Lecture 6: Programming in the Large II

Instructor: Musard Balliu, musard@chalmers.se

http://www.cse.chalmers.se/~musard

QUESTIONS?

Plan

- Last time
  1. algorithm design and development

- Today's Plan:

  1. Assignment 1

  2. introduction object-oriented programming

  3. classes and objects

Datatypes

The types we have seen so far are the eight primitive types, `String`, and arrays. We can have arrays of any type, by postfixing the type name with `[]`. Thus, we can have arrays of `String` (`String[]`, arrays of arrays of `String` (`String[][]`), and so on. But all the elements in the array *must have the same type*.

Problem 1: Let's talk about football

Suppose we want to keep a list of football players and the numbers of goals they've scored, for example:

Ibrahimovic	5
Berg	2
Toivonen	3

The names of the players are of type `String` and the numbers of goals is of type `int`. Therefore, we cannot use an array of two elements to represent one of the rows in this table, and then an array of such rows to represent the table. The only possibility is to use two arrays, one for the names column, the other for the number of goals column.

Problem 1: Lists of related types

We end up with something like this:

```
String[] names = {"Ibrahimovic", "Berg", "Toivonen"};
int[] goals    = {5, 2, 3};
```

The player with name `names[i]` has scored `goals[i]` goals.

Problem 1: Lists of related types

Now suppose we want to sort the players in alphabetical order. We can use one of the Java subroutines to do the sorting:

```
Arrays.sort(names);
```

but now the `names` array is

```
[Ibrahimovic, Berg, Toivonen]
```

and it no longer corresponds to the `goals` array.

Moreover, since the `names` array was sorted "in place", we have lost the old order, and we can no longer restore the correspondence.

The same problem arises if we try to rank players according to the goals they scored.

The code

A first look at the `class` construct

With `class`, we can introduce new datatypes. The *simplest* form of such a declaration is:

```
class Type {
    type1 var1;
    ...
    typeN varN;
}
```

The semantics of this declaration is: `Type` is a new type. Values of this type collect together values of type `type1`, ..., `typeN`. If `x` is a value of `Type`, then `x.var1` is a value of type `type1`, etc.

A variable of type `Type` is called an *instance* of the class `TypeName`, or an *object* of type `Type`. Variables `var1`, ..., `varN` are called *fields*, or *instance variables*.

A first look at the `class` construct

Therefore, we can have declarations such as

```
TypeName x;
```

which introduces a new variable, `x`, of type `TypeName`. To initialize `x`, use the keyword `new`:

```
x = new TypeName();
```

This will *allocate* space for the fields (and give them more or less sensible initial values).

After that, the instance variables can be accessed with `x.var1`, ...

A first look at the `class` construct

Returning to the players example, we can introduce a new type which will contain both the name and the number of goals scored by a player:

```
public class Player {
    String name;
    int goals;
}
```

The code

We can now introduce variables of type `Player`:

```
Player ibrahimovic;

ibrahimovic= new Player();
ibrahimovic.name = "Ibrahimovic";
ibrahimovic.goals = 5;
```

Problem 2: global variables

We can now have a list of players, and we want to be able to add new players to the list.

Since we haven't discussed other data structures than the arrays, we'll have to keep our list in an array, which we have to initialize to a sufficiently big size so we don't run out of space when we add new elements.

That means, however, that we'll need to remember how many elements of the array have been initialized, and which is the next one to initialize.

We end up writing something like this.

The code

Problem 2: global variables

This is very repetitive code, due to all the initializations. It is also very error-prone: if we forget to increment `nrPlayers`, we'll overwrite the data in the list.

To get rid of the repetitive initializations is easy: we introduce a subroutine.

The code

This is an improvement, but only a very minor one. The problem is that `add` always acts on the same array, `players`, and always needs `nrPlayers`, but these are not in scope for it, and so we have to pass it repeatedly.

