2 - Advanced Class Design

< Previous | Next >

§2.1 Inner Classes

 # Definitions:

- A top level class is a class that is not a nested class.
- A nested class is any class whose declaration occurs within the body of another class or interface.
- Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called static nested classes. Non-static nested classes are called inner classes.
- An inner class is a nested class that is not explicitly or implicitly declared static.
- A static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.
- A class defined inside an interface is implicitly static.

 # Static nested class:

- Declaring a nested class static only means that instances of the class are created without having an outer instance. It does not put any limits on whether the members of the class can be static or not.
- Any class can statically import TestClass.TestInner like this: `import static mypack.TestOuter.TestInner;` and then create an instance of TestInner like this: `TestOuter.TestInner ti = new TestInner();`
  `java public class TestOuter { public static class TestInner { } }`

 # Non-static inner class:

- Non static inner classes are allowed to declare static final fields that are compile time constants as members.
- Non-static inner classes can contain final static fields (but not methods).
- Member variables of the outer instance can only be referred to using only the variable name within the inner instance, if that variable is not shadowed by another variable in inner class.
- An object of the outer class can access private variables of the inner class.
- Every non-static inner class object has a reference to its outer class object which can be accessed by doing OuterClass.this.
- If you just refer to a variable of the outer class by its name `i`, the compiler figures out that `i` is actually `Outer.this.i`.
- Inside a non-static inner class, 'InnerClass.this' is equivalent to 'this'.
- You can create an object of the inner class by calling `new Innerclas` direclty. This is equivalent to `this.new TestInner();`. this is implicit.

 # Local class:

- If the inner class is defined in a **static method** of the encapsulating class the following variables are accessible:
  - static variables of the enclosing class
  - (effectively) final local variables of the method
- If the inner class is defined in an **instance method** of the encapsulating class the following variables are accessible:
  - static variables of the enclosing class
  - all instance variables of the enclosing class
  - (effectively) final local variables of the method

 # Anonymous class:

- An anonymous class can be declared in a static method.
- An anonymous class can never be static. Even if created in a static method.
- Anonymous classes are implicitly final.
- If an anonymous class is created for interface, it extends Object class and implement that interface, if it is created for a class then it extends that class.

- Anonymous inner classes can have an initialization parameter. (If the class they extend has a corresponding constructor).
- You cannot pass parameters when you implement an interface by an anonymous class.
- Anonymous classes cannot have explicitly defined constructors, since they have no names.

 # The rest:

- Inner classes can extend the outer class.
- In general, there is no restriction on what a nested class may or may not extend.
- The modifier static pertains only to member classes, not to top level or local or anonymous classes. That is, **only classes declared as members of top-level classes can be declared static**. Package member classes, local classes (i.e. classes declared in methods) and anonymous classes cannot be declared static.
- Note the difference between an inner class and a static nested class. Inner class means a NON STATIC class defined inside a class. Class B can be used in other places: A.B b = new A.B(); There is no outer instance. Class C can only be used like this: A.C c = new A().new C(); Outer instance is needed.
  `public class A { // outer class static public class B {} //Static Nested class. class C {} //Inner class. }`
- Local variables are sometimes called stack, temporary, automatic, or method variables.
- Putting a return type makes private void Outer() { } a method and not a constructor.
- When a programmer does not define ANY constructor, the compiler inserts one automatically, the access modifier of which is same as that of the class.
- Inside an instance methodof TestClass with a non-static inner class A new TestClass.A(); is same as new A().

§2.2 Enumerated types

 # You need to know the following facts about enums:

1. Enum constructor is always private. You cannot make it public or protected. If an enum type has no constructor declarations, then a private constructor that takes no parameters is automatically provided.
2. An enum is implicitly final, which means you cannot extend it.
3. You cannot extend an enum from another enum or class because an enum implicitly extends java.lang.Enum. But an enum can implement interfaces.
4. Since enum maintains exactly one instance of its constants, you cannot clone it. You cannot even override the clone method in an enum because java.lang.Enum makes it final.
5. Compiler provides an enum with two public static methods automatically - values() and valueOf(String). The values method returns an array of its constants and valueOf method tries to match the String argument exactly (i.e. case sensitive) with an enum constant and returns that constant if successful otherwise it throws java.lang.IllegalArgumentException.

