COMP6700/2140 Packages, Modules and Jigsaw

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. Packages: why?
2. Reasons to package: managed namespaces
3. If you `package`, someone will `import`
4. Package and other access modifiers (revisited)
5. Compiling packages
6. API Profiles (Java's new feature)
7. Modules are coming: the project *Jigsaw*

Java's packages

Lecturettes

COMP 6700

Why to package?

This is the Java mechanism for defining, finding and using types:

- Packages create groupings for related interfaces and classes. The package can be placed in a jar-file (together with a manifest describing the package).
- Packages create namespaces that help avoid naming conflicts between types.
- By design, packages are meant to contain functionality which deals with a dedicated domain (*eg*, statistics package, JSON processing *etc*).
- Packages provide a protection domain for a library (or framework). Inside a package, code can cooperate using access to identifiers which are unavailable to external code (have `"package"` access modifier). In a library, a set of interfaces through which clients will use it is declared with `public` access, but their implementing classes can be "hidden" behind the package walls. One cannot hide a presence of a package, but one can make it a "black box".

The *Java* APIs are organised as follows:

- fundamental classes are in `java.lang` (the only one which is always imported implicitly)
- classes for reading and writing (input/output) are in `java.io`, `java.nio.*`, `java.net`
- useful (utility) classes are in `java.util` (incl. *Java Collection Framework*)
- *JavaFX* GUI classes are in various `javax.*` packages (like `javafx.scene`, `javafx.animation` *etc*)
- there are numerous specialised packages from third parties

package declaration and import statement

A class is defined as a package member via the package declaration in the beginning of every class (*before the class declaration*). There has to be only one package declaration in a source file. The package name is implicitly prefixed to each type name contained within the package. If there is no package declaration, the type is placed in a unnamed package (system dependent, one platform can have more than one unnamed packages — one per class loader).

```
package comp6700.labs.lab1; //in every class you wrote in the lab 1
package comp6700.ass.ass1; //in all assignment 1 classes
```

For a "business" code, a multi-name package often has a reversed form of the company URL:

```
package sucks.mybusiness.myproduct; // domain "sucks" is not a joke
```

Remember that the type/member visibility increases in this gradation:

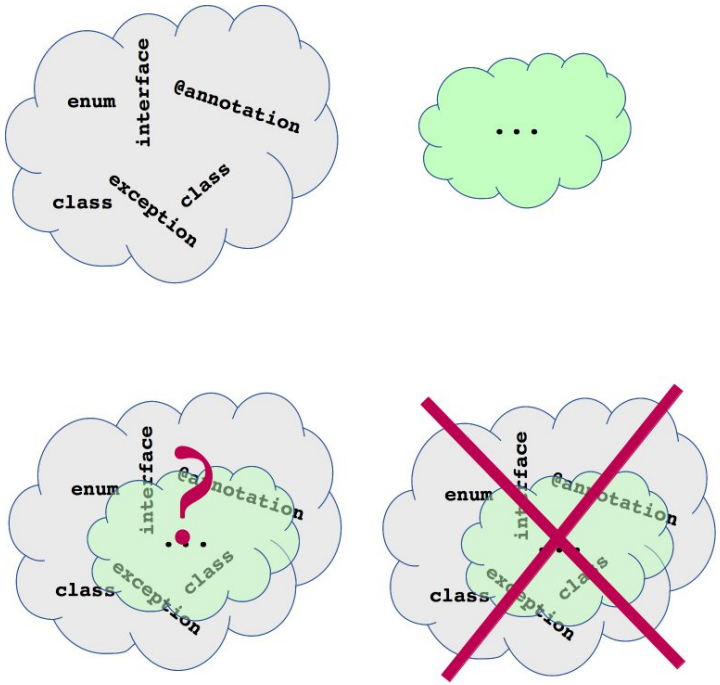
`private` → `<default>` → `protected` → `public`

A name can be used without `import` if it's fully qualified: `java.util.Collections.sort()`. To save on number of statements, use the wildcard `*` import. A `static` member can be imported without other member of a class.

```
import java.io.*; // import on demand: wildcard "*" does not mean *all*
import static java.lang.System.out; // now can simply say out.println()
```

