Using objects for data structuring

This lecture follows up on the earlier Introduction to Objects. You should re-read these earlier lecture notes before starting here.

Contents

What you already know

Since the beginning of the course you have used three object classes extensively: `String`, `PrintStream` and `Scanner`. What do you know already about them?

- they *behave like types*: you can "make" objects of these types using a regular variable declaration, for example:

```
String s = ...;
Scanner in = ...;
```

- they *provide services* via their methods, which you can invoke using dot-notation, for example:

```
in.nextInt()  // for Scanner

s.equals(...)  // for String
```

From these two things alone you already know two fundamental properties of objects:

- two distinct objects of the same class have "their own" different data; for example to variables of type `String` can hold different character strings;
- the dot-notation accesses the method "inside" the object – so that two different classes can provide different methods with the same name.

These two properties are called *encapsulation* in language theory: the methods and data of an object are encapsulated within it, and two different objects encapsulate different data and methods.

Motivation for creating your own objects

The main reason why you would want to start defining your own classes and objects is when you have groups of variables that are always used together, so you may want to give them a common name.

Of course you have already seen how to use arrays for this purpose; arrays like objects are also a way to group values together. (see Introduction to Arrays for a reminder) However, objects provide you with two additional benefits compared to arrays:

- you can *group values of different types* (in contrast to arrays, which must be homogeneous)
- on top of grouping values together, you *can create your own methods*, too!

Example: car modeling

Say for example you want to implement an application that simulates the traffic of cars in a city. For every car, your program must manipulate its coordinates. So for every car in the program, we have at least 2 related variables that must be manipulated together.

To represent this model of a car, we can use a class with 2 *attributes*, defined as follows:

```
class Car
{
    // Coordinates:
    float posX;
    float posY;
};
```

With this very simple class, we can manipulate cars as follows:

```
// Declare a name for a car and "make" a new car
Car a = new Car();

// Change the attributes in the first car
a.posX = 12;
a.posY = 45;
```

```
Car b = new Car();

// Change the attributes in the 2nd car
b.posX = 44;
b.posY = 14;

System.out.println(a.posY);
System.out.println(b.posY);
```

This program fragment prints 45 and 14: each car *instance* (named by `a` and `b`) has its "own" `posY` coordinate, different between objects.

Say now we want to implement a method for a car which tells how far away it is from another car, so we can write for example:

```
a.distanceTo(b)
```

to measure the distance from a to b.

(Remember, the distance between two points (x1,y1) and (x2,y2) is given by the formula $\sqrt{(x2-x1)^2+(y2-y}$

To do this, we can add the *definition of a method* inside our class `Car` as follows:

```
class Car
{
     // Coordinates:
     float posX;
     float posY;

     // Distance from this car to another
     float distanceTo(Car other)
     {
        float dx = this.posX - other.posX;
        float dy = this.posY - other.posY;
        return Math.sqrt(dx*dx + dy*dy);
     }
};
```

What is going on here?

In the expression `a.distanceTo(b)`, the value of `b` is provided as argument "`other`" to the method body. So the expression "`other.posX`" will effectively refer to `b.posX`.

In addition to this, `distanceTo` receives the object to the left ("`a`" in "`a.distanceTo`") as *implicit argument* called "`this`". So when expressing "`a.distanceTo(b)`" the expression "`this.posX`" inside `distanceTo`'s body (ig. the left side in "`this.posX - other.posX`") will effectively refer to `a.posX`.

Base language constructs

There are 5 language constructs relevant to use object in Java:

- the *class definition*, where you define your own classes (eg. "`class Car { ... }`");
- the *method invocation*, where you use methods in objects (eg. "`a.method(b,c,d)`");
- the *object declaration*, where you prepare a name that can refer to objects (eg. "`Car a;`");
- the *object definition*, where you actually reserve space in memory to store an object (eg. "`new Car()`");
- the *attribute access* construct, where you use a variable "inside" an object (eg. "`a.posX`");

Class definition and method invocation

The constructs for *class definition* and the *method invocation* have already been presented in the earlier lecture Introduction to Objects. Read them again to make sure you know their general form.

Object declaration and definition

The difference between object declaration and object definition is conceptually the same as the difference between array declaration and array definition, seen in the earlier lecture Introduction to Arrays. Read this again before you continue here.

