Chapter 1

Declarations, access control, and enums

Business Training

1

Content

- Java Refresher
- Define Classes
- Declare Interfaces
- Declare Class Members
- Enums

2

Java Refresher

3

Java is an OO language

- A Java program is mostly a collection of *objects* talking to other objects by invoking each other's *methods*. Every object is of a certain *type*, and that type is defined by a *class* or an *interface*. Most Java programs use a collection of objects of many different types.
- Following is a list of a few useful terms for this object-oriented (OO) language:
  - **Class** A template that describes the kinds of state and behavior that objects of its type support.
  - **Object** At runtime, when the Java Virtual Machine (JVM) encounters the new keyword, it will use the appropriate class to make an object that is an instance of that class. That object will have its own *state* and access to all of the behaviors defined by its class.
    - **State (instance variables)** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's *state*.
    - **Behavior (methods)** When a programmer creates a class, she creates methods for that class. Methods are where the class's logic is stored and where the real work gets done. They are where algorithms get executed and data gets manipulated.

4

1Z0-808 Chapter 1

Java is an OO language

- **Identifiers and Keywords** All the Java components we just talked about—classes, variables, and methods—need names. In Java, these names are called *identifiers*, and, as you might expect, there are rules for what constitutes a legal Java identifier.
- **Inheritance** Central to Java and other OO languages is the concept of *inheritance*, which allows code defined in one class or interface to be reused in other classes. In Java, you can define a general (more abstract) *superclass* and then extend it with more specific *subclasses*.
- **Interfaces** A powerful companion to inheritance is the use of interfaces. Interfaces are *usually*like a 100 percent abstract superclass that defines the methods a subclass must support, but not *how* they must be supported.

M. Romdhani, 2020 **5**

5

Define Classes

6

Source File Declaration Rules

- **There can be only one public class per source code file.**
- **Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.**
- **If there is a public class in a file, the name of the file must match the name of the public class.**
- **A file can have more than one non public class.**
- **Files with no public classes can have a name that does not match any of the classes in the file.**
- **If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.**
- **If there are import statements, they must go between the package statement (if there is one) and the class declaration.**
- **import and package statements apply to all classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.**

Class Declarations and Modifiers

- **In general, modifiers fall into two categories:**
  - **Access modifiers** (public, protected, private)
  - **Nonaccess modifiers** (including strictfp, final, and abstract)
- **Access control in Java is a little tricky because there are four access controls (levels of access) but only three access modifiers (private, protected, public).**
  - The fourth access control level (called default or package access) is what you get when you don't use any of the three access modifiers.
    - In other words, every class, method, and instance variable you declare has an access control, whether you explicitly type one or not. Although all four access controls (which means all three modifiers) work for most method and variable declarations, **a class can be declared with only public or default access**; the other two access control levels don't make sense for a class, as you'll see.
    - Note: As of Java 8, the word default can ALSO be used to declare certain methods in interfaces. When used in an interface's method declaration, default has a different meaning than the default access level.

Class Access

- **Default Access**
  - A class with default access has no modifier preceding it in the declaration! A class with default access **can be seen only by classes within the same package**.
- **Public Access**
  - A class declaration with the public keyword gives **all classes from all packages access to the public class**.

Other (Nonaccess) Class Modifiers

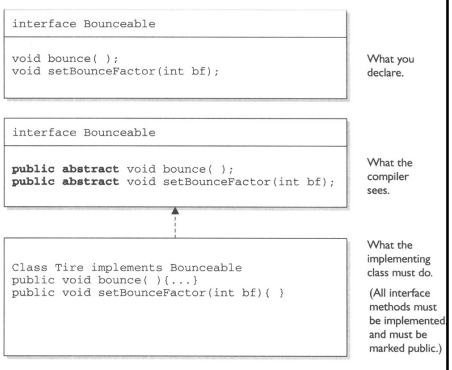
- **You can modify a class declaration using the keyword final, abstract, or strictfp.**
  - Marking a class as **strictfp** means that any method code in the class will conform to the IEEE 754 standard rules for floating points.
  - **Final Classes**
    - When used in a class declaration, the final keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a final class, and any attempts to do so will give you a compiler error.
  - **Abstract Classes**
    - An abstract class can never be instantiated, Its sole purpose is to be extended (subclassed).
- **You can't mark a class as both abstract and final. They have nearly opposite meanings.**

Declare Interfaces

11

Declaring an Interface

- **Think of an interface as a 100-percent abstract class. These rules are strict:**
  - **All interface methods are implicitly public and abstract.** You do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.
  - **All variables defined in an interface must be public, static, and final—**in other words, interfaces can declare only constants, not instance variables.
  - **Interface methods must not be static**.
  - Because interface methods are abstract, they cannot be marked final, strictfp, or native.
  - An interface can extend **one or more other interfaces.**
  - An interface cannot extend anything but another interface.
  - **An interface cannot implement another interface or class.**
  - Interface types can be used polymorphically.