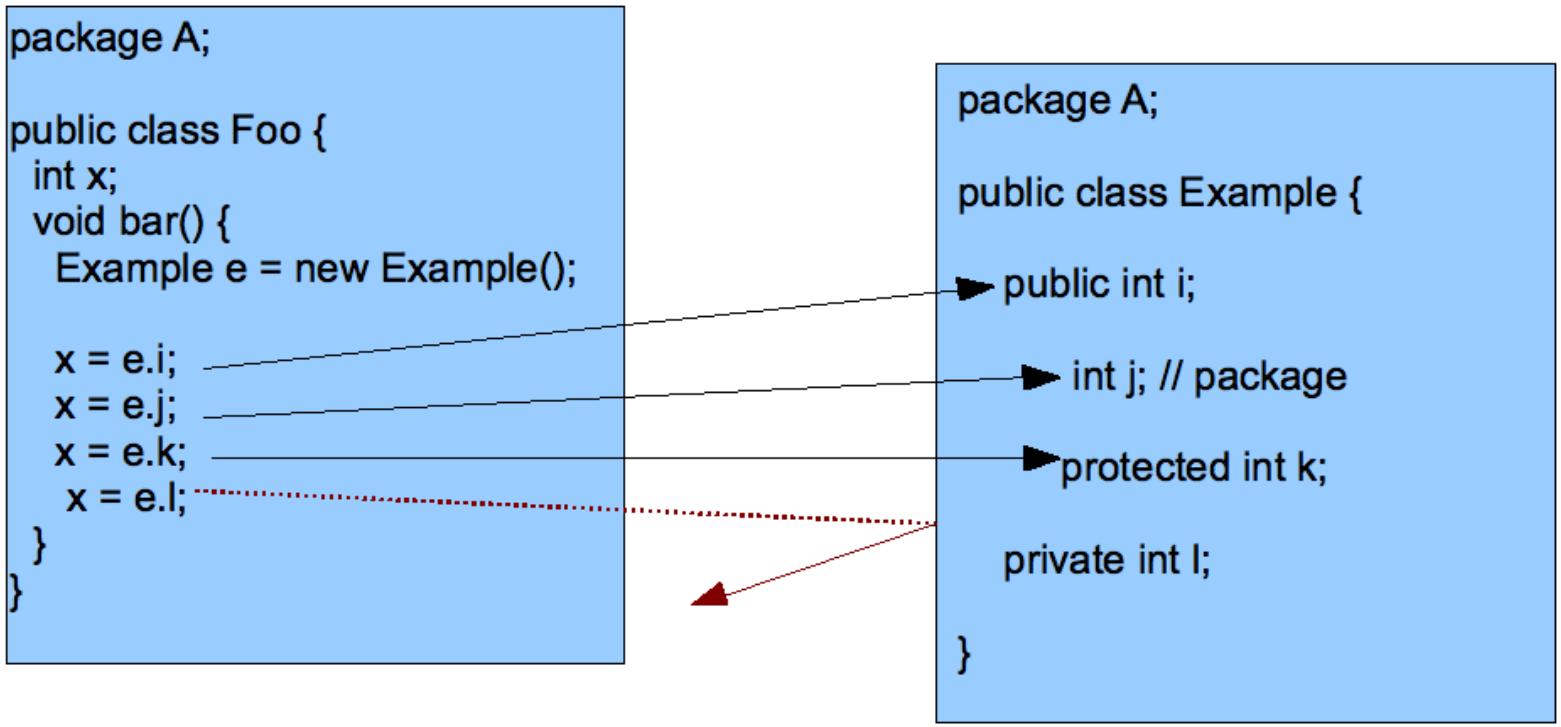
Packages and "sub-"packages

Package names can be *nested* (`java.nio` and `java.nio.files`, or `java.lang` and `java.lang.reflect`), BUT it provides no special access between packages. Nesting helps logic of code organisation, it provides no other benefits. Does the statement `import sucks.mybusiness.*;` also imports everything from `sucks.mybusiness.ass1`?



