Introduction to JAVA Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

October 9, 2020

Outline

Package

- It used to group related classes
- These are container for classes, just like a directory containing files
- It help resolving name collision of classes
- The idea is similar to namespace in C++
- Helps writing better maintainable code

Creating a Package

- Write `package packageName;` as the first statement in a source file
- This places all classes defined in the file into the package
- Here `package` is a keyword
- The package name should be written in lower case to avoid conflict with class names
- Packages may be arranged in multilevel, e.g. writing `package a.b.c;` creates a package `a` containing package `b` which again contains package `c`
- Java places all classes having the same package name declaration together

Creating a Package

```
package a.b.c;
class ABC {
    public static void main(String[] args) {
        //...
    }
}
```

- Java manages its packages by creating directories of same names
- Here the class file `ABC.class` should be placed under the directory structure `./a/b/c/`

Creating a Package

```
package a.b.c;
class ABC {
    public static void main(String[] args) {
        //...
    }
}
```

- Java manages its packages by creating directories of same names
- Here the class file `ABC.class` should be placed under the directory structure `./a/b/c/`
- This can be done automatically by passing `-d dest` parameter while compiling

  ```
  javac -d . ABC.java
  ```

Creating a Package

```
package a.b.c;
class ABC {
    public static void main(String[] args) {
        //...
    }
}
```

- Java manages its packages by creating directories of same names
- Here the class file `ABC.class` should be placed under the directory structure `./a/b/c/`
- This can be done automatically by passing `-d dest` parameter while compiling

  ```
  javac -d . ABC.java
  ```

- After compiling the program can be run by specifying the fully qualified name of the class

  ```
  java a.b.c.ABC
  ```

Using a Package

- To access a class `ABC` from package `a.b.c` we have two options

Using a Package

- To access a class `ABC` from package `a.b.c` we have two options
- Import the class into current scope and refer it to normally by specifying its name: `import a.b.c.ABC;`
  - Here `import` is a keyword
  - We specify a class by its fully qualified name
  - An import statement must be written immediately after package declaration and before any classes declarations
  - Writing `import a.b.c.*;` will import the whole package

Using a Package

- To access a class `ABC` from package `a.b.c` we have two options
- Import the class into current scope and refer it to normally by specifying its name: `import a.b.c.ABC;`
  - Here `import` is a keyword
  - We specify a class by its fully qualified name
  - An import statement must be written immediately after package declaration and before any classes declarations
  - Writing `import a.b.c.*;` will import the whole package
- Alternatively we may directly use a class without importing it, and always refer it by its fully qualified name

Using a Package

- To access a class `ABC` from package `a.b.c` we have two options
- Import the class into current scope and refer it to normally by specifying its name: `import a.b.c.ABC;`
  - Here `import` is a keyword
  - We specify a class by its fully qualified name
  - An import statement must be written immediately after package declaration and before any classes declarations
  - Writing `import a.b.c.*;` will import the whole package
- Alternatively we may directly use a class without importing it, and always refer it by its fully qualified name
- Java automatically imports the package `java.lang` which contains definitions of many fundamental classes like:
  `Object, String, Thread` etc. ref here for more

Access Protection

	private	default	protected	public
same class				
same package subclass				
same package non-subclass				
different package subclass				
different package non-subclass				

Access Protection

	private	default	protected	public
same class				✓
same package subclass				✓
same package non-subclass				✓
different package subclass				✓
different package non-subclass				✓

Access Protection

	private	default	protected	public
same class	✓			✓
same package subclass	×			✓
same package non-subclass	×			✓
different package subclass	×			✓
different package non-subclass	×			✓

Access Protection

	private	default	protected	public
same class	✓	✓		✓
same package subclass	×	✓		✓
same package non-subclass	×	✓		✓
different package subclass	×	×		✓
different package non-subclass	×	×		✓

Access Protection

	private	default	protected	public
same class	✓	✓	✓	✓
same package subclass	×	✓	✓	✓
same package non-subclass	×	✓	✓	✓
different package subclass	×	×	✓	✓
different package non-subclass	×	×	×	✓

