Packages

To make classes easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and interfaces into **packages**.

A **package** is a program module that contains classes, interfaces and other packages (**subpackages**).

Each package has a **name**:

- a single identifier – a name of a top level package
- of the form `Q.Id`, where `Q` is a name of a package and `Id` is an identifier – a name of a subpackage

Examples of package names:

```
points
java.lang
com.sun.security
drawing.figures
```

Packages (cont.)

A package to which a class or an interface belongs is specified at the beginning of the source file containing this class or interface using the following syntax, where `name` is the name of the package:

```
package name;
```

For example, a file `Line.java` may look like this:

```
package drawing.figures;

class Line extends Figure {
      ...
}
```

This specifies the class `Line` that belongs to the package `drawing.figures`.

When no package is specified at the beginning of the source file, the class or interface defined in this file belongs to a special **unnamed package**.

Canonical Names

Each class or interface has a **fully qualified (canonical) name** that specifies also the package to which it belongs.

For example, the fully qualified name of the previously defined class `Line` is:

```
drawing.figures.Line
```

It is possible to use the same name for two classes or interfaces as long as they belong to different packages (and so they have different canonical names).

For example, can define another class `Line` in a package `net.connections` with the canonical name:

```
net.connections.Line
```

The canonical name of a class or interface that belongs to the unnamed package is the name of this class or interface, for example:

```
Line
```

Canonical Names (cont.)

We can always use a fully qualified name of a class or interface when we refer to this class or interface:

```
drawing.figures.Line line = new drawing.figures.Line();
```

When we use a simple name (i.e., when we do not use the canonical name) we refer to a class or interface in the current package:

```
Line line = new Line();
```

A package may not contain two members of the same name, or a compile-time error results. For example:

- The package `drawing.figures` cannot contain other class or interface named `Line`.
- The package `drawing.figures` cannot contain a subpackage `Line`.
- The package `drawing` cannot contain a class or interface named `figures`.

Public Classes and Interfaces

Only classes or interfaces declared **public** can be accessed in other packages.

For example, the class drawing.figures.Line declared this way can be accessed in all packages:

```
package drawing.figures;

public class Line extends Figure {
      ...
}
```

If it would be declared the following way then it can be accessed only in (classes and interfaces in) the package drawing.figures:

```
package drawing.figures;

class Line extends Figure {
      ...
}
```

Hierarchy of Packages

The hierarchical naming structure for packages is intended to be convenient for organizing related packages, but has no other significance.

For example there is no special access relationship between classes defined in the following packages:

```
drawing.figures
drawing.figures.colors
drawing.menu
```

Package names correspond to directories in a file system. For example the class drawing.figures.Line should be stored in a file named

```
drawing/figures/Line.java      (on Unix)
```

resp.

```
drawing\figures\Line.java      (on MS Windows)
```

Note: Files containing classes from the unnamed package should be stored in the current working directory.

Import Declarations

It is always possible to refer to classes and interfaces from other packages using their canonical names.

It is possible to use import declarations to import classes and interfaces from other packages and to refer to them using simple names.

The are two types of import declarations:

- single type declarations – imports one class or interfaces

```
import drawing.figures.Line;
```

- import on demand declarations – imports all public classes and interfaces from the given package

```
import drawing.figures.*;
```

A source file can contain any number of import declarations.

Import Declarations (cont.)

The file drawing/figures/Line.java contains:

```
package drawing.figures;

public class Line extends Figure {
      ...
}
```

The file drawing/menu/Commands.java contains:

```
package drawing.menu;

import drawing.figures.Line;   // single type import

class Commands {
    public Line createLine() {   // Line refers to the class
        Line line = new Line();  // drawing.figures.Line
           ...
    }
       ...
}
```

Import Declarations (cont.)

The file drawing/figures/Line.java contains:

```
package drawing.figures;

public class Line extends Figure {
      ...
}
```

The file drawing/menu/Commands.java contains:

```
package drawing.menu;

import drawing.figures.*;    // on demand import

class Commands {
    public Line createLine() {    // Line refers to the class
        Line line = new Line();   // drawing.figures.Line
           ...
    }
       ...
}
```

Import Declarations (cont.)

When a source file contains a single class (or interface) name then the definition of the class is found using the following procedure:

- If the class is defined in the source file or if it is imported using single type import declaration, the corresponding definition is used.
- If the class with the given name is defined in the same package (but in other file), that definition is used.
- If the class is defined in some package imported using on demand import declaration, that definition is used.
- Otherwise a compile-time error results.

Note: Only public classes and interfaces can be imported from other packages.

Import Declarations (cont.)

Some remarks:

- When a class defined in a source file has the same single name as the class imported using **single type** import declaration, a compile-time error results.
  So the following program causes a compile time error.

```
package drawing.menu;

import drawing.figures.Line;

class Line {  // Compile error!
       ...
}
```

- It is also not possible to use more than one **single type** import declaration with the same single name of a class or interface:

```
import drawing.figures.Line;
import net.connections.Line;  // Compile error!
```

Import Declarations (cont.)