12

Declaring Interface Constants

- **You need to remember one key rule for interface constants. They must always be public static final**
  - Because interface constants are defined in an interface, they don't have to be declared as public, static, or final. They must be public, static, and final, **but you don't have to actually declare them that way**.
    - Example :

```
interface Foo {
        int BAR = 42;
        void go();
}
```

- **For example, the following interface definitions that define constants are correct and arre all identical.**

```
public int x = 1;          // Looks non-static and non-final,
                           // but isn't!
int x = 1;                 // Looks default, non-final,
                           // non-static, but isn't!
static int x = 1;        // Doesn't show final or public
final int x = 1;         // Doesn't show static or public
public static int x = 1;       // Doesn't show final
public final int x = 1;        // Doesn't show static
static final int x = 1         // Doesn't show public
public static final int x = 1;  // what you get implicitly
```


Declaring default Interface Methods

- As of Java 8, interfaces can include inheritable* methods with concrete implementations. For now we'll just cover the simple declaration rules:
  - default methods are declared by using the **default** keyword. The default keyword can be used only with interface method signatures, not class method signatures.
  - default methods are **public by definition**, and **the public modifier is optional**.
  - default methods **cannot be marked as private, protected, static, final, or abstract**.
  - default methods must have a concrete method body.
- **Here are some examples of legal and illegal default methods:**

```
interface TestDefault {
  default int m1(){return 1;} // legal
  public default void m2(){;} // legal
  static default void m3(){;} // illegal: default cannot be marked static
  default void m4();          // illegal: default must have a method body
}
```

1Z0-808 Chapter 1

Declaring static Interface Methods

- As of Java 8, interfaces can include **static** methods with concrete implementations.
  - static interface methods are declared by using the static keyword.
  - static interface methods are public by default, and the public modifier is optional.
  - static interface methods cannot be marked as private, protected, final, or abstract.
  - static interface methods must have a concrete method body.
  - When invoking a static interface method, the method's type (interface name) MUST be included in the invocation.
- **Here are some examples of legal and illegal static interface methods and their use:**

M. Romdhani, 2020 **15**

15

1Z0-808 Chapter 1

Declaring static Interface Methods

```
interface StaticIface {
  static int m1(){ return 42; }      // legal
  public static void m2(){ ; }       // legal
  // final static void m3(){ ; }     // illegal: final not allowed
  // abstract static void m4(){ ; }  // illegal: abstract not allowed
  // static void m5();               // illegal: needs a method body
}

public class TestSIF implements StaticIface {
  public static void main(String[] args) {
    System.out.println(StaticIface.m1());   // legal: m1()'s type
                                            // must be included
    new TestSIF().go();
    // System.out.println(m1());      // illegal: reference to interface
                                      // is  required
  }
  void go() {
    System.out.println(StaticIface.m1());  // also legal from an instance
  }
}
```

- which produces this output:

  42

  42

M. Romdhani, 2020 **16**

16

Declare Class Members

17

Access Modifiers

- **Members can use all four:**
  - public , protected , default, private
  - Public Members
    - When a method or variable member is declared public, it means all other classes, regardless of the package they belong to, can access the member (assuming the class itself is visible).
  - Private Members
    - Members marked private can't be accessed by code in any class other than the class in which the private member was declared.
  - Protected and Default Members
    - The protected and default access control levels are almost identical, but with one critical difference. **A default member may be accessed only if the class accessing the member belongs to the same package, whereas a protected member can be accessed (through inheritance) by a subclass even if the subclass is in a different package**. **Local Variables and Access Modifiers**
    - Can access modifiers be applied to local variables? NO!
- **Local Variables and Access Modifiers**
  - Can access modifiers be applied to local variables? NO!