Problem 2: global variables

The solution is therefore to move the variables so that they *are* in scope for `add`. We now have a better looking program:

The code

The function `add` has only the arguments it really needs, the error-prone incrementing of `nrPlayers` appears in only one place, and we have no superfluous duplication of code.

However. . .

This pleasant design is surprisingly *brittle*. Consider the following extension of the program: we want to have *a second list* of players, perhaps players of another team.

Problem 2: global variables

We need another array to store the second list, and another counter for the number of players added to this list. We also need code to add players to this second list.

Now, whatever we do seems wrong: if the second array and counter are not global, we have repetitive, error-prone code.

If we use the same `add` subroutine, we need a conditional to tell us which array to add to, and we get a lot of code duplication.

The code

If we introduce a new `add` subroutine, it will be an almost exact copy of the first:

The code

Object-oriented programming

The problem is that global variables, which allowed us to re-factor our first design, are a double-edged sword. They are in scope for the subroutine `add`, where we need them to be, but they're also in scope for all other subroutines!

The solution, brought about by object-oriented programming, is to use the data structuring mechanism we've needed for `Player` also as a scope controlling mechanism. We'll bundle together the list of players and the counter (which we should have really done from the start) *and with the* `add` *subroutine*.

The PlayerList class
Using the PlayerList class

The essence of OOP is the combination of data structure creation with scope control.

A second look at classes

```
class Type {
    type1 var1;
    // ...
    typeN varN;

    mType1 method1(type1 arg1, ..., typeN1 argN1) {
        statements;
    }

    // ...

    mTypeM methodM(type1M arg1M, ..., typeNM argNM) {
        statements;
    }
}
```

A second look at classes

As before, the class declaration introduces a new type, which collects values together, and values of this type must be allocated using `new` before their fields can be initialized.

The class declaration also introduces `M` new subroutines, for which these variables are in scope.

Subroutines embedded in classes are called *methods*.

Like fields, the methods associated with an instance `x` of type `Type` are called with `x.method1(arg1, ..., argN)`, etc.

When a method references one of the field variables, say `var1`, the value used is that of the field `var1` in the instance with which the method was called.

A first look at constructors

When we use `new` to initialize a new variable of some type, Java, will allocate space for the fields and the methods and will initialize them in some default way, which might not be what we want. For example, integers are initialized to `0`, but strings are not initialized to the empty string, `""`, but rather to `null`, a sort of "no value". If you try to print an empty string, nothing is printed, but if you try to print a null string, the program crashes.

We are given the possibility of controlling the allocation process and deciding the initial values of the fields. This is done by implementing *c*onstructors. These are methods with the same name as the type being introduced by `class`, and with *no return type*.

A first constructor

For example, we can implement a constructor for the class `Player`, which takes as arguments the values of the two fields.

```
public class PlayerC {
    String name;
    int goals;
    PlayerC(String name, int goals) {
        this.name = name;
        this.goals = goals;
    }
}
```

Now we can allocate and initialize the fields at the same time with:

```
PlayerC ibrahimovic;
// ibrahimovic = new PlayerC(); error!
ibrahimovic = new PlayerC("Ibrahimovic", 5);
```

Default constructors vs. explicit constructors

Note that now we can no longer use the default constructor! Once we have provided a constructor (or more), Java will no longer initialize the variable in the default way. If we then want a constructor with no arguments, we have to provide it explicitly.

The keyword `this`

In the constructor, we had in scope two variables called `name` and two called `goals`. For each, we had the field and the constructor argument. We wanted to initialize the instance variables to the values of the constructor arguments, so we needed access to both.

When the constructor is called, as with every subroutine or method, a new block is created and new variables with the names of the arguments are introduced. If these variables have the same name as those in a larger enclosing block, then the latter will be *shadowed* and no longer visible.

In our case, that means that `name` and `goal` in the body of the constructor are the arguments, not the fields. This is why we need the keyword `this`. It allows us to access the field variables (the instance variables of "this" instance) when they would otherwise be shadowed.

Homework

Read the sections 4.2, 5.1, 5.2, and 5.3 of David Eck's book.