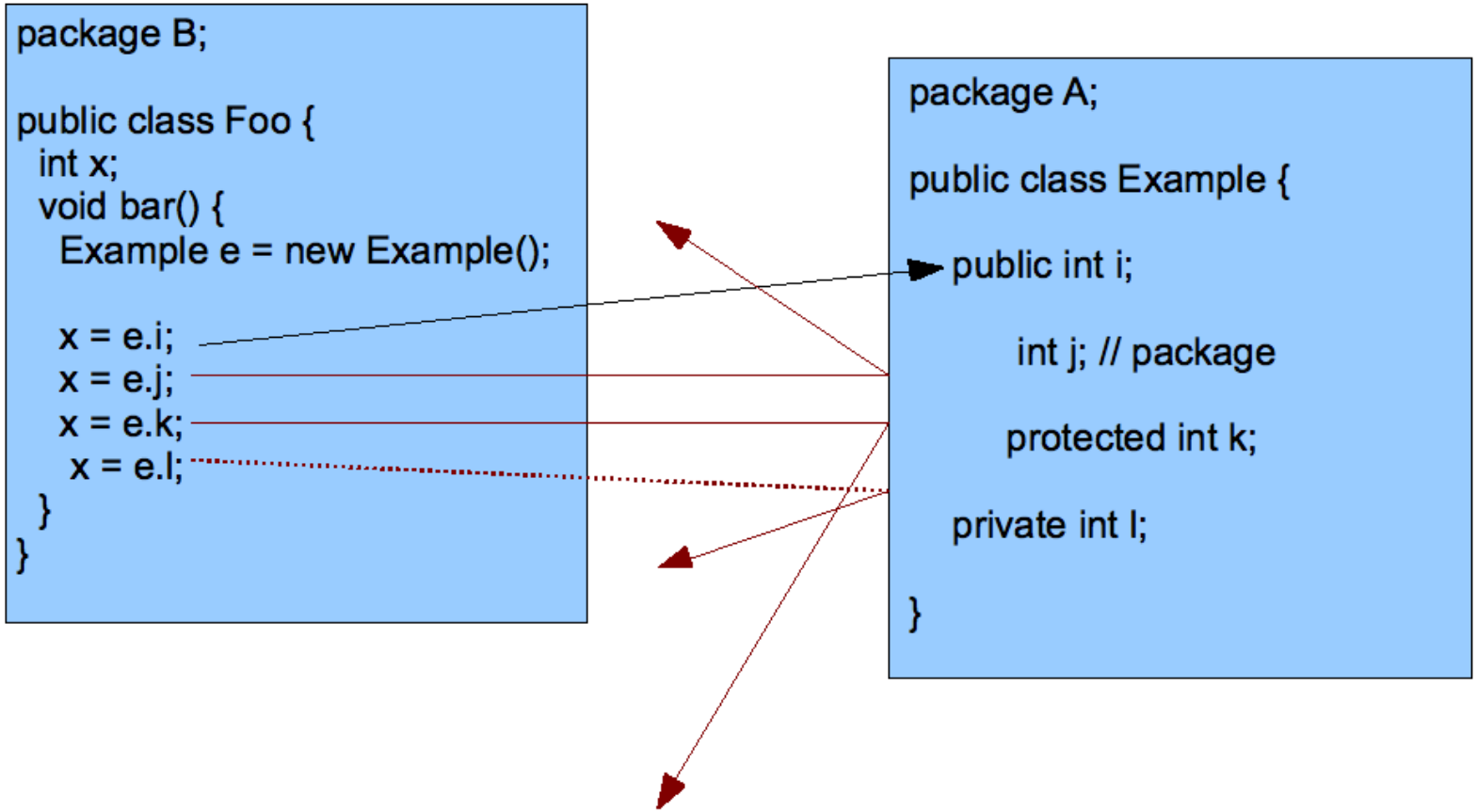
`sucks.mybusiness.ass1` is **not** a subset of `sucks.mybusiness`.

