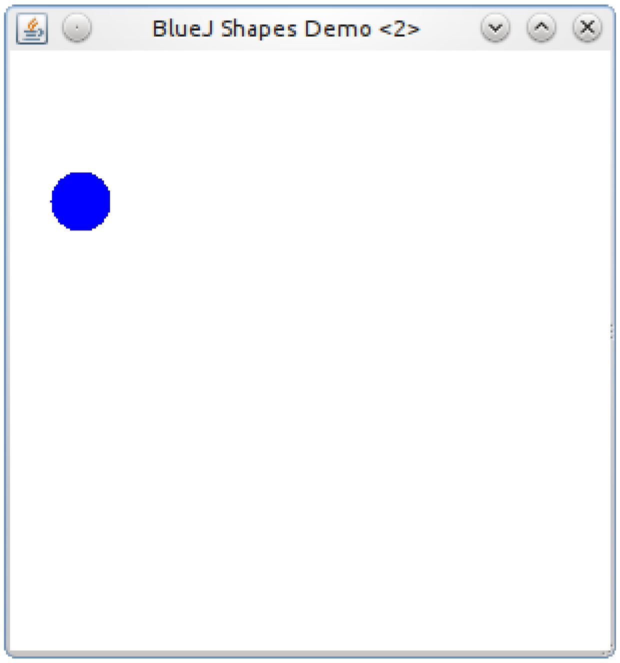
Doing more

A more interesting application is in the *figures* package[2].

Executing this application should bring up a window like:

1 https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

2 Note that this application uses the Swing GUI (Graphical User Interface). Swing is no longer the recommended GUI for Java applications. In fact, GUIs in general are not really used in Java any more, having been mostly replaced by HTML5 + JavaScript. This example is just to illustrate using objects.



Here's the code for the class (again called **`Main`**) containing the **`main`** method:

```
package figures;

class Main {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.makeVisible();
    }
}
```

The code

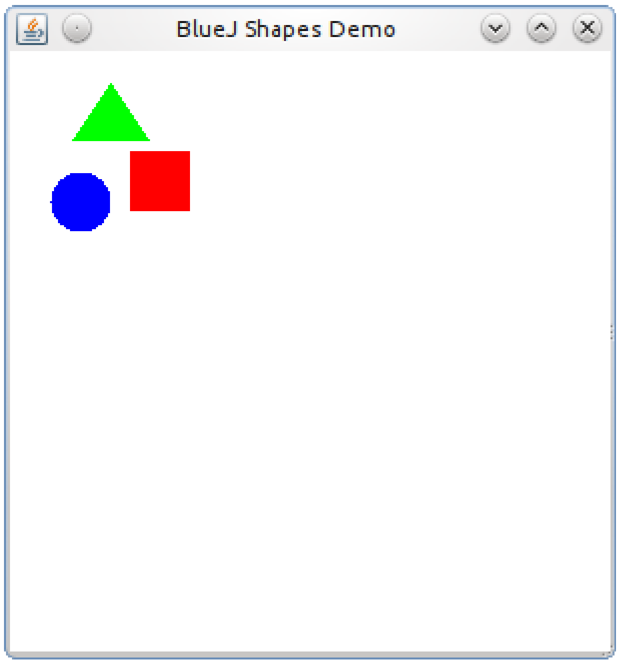
```
    Circle circle = new Circle();
    circle.makeVisible();
```

creates a new object of type **`Circle`** and assigns it to the variable **`circle`** of type **`Circle`**.

Exercise

1. Edit the code to create a second **`Circle`** object (you'll have to give the variable a different name), then create a **`Square`** object and a **`Triangle`** object. Run the code as above. Is the result as expected?

   You now have a Java application consisting of four objects: two circles, a square and a triangle:



Oops - why don't you see two circles?

Invoking object methods

You make objects do things by modifying the code for their classes, in particular by modifying and adding methods to the classes.

Exercise

2. We're going to invoke the **moveDown** method on the **circle1** object to try to figure out why we can't see it. Modify your code to look like:

```
Circle circle1 = new Circle();
circle1.moveDown();
circle1.makeVisible();
```

Aha! What happens if you call **moveDown** twice? Or three times? What happens if you call **makeInvisible** after **makeVisible**? What does calling **makeInvisible** twice change? What other moving methods are available?

Calling methods with parameters

For more flexibility, you can give additional information to a method to tell it how to do its job by passing it a parameter or passing it an argument. Instead of **moveDown**, invoke the **moveHorizontal** method on **circle1** by modifying the code as follows:

```
Circle circle1 = new Circle();
circle1.moveHorizontal(100);
circle1.makeVisible();
```

The argument to the method is the number of pixels to move the object horizontally across the screen.