- When a source file contains single type import declaration and there is a class with the same single name defined in the same package, but in other file, then the class specified in the import declaration is used.
- It is a compile-time error when a single name is used to refer to a class that is defined in two or more packages imported using on demand import declarations:

```
import drawing.figures.*; // contains class Line
import net.connections.*; // contains class Line
  ...
Line line = new Line();   // Compile error!
                          // Fully qualified
                          // name must be used.
```

- On demand import declarations do not conflict with single type import declarations or with classes defined in the given package.

Source Files

A Java source file (also called **compilation unit**) has the following structure:

- package declaration
- import declarations
- type declarations

The ordering of these parts is mandatory. Each of them is optional.

- **Package declaration** is always of the form

  ```
  package name;
  ```

  where name is the (canonical) name of the package.
- **Import declarations** is a sequence of any number of import declarations (single type and on demand).
- **Type declarations** is a sequence of any number of class and interface definitions.

Source Files (cont.)

- When a source file contains a public class or interface named X, the name of the source file must be `X.java`.
- When a source file contains a class or interface named X that is referred from other source files, the name of the source file must be `X.java`
- The above rules mean that a file may contain at most one class or interface that is either public or that is referred from other source files.
- When source files are compiled, a file `X.class` is created for each class or interface named X.
- Names of directories must correspond to names of packages.

Ant

When we have a program consisting of many source files, we usually do not run compiler manually, but use some special tool for it.
A standard tool used for this purpose for Java programs is called **Ant**.

- Dependencies between files can be specified using a special language.
- It is not always necessary to compile all source files, but only some of them. Ant automatically figures out which files should be compiled and calls compiler on them.
- Ant is not a part of JDK, but can be downloaded from `http://ant.apache.org`.
- Most of development environments use Ant internally for management of projects.

Package Names

Packages **java** and **javax** and their subpackages are reserved for standard classes, so no classes or interfaces should be defined by a user in these packages.

A short overview of the most important standard packages:

- **java.lang** – fundamental classes for Java programming language
- **java.util** – miscellaneous utility classes (abstract data types, manipulation with date and time, ...)
- **java.io** – classes for input and output from and to files and for manipulation with files
- **java.net** – classes for network communication
- **java.awt** – Abstract Window Toolkit – classes for creating user interfaces and for painting graphics and images
- **java.applet** – classes for creating applets
- **javax.swing** – modern user interface Swing
- **javax.sound** – classes for working with sound

Package Names (cont.)

Some remarks:

- Classes and interfaces from package **java.lang** are always automatically imported as if the following import declaration would be used:

```
import java.lang.*;
```

- By convention package names contain only lower-case letters.

Package Names (cont.)

When some packages are widely distributed the following convention is suggested:

- We create a unique package name from an Internet domain name belonging to an organization that produces the package by reversing this domain name component by component.
  For example from a domain name

```
mycompany.com
```

  we create a package name

```
com.mycompany
```

- All other packages are then created as subpackages of this package.

This convention allows to avoid package names conflicts.

Access Control

Access to a member of a class can be specified using one of the keywords **`public`**, **`protected`** or **`private`**, or it may not be specified (**default access**):

- **`public`** – it can be accessed from any class
- **`protected`** – it can be accessed from any subclass and from any class in the same package
- **(default)** – it can be accessed from any class in the same package
- **`private`** – it can be accessed only from the class where it is defined.

When we override a method in a subclass it must be declared with the same or more permissive access than the method in the superclass.

Classes Without Instances

The standard way how to define a class such that it is not possible to create instances of this class is:

- to declare a constructor of no arguments and make it **`private`**
- never invoke this constructor
- declare no other constructors

Class of this form usually contains class methods and variables.

An example of such class is the class `java.lang.Math` containing standard mathematical functions:

```
public final class Math {

    private Math() { }   // never instantiate this class

          ...       // class variables and methods
}
```

Modifiers

A **modifier** specifies some special property of a class, an interface, a method, an attribute, or a constructor.

All possible modifiers in Java are:

public	**protected**	**private**
static	**abstract**	**final**
native	**synchronized**	**transient**
volatile	**strictfp**	

- Modifiers are used in front of a declaration.
- The ordering of modifiers is not important when more than one modifier is used.
- Some modifiers can be used only in some contexts.
- At most one of keywords **`public`**, **`protected`** and **`private`** can be used in one declaration.
- It is an error to use the same modifier more than once.

Modifiers (cont.)

- Class modifiers:
  public abstract final strictfp
- Field (attribute) modifiers:
  public protected private
  static final transient volatile
- Method modifiers:
  public protected private abstract static
  final synchronized native strictfp
- Constructor modifiers:
  public protected private
- Interface modifiers:
  public