Intra-package access



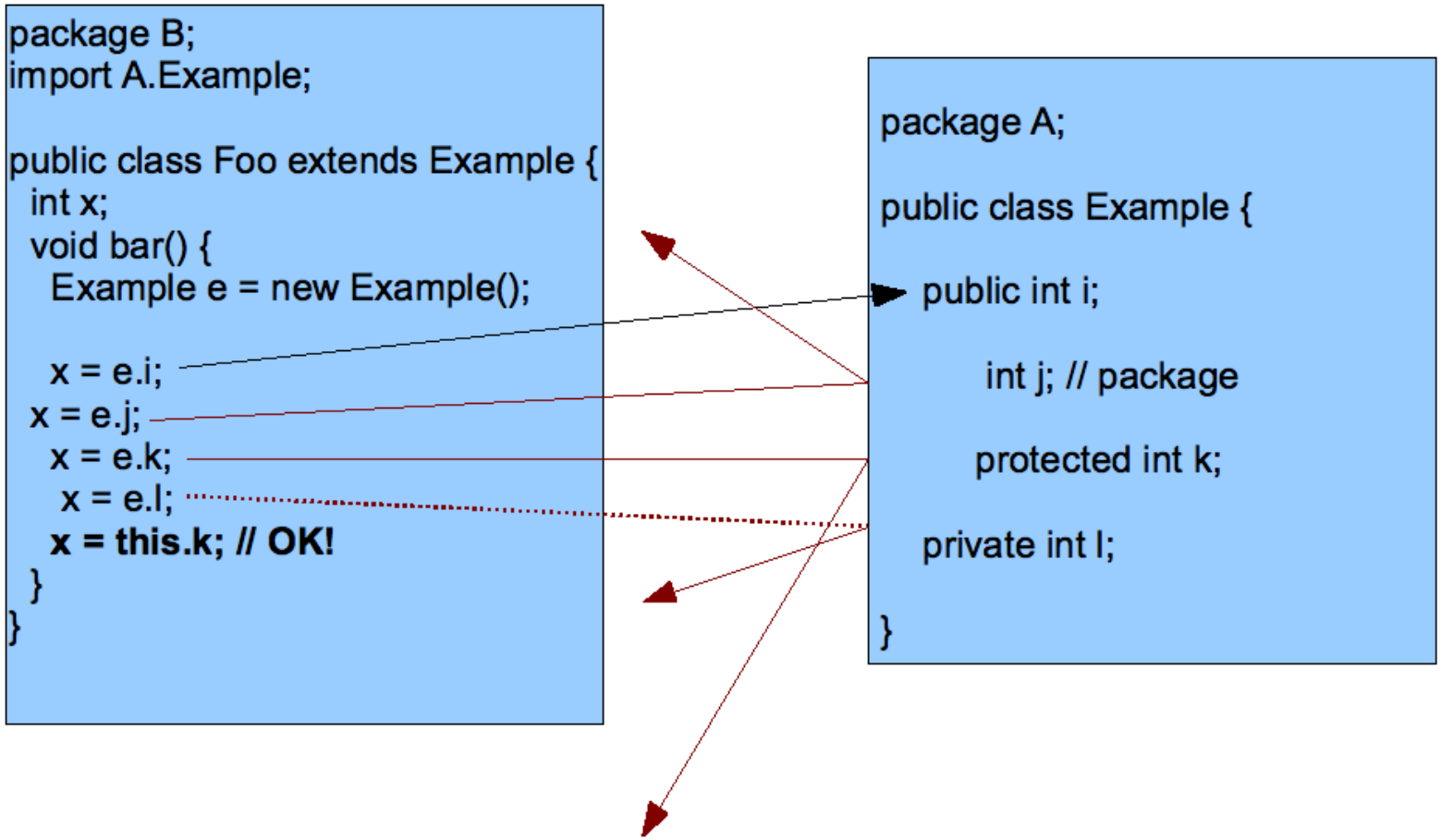
Access two classes in the same package

Trans-package access



Access between two classes in different packages

Package and protected access



Access between a parent and child in different packages

Compiling to package structure

Why to package?

