Packages and Interfaces

Packages

- Packages are containers for classes.
- Packages are used to keep the class name space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- The package is both a naming and a visibility control mechanism.
    - Define classes inside a package that are not accessible by code outside that package.
    - Define class members that are exposed only to other members of the same package.
- The package statement syntax.

```
package <top_pkg_name>[.<sub_pkg_name>];
```

Packages and Member Access

- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- Java addresses four categories of visibility for class members.
    - Subclasses in the same package.
    - Non-subclasses in the same package.
    - Subclasses in different packages.
    - Classes that are neither in the same package nor subclasses.

Understanding Protected Members

	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Importing Packages

- Java includes the import statement to bring certain classes, or entire packages, into visibility.
- In a Java source file, import statement occur immediately after the package statement, and before any class definitions.
- The import statement syntax.

```
import pkg1 [.pkg2].(classname | *);
```

Java's Class Library Is Contained in Packages

- All of the standard Java classes included with Java are stored in a package called java.
- The basic language functions are stored in a package inside of the java package called java.lang.
- The java.lang package is imported implicitly by the compiler for all programs.

Interfaces

- Using the interface keyword, Java allows you to fully abstract a class's interface from its implementation.
- Interfaces are syntactically like classes, but they lack instance variables, and as a rule, their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface.
- Also, one class can implement any number of interfaces.

Defining an Interface

General syntax of an interface.

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        //...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

Implementing Interfaces

- To implement an interface, include the implements clause in a class definition, and then create the methods required by the interface.

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]] {
    // class-body
}

- If a class implements more than one interface, the interfaces are separated with a comma.

Using Interfaces References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- The method to be executed is looked up dynamically at run time.

Variables in Interfaces

- Variables declared in an interface are constants that can be shared by multiple classes.

```
interface SharedConstants {
       int NO = 0;
       int YES = 1;
       int MAYBE = 2;
       int LATER = 3;
       int SOON = 4;
       int NEVER = 5;
}
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword extends.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

```java
interface FirstInterface {
	public void first();
}

interface SecondInterface extends FirstInterface {
	public void second();
}

class MyClass implements SecondInterface {

	@Override
	public void first() {
		System.out.println("This is the first method");
	}

	@Override
	public void second() {
		System.out.println("This is the second method");
	}

}
```

Default Interface Methods

- Since JDK 8, Java added a new capability to interface called the default method.
- A default method lets you define a default implementation for an interface method.
- A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code.
- The declaration is preceded by the keyword default.

```java
public interface MyIF {
	// This is a "normal" interface method declaration.
	// It does NOT define a default implementation.
	int getNumber();

	// This is a default method. Notice that it provides
	// a default implementation.
	default String getString() {
		return "Default String";
	}
}
```

Use static Methods in an Interface

- JDK 8 added another new capability to interface; the ability to define one or more static methods.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- A static method is called by prefixing the interface name.

Private Interface Methods

- Java 9 onward, you are allowed to include private methods in interfaces.
- Using private methods, now encapsulation is possible in interfaces as well.
- These private methods will improve code re-usability inside interfaces.
- For example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to its implementing classes.
- Using private methods in interfaces have four rules:
    - Private interface method cannot be abstract.
    - Private method can be used only inside interface.
    - Private static method can be used inside other static and non-static interface methods.
    - Private non-static methods cannot be used inside private static methods.

```
interface MyInterface {
	default void first () {
		System.out.println();
		second();
	}

	private void second() {
		System.out.println();
	}
}
```