Exercise

3. Try invoking the **`moveVertical`** and **`changeSize`** methods. Find out how you can use **`moveHorizontal`** to move **`circle1`** 20 pixels to the left.

Data types

Now let's open up another of the classes and see what's inside. Open **`Circle.java`** and look at the code. The line **`void moveHorizontal(int distance)`** indicates the *signature* of the method **`moveHorizontal`** and gives the following information

- **`moveHorizontal`** - name of the method;
- **`(int distance)`** - the number of parameters (1), type (**`int`**) and name of the parameter (**`distance`**);
- **`void`** - the return value of the method (we'll see this later).

The type of the parameter

`int` (integer) is the only type of value that is acceptable as an argument to **`moveHorizontal`**. Try setting something which is not an integer as the parameter, eg, **`123.45`**, **`toto`**, or **`"toto"`**.

Exercises

4. Invoke **`circle1`**'s **`changeColor`** method with the character string **`"red"`** as argument (including the quotes). The method signature is

   ```
   void changeColor(String newColor)
   ```
   indicating that it takes a single parameter of type **`String`**, meaning a character string enclosed in double quotes.
5. Try changing other objects to other colours. Does it work for all colours?
6. Try leaving out the double quotes. What happens?

We will encounter other Java types besides **`int`** and **`String`** later.

Multiple instances

Many objects can be created from a single class, eg, you have already created the objects **`circle`** and **`circle1`** from the **`Circle`** class.

Exercise

7. Create several circle objects. Make them visible, then move them around on the screen using the move methods. Make one big and yellow, make another one small and green. Try the other shapes too: create a

few triangles and squares. Change their positions, sizes, and colours.

State

An object's state is determined by the values assigned to its instance variables.

1. Create two objects **`triangle1`** and **`triangle2`** from the **`Triangle`** class and make them both visible.
2. Move **`triangle2`** 50 pixels to the right.
3. What are the differences between the objects **`triangle1`** and **`triangle2`**?
    - Each object has its own state, usually with different values. Even though both triangle objects were created from the same class, they are at different positions on the screen so their **`xPosition`** states are different.
4. Move **`triangle1`** down slowly by 100 pixels. What happens to its state after the move?
5. Change its colour. What happens to its state?

Exercise

8. Make sure you have several objects. How can you determine the state of an object?

The attributes of each object are also called *fields* or *instance varibles* because their values can change for each object (each instance of a class).

What is an object?

Verify that:

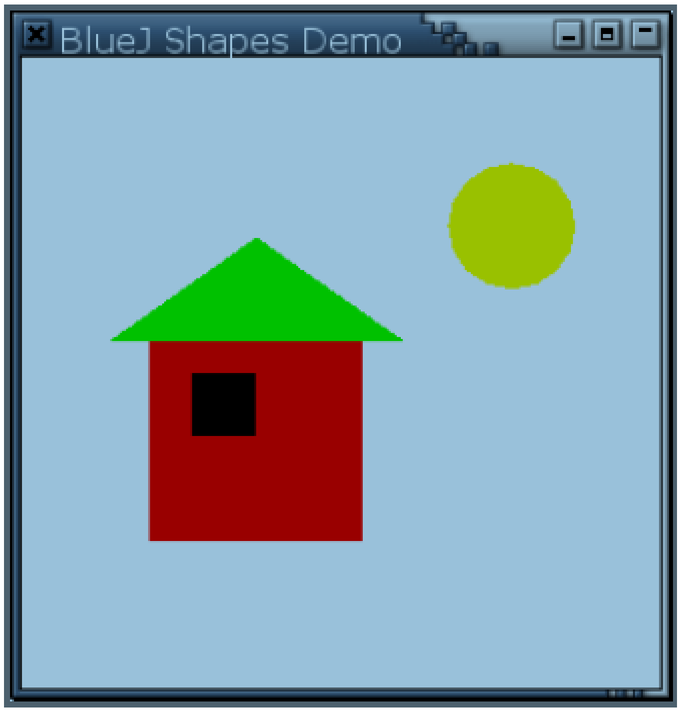
- different instances of the same class have exactly the same fields, but the fields may have different values for each instance;
- objects of different type, eg, **`circle1`** of type **`Circle`** and **`square1`** of type **`Square`**, have different fields.

Class	**Objects**	
`Square`	**`square1`**	**`square2`**
`size`	**`100`**	**`30`**
`xPosition`	**`60`**	**`80`**
`yPosition`	**`130`**	**`150`**
`color`	**`"red"`**	**`"black"`**
`isVisible`	**`true`**	**`true`**

Class	Objects

`Circle`	`circle1`
`diameter`	`60`
`xPosition`	`200`
`yPosition`	`50`
`color`	`"yellow"`
`isVisible`	`true`

- Exercise
    9. Use the different shapes available from the *figures* project to create a drawing like the following. While you are doing this, write down what you have to do to achieve this. Could it be done in different ways?



