DIT948 Programming H16
Lecture 8

Instructor: Musard Balliu, musard@chalmers.se

October 4, 2016

QUESTIONS?

Plan

- Last time
  1. access modifiers
  2. inheritance and polymorphism
- Today's Plan:
  1. packages
  2. modifiers revisited
  3. interfaces and abstract classes

Packages

So far, we have discussed in detail the scope of variables.

The question is: is there a scope of types?

In Java, types are organized in *packages*. Packages are organized similarly (and are related to) the directories in which the compiled classes are stored.

In general, a type will *not* be in scope, unless explicitely brought in scope. Exceptions: the primitive types, the types in the `java.lang` package, and the types in the same package are always in scope.

Packages

A package has the form `level1.level2...levelN`. This corresponds roughly to classes compiled in the directory `levelN`, which is a subdirectory of `levelN-1`, which is a subdirectory of ..., which is a subdirectory of `level2`, which is a subdirectory of `level1`. (In most cases, `N` is not larger than three.)

Assume we want to bring a class `T` which is part of the package `A.B.C`. We can do this in a way similar to accessing the fields of a class:

```
A.B.C.T x = new A.B.C.T();
x.var1 = 3; // fields can then be accessed normally!
```

`A.B.C.T` is called the *fully qualified name* of the type `T`.

Example: Using `Random`

Bringing types in scope: `import`

Accessing types by their fully qualified names is awkward. It is usually easier to use an `import` statement.

Import statements are of the form:

1. individual import of a class: `import java.util.Random;`
   The only class brought in scope is `java.util.Random`, and it can be used throughout the class by its name `Random`.
2. import of all the classes in a package: `import java.util.*;`
   All the classes in the package `java.util` are brought in scope, and can be used by their name (e.g. `Random`, `Scanner`, ...).

Import statements must be put outside the class declaration, at the top of the file.

Adding a class to a package

To add a class to a package `A.B.C` you must do two things:

1. insert the line
   `package A.B.C;`
   at the top of the file, before the import statements and the class definition, and

2. make sure that the file is in the proper place. In principle, that means the file should be in a directory C, which is inside B, which is inside A; but the actual details are a matter of the programming environment and are outside the scope of our discussion.

The default package

If there is no `package` declaration, a class is placed by default in the `nameless` packages.

This package is associated with the current directory. All classes compiled in the current directory will be in scope.

The problem is that classes in the nameless package cannot be brought in scope for classes in other packages. For these would have to use either the fully qualified name, or the package name, but neither exists.

Hence, only "throwaway" code is usually placed in the default package (e.g. code written to test ideas or explain a point).

Access modifiers

Java has an additional mechanism for controlling scope, both at the level of types and at the level of variables and methods.

This is done by adding a keyword to the standard declarations.
The absence of such a keyword is also significant!

Access modifiers for types

If you put the keyword `public` in the beginning of a class declaration, as in

```
public class Player {
```

then the type `Player` can be brought in scope in the usual way in any other class.

If, however, the keyword is missing, as in

```
class Player {
```

then the type `Player` will only be in scope for classes in the same package as `Player` (i.e., for those classes for which it is automatically in scope).

Notice that a `public` class in the default (nameless) package still can't be brought in scope for a class in another package, although for different reasons (see above).

Access modifiers for types

Remark: the default access (no keyword) is called *package-private*.

Question: can a type be out of scope for classes in the same package?

Answer

Access modifiers for fields and methods

In the following, we consider fields and methods of class `T`.

If a member variable or method declaration is preceded by the keyword `public`, as in

```
public int goals;
public int roll();
```

then that member variable or method can be brought in scope in the usual way from any class in which the type `T` is accessible (i.e., if we have an instance `x` of type `T`, we can use `x.goals` or `x.roll()`).

Access modifiers for fields and methods

If a member variable or method declaration is preceded by the keyword `protected`, as in

```
protected int goals;
protected int roll();
```

then that member variable or method can be brought in scope in the usual way only from classes in the same package as `T` and in subclasses of `T` (which could be in a different package).

Access modifiers for fields and methods

If a member variable or method declaration is preceded by the keyword `private`, as in

```
private int goals;
private int roll();
```

then that member variable or method is only in scope in class `T`. `private` variables cannot be brought in scope in any other class, irrespective of whether it is in the same package or not, or whether it's a subclass of `T` or not.

Access modifiers for fields and methods

Finally, if a member variable or method declaration is preceded by no keyword, as in

```
int goals;
int roll();
```

then that member variable or method is only in scope in classes from the same package as `T`. It cannot be brought in scope in a class, from a different package, irrespective of whether it's a subclass of `T` or not.

Remark: the default (no keyword) access is called *package private*, just as in the case of types.

Non-scope-related modifiers

There are two additional non-scope-related modifiers, which apply to fields and methods: `final` and `static`.