Interfaces

- To specify what a class must do, but not how it does that
- It is much alike to abstract classes

Interfaces

- To specify what a class must do, but not how it does that
- It is much alike to abstract classes

- We use the keyword `interface` instead of `class`

Interfaces

- To specify what a class must do, but not how it does that
- It is much alike to abstract classes

- We use the keyword `interface` instead of `class`

- All methods are implicitly treated as `abstract` (no definitions)

Interfaces

- To specify what a class must do, but not how it does that
- It is much alike to abstract classes

- We use the keyword `interface` instead of `class`

- All methods are implicitly treated as `abstract` (no definitions)

- Just like an abstract class, an interface also cannot have any instances

Interfaces

- To specify what a class must do, but not how it does that
- It is much alike to abstract classes

- We use the keyword `interface` instead of `class`

- All methods are implicitly treated as `abstract` (no definitions)

- Just like an abstract class, an interface also cannot have any instances

- All member variables of an interface are implicitly `static` and `final`

Using Interfaces

```
public interface MyInterface {
        void foo(); // implicitly abstract
        int x = 10; // implicitly final and static
}
```

- Typically an interface is inherited by a class and definitions of the unimplemented methods are provided
- We say a class `implements` an interface for this

```
public class MyClass implements MyInterface {
        @Override
        void foo() {
                //method definition
        }
}
```

Using Interfaces

```
public interface MyInterface {
        void foo(); // implicitly abstract
        int x = 10; // implicitly final and static
}
```

- Typically an interface is inherited by a class and definitions of the unimplemented methods are provided
- We say a class `implements` an interface for this

```
public class MyClass implements MyInterface {
        @Override
        void foo() {
                //method definition
        }
}
```

- An implementing class must provide definitions of all unimplemented methods of the interface
- Failing to do so, the class must be declared as `abstract`

Using Interfaces

- Dynamic method dispatch is also possible with interfaces

```
MyInterface obj = new MyClass(); //allowed
obj.foo(); //calls foo() in MyClass
```

Using Interfaces

- Dynamic method dispatch is also possible with interfaces

```
MyInterface obj = new MyClass(); //allowed
obj.foo(); //calls foo() in MyClass
```

- Java does allow multiple inheritance but it is only restricted to interfaces
- Since interfaces do not have method definitions, there is no possibility for ambiguity

```
class MyClass implements Interface1, Interface2 {
        //...
}
```

- In JDK8 Java introduced default methods in interfaces along other features, which can potentially cause ambiguity

Evolving Interfaces

- Suppose we have an Interface `Intf1` that have been used in many classes
- Now we might want to add some a new method for some other implementing classes
- If we change the definition of `Intf1` by adding the method, it will break the code!

Evolving Interfaces

- Suppose we have an Interface `Intf1` that have been used in many classes
- Now we might want to add some a new method for some other implementing classes
- If we change the definition of `Intf1` by adding the method, it will break the code!
- Instead we extend the interface into another interface, `Intf2` say, and use it

```
interface Intf2 extends Intf1 {
        //...
}
```

Anonymous Inner Class

- A class can be nested in another class, and are referred to as inner and outer class respectively
- Similarly an interface can also be nested
- An anonymous type is the one having no name specified

Anonymous Inner Class

- A class can be nested in another class, and are referred to as inner and outer class respectively
- Similarly an interface can also be nested
- An anonymous type is the one having no name specified
- Suppose ABC is an interface (or an abstract class) we may write the following:

```
ABC obj = new ABC() {
        //implementations of all
        //unimplemented methods of ABC
}
```

Anonymous Inner Class

- A class can be nested in another class, and are referred to as inner and outer class respectively
- Similarly an interface can also be nested
- An anonymous type is the one having no name specified
- Suppose ABC is an interface (or an abstract class) we may write the following:

```
ABC obj = new ABC() {
        //implementations of all
        //unimplemented methods of ABC
}
```

- This is a shorthand, Java implicitly creates an anonymous class extending `ABC` [outerClassName+$+serialNumber]
- Must provide definitions for all unimplemented methods of `ABC`
- May also put additional code, or override other methods