Object interaction

We'll now look at a different example.

Exercises

13. Open the *house* package. As usual, you should add a new **`Main`** class with its **`main`** method. Create an object of the **`Picture`** class and invoke its **`draw()`** method. Bingo! you have the same drawing but without having to create it by hand. Also try out the **`setBlackAndWhite`** and **`setColor`** methods.

14. How do you think the **`Picture`** class draws the picture?

The **`Picture`** class was programmed to draw the whole picture by itself. When **`picture1`** was created, it had to create the same objects as you created by hand, ie, two **`Squares`**, one **`Triangle`** and one **`Circle`**, and to call those objects' methods to make them the right size and to move them to the correct position.

Source code

You will spend the rest of the course finding the answer to the question in the previous exercise!

Exercises

15. For starters, open the **`Picture`** class in your editor or IDE. This brings up the source code for the **`Picture`** class, the Java language program that defines the class.
16. Find the code in the **`Picture`** class that does what you did by hand to create and draw the sun. Change the colour of the sun to blue.
17. Add a second sun to the picture. In the **`Picture`** class source code, find the code that looks like

    ```
    private Square wall;
    private Square window;
    private Triangle roof;
    private Circle sun;
    ```

    and add a new field for the second sun, eg,

    ```
    private Circle sun;
    private Circle sun2;  // new field
    ```

    The first line is already in the code, add the second line. In the **`draw`** method, add the code to draw the second sun.
18. Create a sunset by invoking the **`slowMoveVertical`** on the **`sun2`** object.
19. If you added your sunset to the end of the **`draw`** method (so that the sun goes down automatically when the picture is drawn), change this now. Create a new method **`sunset`** containing the sunset code. Now we can call **`draw`** and see the picture with the sun up, and then call **`sunset`** (a separate method) to make the sun go down.

Another example

This example stores student information in a database. Open the *labclasses* package and consult the classes.

Exercise

21. Create an object of type **`Student`**. The constructor of this class needs some parameters. Remember that parameters of type **`String`** must be enclosed in double quotes.

    “What's that, there's no executable class?”, you say. So, go make one.

Return values

Exercise

22. Create some **`Student`** objects. Invoke the **`getName`** method on each object . Explain what is happening.

Methods can return values as indicated by their signature. For example, **`String getName()`** indicates that the method called **`getName`** takes no argument and returns a value of type **`String`**. The signature v**`oid addCredits(int additionalPoints)`** says that method **`addCredits`** takes one argument of type **`int`** and returns no result value, ie, **`void`**.

Objects as parameters

Exercises

23. Create a **`LabClass`** object. You need to specify the maximum number of students in that class.
24. Call the **`numberOfStudents`** method of that object. What does it do?
25. Look at the **`enrollStudent`** method of the **`LabClass`** object. Note that this method takes an argument of type **`Student`**, meaning you need to pass it an object of this type. Enroll a student in the class by invoking the **`enrollStudent`** method.
26. Call the **`numberOfStudents`** and **`printList`** methods on the **`LabClass`** object. What happens?
27. Create the following students and add them to the lab class

Name	Student ID	Credits
Joe Dalton	123	6
Timothy Dalton	456	4

    Print the student information.

28.

29. Set the additional information about the lab: instructor, room, time.

Summary

Exercises

30. In this chapter we have mentioned the data types **`int`** and **`String`**. Java has more predefined types. Find out what they are by looking at the documentation[3].
31. What are the types of the following values?

    0

    "hello"

3 https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

101

-1

true

"33"

3.14159

32. What would you have to do to add a new field, for example one called **`name`**, to a **`circle`** object?
33. Write the signature for a method named **`send`** that has one parameter of type **`String`**, and does not have a return value.
34. Write the signature for a method named **`average`** that has two parameters of type **`int`**, and returns an **`int`** value.

Concept summary

- **object** Java objects model objects from a problem domain.
- **class** Objects are created from classes. The class describes the kind of object; the object represents individual instantiations of the class.
- **method** We can communicate with objects by invoking methods on them. Objects usually do something.
- **parameter** Methods can have parameters to provide additional information for a task.
- **signature** The header of a method is called its signature. It provides information needed to invoke that method.
- **type** Parameters have types. The type defines what kinds of values the parameter can take.
- **multiple instances** Many similar objects can be created from a single class.
- **state** Objects have state. The state is represented by storing values in fields.
- **method-calling** Objects can communicate by calling each others' methods.
- **source code** The source code of a class determines the structure and the behaviour (the fields and methods) of each of the objects in that class.
- **result** Methods may return information about an object via a return value.