Lecture 13 — Packages

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 8 of the textbook
- Using packages
- Importing packages
- The classpath
- Access modifiers and packages

Names and Namespaces

- In programming, we need to give everything a name (classes, methods, variables...)
- Classes in particular must all have different names
- If there are many programmers working on a large code base name collisions are bound to occur
- Java will allow two classes to have different names so long as they are in different *namespaces*

Packages to the Rescue

- We can bundle together groups of related classes into *packages*
- So far, we have been using a single package: the default package
- Each package is a new namespace
- A package is basically a group of classes in a namespace
- It *encapsulates* a collection of classes
- It is a bit like a meta-level class

package

```java
package pkga;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("MyClass");
    }
}
```

- To put a class in a package you need to do 2 things
- First, put `package pkgname;` at the top of the .java file
- Second, put the class in a folder with the same name as the package: `/path/to/project/pkgname/`

package

```java
package pkga;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("MyClass #1");
    }
}
```

- Assume you are in the directory where you want to store your project (e.g., the `Lecture13/` directory)
- We can compile `MyClass` via `javac pkga/MyClass.java`
- We can run it via `java pkga.MyClass`
- Notice that we use dot (.) to access classes inside a pacakge

package

```
public class NoPackage {
    public static void main(String[] args) {
        System.out.println("NoPackage");
    }
}
```

- What happens if we forget package? We get an error
- `Error: Could not find or load main class pkga.NoPackage`
- Generally speaking, the directory name and `pacakge name;` must match
- Java will search for a package by looking in the corresponding directory, if your class isn't there, it will crash

Nested Packages

```
package pkgb.subpackage1;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("MyClass #2");
    }
}
```

- Nested packages work. All we need to do is nest the directories
- They can be accessed by chaining the dot operator: `pkgb.subpackage1.MyClass`
- Notice that now we have `pkga.MyClass` and `pkgb.subpackage1.MyClass`?
- We could go further and add `pkgb.subpackage2.MyClass`!
- Let's take a look at the Lecture13 code folder to see this in action

Using `import`

- All this has assumed we are calling `javac` and `java` from whatever we decided is the project directory
- By default, java will import the default package, which is the directory it was called from
- All the packages in sub directories will also be available for import
- We can access the contents of another package using `import pkg.subpkg.etc;`

The `vehicle` Package

- Let's take a look at the `vehicles` package
- It contains two classes: `Car` and `Bicycle`
- We're going to see how to import it and use it in a program
- The directory structure looks like:
  - `Lecture13/`
  - `Lecture13/other_directories...`
  - `Lecture13/vehicles/`
  - `Lecture13/vehicles/Bicycle.java`
  - `Lecture13/vehicles/Car.java`
- Let's see how to import from `vehicles`

The import keyword

```
import vehicles.Bicycle;
import vehicles.Car;

public class VehicleExample {
    public static void main(String[] args) {
        Bicycle bike = new Bicycle(20);
        Car car = new Car(100);

        System.out.println("Bike top speed: " + bike.getTopSpeed());
        System.out.println("Car top speed: " + car.getTopSpeed());

        System.out.println("Bike wheels: " + bike.wheels());
        System.out.println("Car wheels: " + car.wheels());
    }
}
```

The import keyword

```
import vehicles.*

public class VehicleExample {
    public static void main(String[] args) {
        Bicycle bike = new Bicycle(20);
        Car car = new Car(100);

        System.out.println("Bike top speed: " + bike.getTopSpeed());
        System.out.println("Car top speed: " + car.getTopSpeed());

        System.out.println("Bike wheels: " + bike.wheels());
        System.out.println("Car wheels: " + car.wheels());
    }
}
```

- It is possible to import everything in a package at once
- `import vehicles.*;`

The `import` keyword

```
package pkgc;
import vehicles.*;

public class VehicleExample {
    public static void main(String[] args) {
        Bicycle bike = new Bicycle(20);
        Car car = new Car(100);

        System.out.println("Bike top speed: " + bike.getTopSpeed());
        System.out.println("Car top speed: " + car.getTopSpeed());

        System.out.println("Bike wheels: " + bike.wheels());
        System.out.println("Car wheels: " + car.wheels());
    }
}
```

- If we copy VehicleExample into the package/directory `pkgc` the import statement is the same
- Imports are not relative to the package you are in!
- They describe the location of the package from wherever you ran the program from
- You should always run programs (e.g., using `java pkga.subpkgb.ClassName`) from your top-level directory to avoid confusion!