The constructs for object declaration and object definition are defined like for arrays:
Object declaration:

Syntax:

<classname> <objectname> ;

Semantics:

Prepare the name <objectname> so that it can later designate an object of class <classname>.

For example:

```
Car a;
```

prepares the name `a` so it can be later bound to an instance of class `Car`.
Object definition:

Syntax:

`new` <classname> `()`

Semantics:

Reserve space in memory for an instance of the class <classname>. The construct evaluates to the newly created instance.

For example:

```
new Car()
```

creates a new instance of class `Car`.
Like usual in Java, the two constructs can be combined in a single declaration and definition:
Combined object declaration and definition:

Syntax:

<classname> <objectname> `=` `new` <classname> `()` `;`

Semantics:

"Make" a new object of class <classname> and bind it to the name <object-name>.

For example:

```
Car a = new Car()
```

creates a new object of class `Car` and binds it to the name `a`.

Attribute access

The new construct introduced in this lecture is the attribute access, which is similar to the method invocation:

Syntax:

<expression> . <name>

Semantics:

Evaluate the expression to the left. The result of this first evaluation must be an instance containing an instance variable with the name given on the right. The result of the entire evaluation is the instance variable (attribute in the object).

For example:

```
a.posX
```

accesses the variable `posX` in the instance named by `a`.

Class constructor

When using your own objects, there is a pattern that often comes back in programs, for example:

```
// Example 1
Car a = new Car();
a.posX = 12;
a.posY = 45;

// Example 1
Car b = new Car();
b.posX = 121;
b.posY = 11;

// Example 1
Car c = new Car();
c.posX = 66;
c.posY = 86;
```

The common pattern in these examples is "create an object and set its attributes to an initial value." The idea here is that anytime an object is created in a program, almost certainly we want its attributes to be initialized at the same time.

For this all object-oriented languages define a common construct, called the *class constructor*. It is best shown with an example:

```
class Car
{
     // Coordinates:
     float posX;
     float posY;

     // Constructor
     Car(float x, float y)
     {
        this.posX = x;
        this.posY = y;
     }

     ...
};

// then elsewhere:

Car a = new Car(12, 45);
Car b = new Car(121, 11);
Car c = new Car(66, 86);
```

In general, the constructor of a class is a special function definition inside the class body, with the following two syntactic properties:

- the name of the constructor must be the name of the class;
- it does not have a return type (no "`void`", "`float`" etc before its name).

Semantically, the constructor defines a *function that is called automatically when defining an new object with "new"*. Inside the constructor body, the implicit variable "`this`" refers to the newly created object.

So we must extend the construct for an object definition as follows:

Object definition:

Syntax:

`new` <classname> `(` <arguments> `)`

Semantics:

Reserve space in memory for an object of the class <classname>. *If a constructor exists in the class, it is called with the arguments specified between the parentheses.* The definition evaluates to the newly created object after the constructor finishes execution.

For example:

```
new Car(12, 13)
```

This creates a new object of class `Car`, then calls its constructor with the two arguments 12 and 13, then returns the newly created object.

When to use a constructor? As seen in the example above, *the constructor is the place in a program where you set the initial values for an object's attributes*.

Vocabulary

Term	Definition
class	Family of objects, "blueprint" for particular instances
instance	One specific object with its own variables
object	Synonym of "instance"
method	Function in a class, that can be invoked on instances
constructor	Function in a class, automatically invoked by `new`
attribute	Synonym of "instance variable", a variable "inside" an object

Important concepts

- the terms "class", "attribute" and "method" (from earlier lecture);
- the term *"instance"*
- class definition (from earlier lecture);
- method invocation (from earlier lecture);
- method definitions and the special variable "`this`";
- *object declaration* (analogous to array declaration);
- *object definition* with `new` (analogous to array definition);
- *attribute access*;
- *constructor* and providing arguments via `new`;
- when to use constructors.

Further reading

- Think Java, sections 11.1-11.6 (pp. 131-137), section 11.15 (exercises);
- Introduction to programming, section 5.1 up to and including 5.1.1 (pp. 189-192), section 5.2 up to and including 5.2.2 (pp. 199-206);
- Absolute Java, sections 4.1 (pp. 174-184), 4.4 (pp. 226-241).

Copyright and licensing