- easy to determine that types are related
- easy to find types that can provide specific functions (like GUI etc)
- to avoid conflict with names in other packages
- to allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package

To create the byte code for your application which has the correct mapping of the package structure to the file structure of compiled class files, use the `-d` option with `javac` command. To make classes available during the compiling or execution, use `-classpath` option, or set `CLASSPATH` *environment variable*.

```
javac -d bin -sourcepath ./src src/RunningHeadline.java
```

This will compile the ass2 classes (from 2017) to their package structure. From the `java` man page (run `man java`):

```
For example, if you specify "-d /home/myclasses" and the class is called
com.mypackage.MyClass, then the class file is called
/home/myclasses/com/mypackage/MyClass.class. If -d is not specified, javac puts
the class file in the same directory as the source file.
```

Profiles

~~The Universe~~ Java is big! Can we optimise which part of it we use when we don't need it all? This can extend the range of devices on which one can run a Java-based software (*Mobile Java*).

One step is already made: *Compact Profiles*. Open the standard API docs, and notice `compact1`, `compact2` and `compact3` labels in the main frame (the type documentation window). The labels means that the package belongs a set of API, which is self-contained (only depends on packages from this set). In terms of package content, the profiles are progressively expanding subsets: `compact3` $\subset$ `compact2` $\subset$ `compact1`. For precise attribution of the standard API packages, see JavaSE API profiles,). Not all parts of the SE API are qualified as a profile (namely, *JavaFX* is marked by its jar-file `jfxrt.jar`).

```
% javac -profile compact1 Hello.java
% jdeps -profile java.lang.String
java.lang (rt.jar)
   -> java.io                                        compact1
   -> java.nio.charset                               compact1
   -> java.util                                      compact1
   -> java.util.regex                                compact1
```

(`jdeps` is a new command from the JDK, you may need to provide the full path to it as it may not be represented by a link when installed on a standard platform). The compilation requirement (`-profile compact1`) *can* be made a part of the spec in our assignments to control use of 3d party libraries.

Modules: the Project *Jigsaw*

So far, units ("modules") of programming in Java have been:

- *classes* and *interfaces*: provide abstraction (impl. and behaviour), composition (inheritance, inner classes, λ-s), static reuse (as *code* units)
- *objects*: provide dynamic reuse (allow type sharing, polymorphism)
- *packages*: provide namespace system, (partial) separate compilation
- *jar-files*: grouping classes/interface, static reuse

A new unit system is coming in *Java 9*: modules and the *modular JDK*.

(From the Project Jigsaw: Goals & Requirements): "A module specifies the modules upon which it depends, and it may define APIs for use by modules which depend upon it. Like a package, class, or interface it has both a specification and one or more implementations. It is a large-grained unit of compilation, packaging, release, transport, and re-use."

Modules will allow:

- to define *truly independent* high-level logical units (for building an application)
- to substitute modules which have different interfaces (higher abstraction level)
- to use and substitute modules of different versions, incl. at run-time (what *OSGi* do)
- to compile units in isolation

Will *not* require a new keyword `module` in the source code — module will only be used in a special (for every module definition) file `module-info.java`.

Module Access

After (if, when?) the Java Platform Module System ("Jigsaw") is released, developers will be able to write their own modules with the defined control regarding what parts are *exported* to the outside clients, and what parts are *open* to outside users. This will be a part of definitions included in the `module-info.java` file, which may look as follows:

```
module myModule {
    exports com.mycom.mymodule;
    opens com.mycom.mymodule;
}
```

For example, the module `java.base` does not open itself for reflective access for us and especially not for the unnamed module, which is the code that we run. If we create a module for our code and name it, then the error message will contain the name of that module. **Demo** Entropy.java.

Note the would be keyword `module` is not really a keyword, since it will not have any special meaning in an "ordinary" Java source file; as such, there will be no danger of breaking old code which might have contained identifiers `module`.

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 2.5
- Oracle's Java Tutorial on Packages
- Project Jigsaw: Goals & Requirements
- Project Jigsaw: Module System Quick-Start Guide
- The State of the Module System