CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Package

- A package is a group of classes.
- A package is used to localize the scope of the classes.
  - So the same class name can be used in different packages
- Packages are stored in hierarchical manner using the folder hierarchy of the host machine.
- They are “imported” in a class which makes their classes available to this file.
  - Importing an entire package
    - All the classes are available
    - Note: a public class is available everywhere, whereas a default class, i.e., a class with no access modifier is accessible inside only its own package.
  - Importing a single class from a package: oftentimes we feel no need to “load” all classes, but rather only a single class.

Access Protection Through Package

- We’ve learnt how to maintain access to class members (data, method and nested class)
- Package provides yet another level of access control
  - Whether a class is recognizable in
    - Non-subclass under same package
    - Non-subclass under different package
    - Subclass under same package
    - Subclass under different package
- Notice a similarity between class and package: both provide encapsulation and namespace
  - Classes contain data and methods.
  - Packages contain classes and other sub-packages.

More on Package

- `java.lang` package is automatically imported in every .java file
  - Equivalent to writing the following line: `import java.lang.*;`
  - Contains some frequently-used classes such as `System` so we can use `System.out.println()`.
  - All the standard Java classes are stored in a package called `java`, that's why `java.lang`
- If a package is imported in a class file where another class having the same name exists, the compiler will not raise error message until that class is used in code.
  - At that time we may need to rename the class.
- Import statement is not mandatory to use - we can write the entire package path instead of writing import.
  - `class myclass extends java.util.Date { …`
  - Not handy, however.

Big Picture of Java Code

Level 1: Packages
Level 2: Inside packages: classes, interfaces, enumerations
Level 3: Inside classes: data, methods, nested classes
    Data: static, non-static etc.
    Methods: usual, overloaded, overridden,
    Access modes: apply to almost all constructs: classes, interfaces, data, methods, …

Largely, this picture applies to any OOP code.

New Features of Interface (Since JDK 8) Self-Study

JDK 8 adds a new capability to interface called the default method.

A default method lets you define a default implementation for an interface method.

In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract.

New Features of Interface (Since JDK 8): Example from Textbook

```
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 the default method.
class DefaultMethodDemo {
  public static void main(String[] args) {

    MyIFImp obj = new MyIFImp();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
  }
}
```

```
// Implement MyIF.
class MyIFImp implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

```
class MyIFImp2 implements MyIF {
  // Here, implementations for both getNumber( ) and getString( ) are provided.
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "This is a different string.";
  }
}
```

Problem of Default Implementation

```
interface A {
    default void f() { System.out.println("A: default"); }
}
interface B {
     default void f()     { System.out.println("B: default"); }
}

class myclass implements A, B {
    public void f() { //What happens if myclass does not override f()?
         System.out.println("ab");
    }
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        myclass ob = new myclass();
        ob.f();
    }
}
```

So like many other constructs, default implementation in interface comes with its own problems!

New Features of Interface (Since JDK 8)

In a nutshell

- An abstract class can have both abstract method (i.e., method without body) and normal method and state information (i.e., regular variables).
- A pre-JDK8 interface can only have methods without body and no state information (i.e., no non-static, non-final variable).
- A JDK8+ interface can have both (abstract) method without body, and (default) method with body, but cannot have state information.

Question: When to use what? **Ans.:** Depends on need of the specific problem at hand.

More differences and similarities between abstract class and JDK8+ interface:

Java-8: Interface with default methods vs Abstract class. - Java Interview Questions & Answers
Java67: Difference between Abstract class and Interface in Java 8? Answer
Interface With Default Methods vs Abstract Class | Baeldung

New Features of Interface (Since JDK 8)

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code.

Recall that there must be implementations for all methods defined by an interface.

In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method.

The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided.

Thus, the addition of a default method will not cause pre-existing code to break.

New Features of Interface (Since JDK 8)

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.

For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called `remove()`, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and non-modifiable sequences, then `remove()` is essentially optional because it won't be used by non-modifiable sequences. In the past, a class that implemented a non-modifiable sequence would have had to define an empty implementation of `remove()`, even though it was not needed. Today, a default implementation for `remove()` can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of `remove()`. Thus, by providing a default, the interface makes the implementation of `remove()` by a class optional.

New Features of Interface (Since JDK 8)

It is important to point out that the addition of default methods does not change a key aspect of interface: its inability to maintain state information. An interface still cannot have instance variables.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify what and not how. However, the inclusion of the default method gives you added flexibility, albeit with some caveats.

New Features of Interface (Since JDK 8)

Another capability added to interface by JDK 8 is 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. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.

Static interface methods are not inherited by either an implementing class or a subinterface.

New Features of Interface (Since JDK 9)

Beginning with JDK 9, an interface can include a private method. A private interface method can be called only by a default method or another private method defined by the same interface.

Because a private interface method is specified private, it cannot be used by code outside the interface in which it is defined. This restriction includes subinterfaces because a private interface method is not inherited by a subinterface.

The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication.

Although the private interface method is a feature that you will seldom need, in those cases in which you do need it, you will find it quite useful.

- We have covered what can be called the foundation topics of Java.
- Next topics of this course will require a thorough knowledge of all the previous topics.
  - A topic will require the understanding of multiple basic topics.
  - As mentioned many times, individual basic topics may not sould difficult to grasp, but when it comes to combining their understanding and application, things often veer too much!

A Digression

Now let us discuss the solutions to the Problem 2 of Lab Exam 1 …

This problem shows that how difficult an apparently benign program involving inheritance and superclass variable referring to subclass objects can be.

Bottomline: in real-life, you should not only be syntactically correct, but also need to be aware of the implementation scenarios and real-life situations.

Topics

Exception handling
- Motivation
- Basic constructs
- Exception class hierarchy and exception types
- Multiple catch clauses
- Creating our own exceptions

Exception Handling

- An exception is an abnormal situation that occurs at runtime.
- Exception handling is a mechanism to manage the runtime errors in a controlled way.
  - Keeps the main logic of the code separate from error handling code. That is, it avoids intermingling of these two types of codes.
  - Basic principle: instead of putting error checking code here and there, it provides a systematic way to put the error checking codes.
  - The programmer can create and manage his/her own exceptional situations very easily.
- Offered by almost all OOP languages.
- Take the example of divide by zero error…[EXAMPLE 1]

`try-catch-finally` Statement

```
try {
                // block of code to monitor for errors
}//try ends

catch (ExceptionType1 exOb) {
                // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
                // exception handler for ExceptionType2
}
// ...
finally {
                // block of code to be executed after try block ends
}
```

End of Lecture 12.