18

Nonaccess Member Modifiers

- **Final Methods**
  - The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method.
  - Final Arguments : A final argument must keep the same value that the parameter had when it was passed into the method.
- **Abstract Methods**
  - You mark a method abstract when you want to force subclasses to provide the implementation.
- **Synchronized Methods**
  - The synchronized keyword indicates that a method can be accessed by only one thread at a time.
- **Native Methods**
  - The native modifier indicates that a method is implemented in platform-depen dent code, often in C.
- **Strictfp Methods**
  - strictfp forces floating points to adhere to the IEEE 754 standard
- **Methods with Variable Argument Lists (var-args)**
  - Java 5 allows you to create methods that can take a variable number of arguments.
  - **The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.**

Constructor Declarations

- **Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you.**
- **They must have the same name as the class in which they are declared**
- **Constructor declarations can however have all of the normal access modifiers, and they can take arguments (including var-args), just like methods.**
- **Constructors can't be marked static (they are after all associated with object instantiation), they can't be marked final or abstract (because they can't be overridden).**

Enums

21

Declaring Enums

- **The basic components of an enum are its constants**
  - enum Seasons { Automn, Winter, Spring, Summer};
- **Enums can be declared as their own separate class, or as a class member, however they must not be declared within a method !**
- **So what gets created when you make an enum?**
  - The most important thing to remember is that enums are not Strings or ints! Each of the enumerated *Seasons* types are actually instances of *Seasons* .
  - Think of an enum as a kind of class
- **Declaring Constructors, Methods, and Variables in an Enum**
  - You can add constructors, instance variables, methods, and something really strange known as a *constant specific class body*.

22

Declaring Enums

```
enum CoffeeSize {
    BIG(8), HUGE(10), OVERWHELMING(16);

   CoffeeSize(int ounces) {
       this.ounces = ounces; // assign the value to an instance variable
   }
   private int ounces;      // an instance variable each enum
   public int getOunces() {
    return ounces;
   }
}

class Coffee {
  CoffeeSize size;    // each instance
                      // of Coffee has-aCoffeeSize enum
public static void main(String[] args) {
    Coffee drink1 = new Coffee();
    drinkI.size = CoffeeSize.BIG;

    Coffee drink2 = new Coffee();
    drink2.size = CoffeeSize. OVERWHELMING;

    System.out.println(drinkI.size.getOunces());   // prints 8
    System.out.println(drink2.size. getOunces());  // prints 16
  }
}
```


Enums Constructors

- **The key points to remember about enum constructors are**
  - You can **NEVER** invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you define after the constant value.
    - For example, BIG(8) invokes the CoffeeSize constructor that takes an int, passing the int literal 8 to the constructor. (Behind the scenes, of course, you can imagine that BIG is also passed to the constructor, but we don't have to know—or care—about the details.)
  - You can define more than one argument to the constructor, and you can overload the enum constructors, just as you can overload a normal class constructor.
    - We discuss constructors in much more detail in Chapter 2. To initialize a CoffeeType with both the number of ounces and, say, a lid type, you'd pass two arguments to the constructor as BIG(8, "A"), which means you have a constructor in CoffeeSize that takes both an int and a string.

Enums – Some hints

- **This wont compile without a no-arg constructor defined:**

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

- **Enum constants (here, DOG, CAT, and FISH) must be declared before anything else.**

```
enum Pets {
  String name;
  DOG("D"), CAT("C"), FISH("F"); // BAD placement
  Pets(String s) { name = s;}
}
```

- **Unlike a regular java class, you cannot access a non-final static field from an enum's constructor.**

```
enum Pets {
  DOG("D"), CAT("C"), FISH("F");
  static String prefix = "I am ";
  String name;
  Pets(String s) { name = prefix + s;}  // BAD
  public String getData(){ return name; }
}
```


Enums – Some hints

- **enum constructors must be private, meaning a protected constructor does not compile.**
  - A package constructor is "assumed" to be private, and therefore effectively private
- **enums cannot extend other enums**
- **An enum cannot be marked abstract, nor can any of its values, but its methods can be marked abstract**

```
public enum Level {
  HIGH{
     @Override
     public String asLowerCase() {
        return HIGH.toString().toLowerCase();
     }
  },
LOW{
     @Override
     public String asLowerCase() {
        return LOW.toString().toLowerCase();
     }
  };
  public abstract String asLowerCase();
}
```

Enums – Some hints

- **Enum Implementing Interface**

```
public enum EnumImplementingInterface implements MyInterface {
   FIRST("First Value"), SECOND("Second Value");
   private String description = null;

   private EnumImplementingInterface(String desc){
      this.description = desc;
   }
   @Override
   public String getDescription() {
      return this.description;
 }}
```

- **Enum values in Switch/case**

```
enum DaysOff { Thanksgiving, Christmans, Easter }
class Main {
  public static void main(String... args) {
    final DaysOff input = DaysOff.Thanksgiving;
    switch ( input) {
      default:
      case DaysOff.Thanksgiving: // Compilation error: amust be the unqualified name of an enumeration constant
        System.out.println("Thanks");
    }}}
```