 # The following are a few more important facts about java.lang.Enum which you should know:

1. It implements java.lang.Comparable (thus, an enum can be added to sorted collections such as SortedSet, TreeSet, and TreeMap).
2. It has a method ordinal(), which returns the index (starting with 0) of that constant i.e. the position of that constant in its enum declaration.
3. It has a method name(), which returns the name of this enum constant, exactly as declared in its enum declaration.

- A public (or non-public) enum can be defined inside any class.
- An enum can be defined as a static member of any class. You can also have multiple public enums with in the same class.
- An enum cannot be defined inside any method or constructor. **Enum must not be local!**

```
public enum Coffee {
    ESPRESSO("Very Strong"), MOCHA, LATTE;
    public String strength;

    Coffee(String strength) {
        this.strength = strength;
    }
}
```

- The enum definition contains an explicit constructor that takes a parameter. Therefore, the compiler will not provide a default no-args constructor for this enum. Hence, the declaration of MOCHA and LATTE will fail since they need a no-args constructor. It will compile if you add the following constructor: `Coffee(){ }`

- The natural order of enums is the order in which they are defined. It is not necessarily same as alphabetical order of their names.
- Unlike a regular java class, you cannot access a non-final static field from an enum's constructor.

§2.3 Interfaces

- Java 8 requires a static interface method to be invoked using the interface name instead of a reference variable.
- Having ambiguous fields or methods does not cause any problems by itself but referring to such fields/methods in an ambiguous way will cause an error at compile time.
- When interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a throws clause that is compatible with ALL the overridden declarations.
- Remember that the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

```
interface Account {
    public default String getId() {
        return "0000";
    }
}

interface PremiumAccount extends Account {
    public String getId();
}

public class BankAccount implements PremiumAccount {
    public static void main(String[] args) {
        Account acct = new BankAccount();
        System.out.println(acct.getId());
    }
}
```

- Since interface PremiumAccount redeclares getId method as abstract, the BankAccount class must either provide an implementation for this method or be marked as abstract. In this case, making the class abstract will not help because of the statement - Account acct = new BankAccount();
- You cannot use super keyword to call a method defined in the super interface.
- A class (or an interface) can invoke a default method of an interface that is explicitly mentioned in the class's implements clause (or the interface's extends clause) by using the same syntax i.e. .super.. However, this technique cannot be used to invoke a default method provided by an interface that is not directly implemented (or extended) by the caller.
- Java prohibits a class from inheriting multiple implementations of the same method from different unrelated interfaces. However, if a class provides its own implementation of the same method, the ambiguity is removed because the compiler can unambiguously invoke the implementation provided by the class itself.
- A default method of a super interface cannot be overridden by a static method in a sub interface. You can, however, redeclare a static method of a super interface as a default method in the sub interface.
- Interface methods can never be declared final.

§2.4 Abstract Classes

- Valid reasons for creating an abstract class are:
  1. You need to have a root class for a hierarchy of related classes.
  2. You want to pass different implementations for the same abstraction in method calls. (also a valid reason for using an interface)

- Valid reasons for creating an interface are:

  1. The implementation details are not known at the time. This is a valid reason for declaring an interface but not an abstract class. Generally, the purpose of an interface is to identify and declare just the behavior. Actual implementation can come later. While abstract class is used when a common implementation is also identified. In that respect, an abstract class actually provides less abstraction than an interface.
  2. You are only modeling the behavior of an abstract concept.

- You need to create an abstract class if you want to define common method signatures in the class but force subclasses to provide

implementations for such methods. You will make such methods signatures (and therefore the class) abstract to achieve this.

- A class cannot be both final and abstract.
- An abstract class may have final methods.

< Previous | Next >