The `import` keyword

- Do not do this
- Open folder `pkgc`
- `javac VehicleExample.java` then `java VehicleExample`
- Always do this
- Stay in the root directory of the project
- `javac pkgc/VehicleExample.java` then `java pkgc.VehicleExample`
- Note: IDEs such as Eclipse will handle this for you. However, it's good to know what they are doing under the hood

The Classpath

- When we run a Java program it looks for packages that are in sub directories
- Whenever Java sees an `import` statement, it looks in the corresponding sub directories starting from the active directory
- You can also tell Java to look for packages in other places too
- It will look in all the directories specified in the *classpath*
- It is possible to modify the classpath. More on that later

Java API

- By default, the classpath will contain the current directory
- In addition to the classpath, it will always search the Java API
- This is why you can `import java.util.Scanner;`
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html`
- Note that `java.lang` contains classes like `String` and `Object` and is always implicitly imported along with everything in the default package

Adding to the Classpath

- If we want to change the classpath we can use the `--class-path` flag
- We can specify multiple directories by separating them with ":"
- `javac --class-path=path/to/package:path/to/other/package ...`
- `java --class-path=path/to/package:path/to/other/package ...`
- Note that on Windows paths to directories use backslash!
- Don't forget to include the current directory! (e.g., `--class-path=some/path:.`)
- An IDE will usually handle this for you after you tell it to add something to the classpath
- It is also possible to modify the `CLASSPATH` environment variable the same way

`Greeter` Example

- There are two code folders for this lecture: Lecture13 and Lecture13_Extra
- We are going to add Lecture13_Extra to the classpath
- Let's first take a look at Lecture13/ClasspathExample.java and Lecture13_Extra/pkgd/Greeter.java
- Now, we can run the program
- `javac --class-path=../Lecture13_Extra/:.  ClasspathExample.java`
- `java --class-path=../Lecture13_Extra/:.  ClasspathExample`

.jar Files

- Packages can be turned into a .jar file
- JAR stands for Java Archive
- Straightforward to create:
  `https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html`
- They are an convenient way to distribute packages
- They can be added to the classpath too
- `javac --class-path=path/to/mypackage.jar ...`

Packages and Access Modifiers

- We have seen `private`, `public`, and `protected`
- You may have noticed we sometimes do not specify any of these 3
- These all modify how classes are accessed
- They also interact with packages!

Packages and Access Modifiers

Access Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
(default)	Yes	Yes	No	No
private	Yes	No	No	No

- Note that "(default)" is what happens when we do not specify an access modifier
- If a two classes are in the same package, they can access all (default) members of one another
- In addition, any subclass can access protected members even if the subclass is in a different package
- This distinction is useful: if you want people to extend your class and use a member, make it protected, otherwise make it (default)

Packages and Access Modifiers

```
package vehicles;

public class Boat {
    private int topSpeed;

    public Boat(int topSpeed) {
        this.topSpeed = topSpeed;
    }

    int getTopSpeed() {
        return topSpeed;
    }

    protected String description() {
        return "Boat with top speed " + getTopSpeed();
    }
}
```

- We're going to look at an example of access modifiers and packages
- Consider this `Boat` class

Packages and Access Modifiers

```java
package vehicles;

public class MotorBoat extends Boat {
    public MotorBoat(int topSpeed) {
        super(topSpeed);
    }

    public String description() {
        // We can access getTopSpeed because we're in the same package
        return "Motorboat with top speed " + getTopSpeed();
    }
}
```

- `MotorBoat` is a subclass in the same package

Packages and Access Modifiers

```java
package vehicles.extra_vehicles;

import vehicles.Boat;

public class Kayak extends Boat {
    public Kayak(int topSpeed) {
        super(topSpeed);
    }

    @Override
    protected String description() {
        // Oh no, can't access getTopSpeed because it is package only
        return "Kayak with top speed " + getTopSpeed();
    }
}
```

- `Kayak` is a subclass in a different package (sub packages don't count as the same package)