A method which is declared `final` in a class cannot be overridden in the subclasses. For example, in order to ensure that a certain quantity will always be computed in the correct way.

A field which is declared `final` cannot be modified (and therefore must be assigned a value), but it can be shadowed in a subclass.

The `static` modifier

In the general case, each object will have its own copy of the value of a field or the body of a method.

However, if the field or method has been declared `static`, then its value or body will be shared by all objects. In other words, it will be independent of the existence of individual instances.

We have often used such fields and methods, for example:

```
Arrays.sort()
System.out.println()
```

and, of course

```
public static void main(String[] args)
```

The `static` modifier

Because `static` fields and methods are independent of any object, they behave similarly to variables and subroutines in the classical programming languages.

That is why, in the beginning, we prefixed everything with `static`.

Object-oriented design

The package mechanism and the access modifiers allow a fine-grained control over what a class *exports* to other classes, i.e., what is introduced in the scope of an other class which imports it.

The elements which are exported form the *interface* of the class (note the name-clash with `interface`). As long as the interface of a class does not change, any program using the class will remain unaffected by changes in the implementation.

Object-oriented design

Because of this, object-oriented programming languages are suitable for large-scale development, where several programmers have to work together on a system. As long as the interfaces are well-defined, each programmer can work independently of the rest.

However, knowing how to split as system in such interfacing units is non-trivial (often more difficult than designing algorithms). This is the subject of the (vast) literature on object-oriented methodology.

Elementary rules of object-oriented design

1. Export as little as possible (do not pollute the namespace of your clients with redundant expressions).

2. Prefer exporting methods to exporting variables; use `get` and `set` methods to access variables;

3. If you have difficulties documenting your interfaces *clearly* and *concisely*, then your interface design is probably wrong;

4. Keep your class hierarchies simple: if you have a superclass which can only have one kind of subclass, the two probably belong together.

There are many such rules: the best guide remains experience.

Optional slides

The following teaching material is optional.

Interfaces and abstract classes

Now that we can put players in an array, we want to sort the array.

The `Arrays` class in Java provides many sorting routines, but, of course, none that has a `PrintablePlayer` as argument. Instead, they can sort instances of `Object`.

However, in order to sort objects, one has to be able to compare them. But there is no corresponding method in `Object`, since there is no default way of comparing arbitrary objects and deciding which is "better".

Interfaces and abstract classes

One possibility would have been to create an *abstract class* to represent objects that can be compared. An abstract class is a type for which no instances can be created, because in general it will have "holes".

The holes are methods which are only declared, not defined. They must be prefixed with the keyword `abstract`. For example:

```
public abstract class Ordered {
    public abstract int compareTo(Object other);
}
```

Interfaces and abstract classes

Subclasses of `Ordered` must then define `compareTo` or be declared abstract themselves.

The sorting method can then be written to work with instances of `Ordered` subclasses of `Ordered`. Note that no instances of `Ordered` as such can exist, but every instance of a subclass of `Ordered` "is an" `Ordered`.

Interfaces and abstract classes

This is not the approach actually used in Java. That is because, in a system which only supports single inheritance, every time the user is required to extend a class, a new level is added to the class hierarchy. This leads to deep hierarchies, which are sometimes difficult to work with.

In order to avoid this, Java provides an additional mechanism which can be used in situations in which a base class only specifies the required signature of methods, but does not implement anything itself. Instead of a class (or an abstract class), one uses an `interface`.

Interfaces are similar to abstract classes, in that they can have no instances. Also, none of the methods may have implementations. Interfaces just collect method declarations.

Since interfaces do not contain any definitions, there is no possible ambiguity in *implementing* them, and a class can implement any number of interfaces simultaneously.

Interfaces and abstract classes

For example, here is the Java interface for objects that can be compared:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

This interface uses *generics*: when implementing the interface, we need to specify the type `T` of objects which we can compare with instances of our class. For example, here is the code which allows us to sort `PrintablePlayers`:

The `PrintablePlayers` with order.

Interfaces and abstract classes

The version we have seen has many flaws. First, we are not really sorting `PrintablePlayers`, but rather `PrintablePlayerOs`. Second, what happens if we want to sort according to the number of goals scored, instead of name? We'd need yet another class, `PrintablePlayerOG`.

The `Arrays` class provides another solution: a sort using a `Comparator`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Classes which implement `Comparator<T>` define a way in which objects of type `T` can be compared. (By the way, since `Object` defines a default `equalsTo`, only the implementation of `compare` is required).

Interfaces exercises

Using comparators, we can sort `PlayerC` instances directly, without having to create new types. Moreover, we can have one comparator for sorting by name, and another for sorting by goals.

Exercise: implement the two comparators and use them to sort a list of players.

Initial code.

Solution for comparison by name.

Solution for comparison by goals scored.

Homework

Read sections 11.7 - 11.15 of Liang's book.