Chapter 12 (ocp) - Java Fundamentals

Writing *final* Methods

What happens if a method is marked both `abstract` and `final`? Well, that's like saying, "I want to declare a method that someone else will provide an implementation for, while also telling that person that they are not allowed to provide an implementation." For this reason, the compiler does not allow it.

Working with Enums

In programming, it is common to have a type that can only have a finite set of values, such as days of the week, seasons of the year, primary colors, etc. An *enumeration* is like a fixed set of constants. In Java, an *enum*, short for "enumerated type," can be a top-level type like a class or interface, as well as a nested type like an inner class.

An enum provides a `values()` method to get an array of all of the values. You can use this like any normal array, including in an enhanced `for` loop, often called a *for-each loop*.

```
for(Season season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

Another useful feature is retrieving an enum value from a `String` using the `valueOf()` method. This is helpful when working with older code. The `String` passed in must match the enum value exactly, though.

```
Season s = Season.valueOf("SUMMER"); // SUMMER

Season t = Season.valueOf("summer"); // Throws an exception at runtime
```

Using Enums in Switch Statements

Enums can be used in `switch` statements. Pay attention to the `case` values in this code:

```
Season summer = Season.SUMMER;
switch (summer) {
   case WINTER:
      System.out.println("Get out the sled!");
      break;
   case SUMMER:
      System.out.println("Time for the pool!");
      break;
   default:
      System.out.println("Is it summer yet?");
}
```

The code prints `"Time for the pool!"` since it matches `SUMMER`. In each `case` statement, we just typed the value of the enum rather than writing `Season.WINTER`. After all, the compiler already knows that the only possible matches can be enum values. Java treats the enum type as implicit. In fact, if you were to type `case Season.WINTER`, it would not compile.

```
public enum Season {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   FALL {
      public String getHours() { return "9am-5pm"; }
   };
   public abstract String getHours();
}
```

What's going on here? It looks like we created an `abstract` class and a bunch of tiny subclasses. In a way we did. The enum itself has an `abstract` method. This means that each and every enum value is required to implement this method. If we forget to implement the method for one of the values, then we get a compiler error

Creating Nested Classes

A *nested class* is a class that is defined within another class. A nested class can come in one of four flavors.

- *Inner class*: A non- `static` type defined at the member level of a class
- *Static nested class:* A `static` type defined at the member level of a class
- *Local class*: A class defined within a method body
- *Anonymous class*: A special case of a local class that does not have a name

*interfaces and enums can be declared as both inner classes and `static` nested classes, but not as local or anonymous classes.

Declaring an Inner Class

An *inner class*, also called a *member inner class*, is a non- `static` type defined at the member level of a class (the same level as the methods, instance variables, and constructors). Inner classes have the following properties:

- Can be declared `public`, `protected`, package-private (default), or `private`
- Can extend any class and implement interfaces
- Can be marked `abstract` or `final`
- Cannot declare `static` fields or methods, except for `static final` fields
- Can access members of the outer class including `private` members

```
public class Outer {
2:     private String greeting = "Hi";
3:
4:     protected class Inner {
5:        public int repeat = 3;
6:        public void go() {
7:           for (int i = 0; i < repeat; i++)
8:              System.out.println(greeting);
9:        }
10:    }
11:
12:    public void callInner() {
13:       Inner inner = new Inner();
14:       inner.go();
15:    }
16:    public static void main(String[] args) {
17:       Outer outer = new Outer();
18:       outer.callInner();
19: } }
```

There is another way to instantiate `Inner` that looks odd at first. OK, well maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:    public static void main(String[] args) {
21:       Outer outer = new Outer();
22:       Inner inner = outer.new Inner(); // create the inner class
23:       inner.go();
24:    }
```

Inner classes can have the same variable names as outer classes, making scope a little tricky. There is a special way of calling `this` to say which variable you want to access. This is something you might see on the exam but ideally not in the real world.

In fact, you aren't limited to just one inner class. Please never do this in code you write. Here is how to nest multiple classes and access a variable with the same name in each:

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);        // 30
9:              System.out.println(this.x);   // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

Creating a *static* Nested Class

A *static nested class* is a `static` type defined at the member level. Unlike an inner class, a `static` nested class can be instantiated without an instance of the enclosing class. The trade-off, though, is it can't access instance variables or methods in the outer class directly. It can be done but requires an explicit reference to an outer class variable.

In other words, it is like a top-level class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it.
- It can be made `private` or use one of the other access modifiers to encapsulate it.
- The enclosing class can refer to the fields and methods of the `static` nested class.

Let's take a look at an example:

```
1: public class Enclosing {
```

```
2:    static class Nested {
3:       private int price = 6;
4:    }
5:    public static void main(String[] args) {
6:       Nested nested = new Nested();
7:       System.out.println(nested.price);
8: } }
```

Line 6 instantiates the nested class. Since the class is `static`, you do not need an instance of `Enclosing` to use it. You are allowed to access `private` instance variables, which is shown on line 7.

Writing a Local Class

A *local class* is a nested class defined within a method. Like local variables, a local class declaration does not exist until the method is invoked, and it goes out of scope when the method returns. This means you can create instances only from within the method. Those instances can still be returned from the method.

Local classes have the following properties:

- They do not have an access modifier.
- They cannot be declared `static` and cannot declare `static` fields or methods, except for `static final` fields.
- They have access to all fields and methods of the enclosing class (when defined in an instance method).
- They can access local variables if the variables are `final` or effectively final.

Ready for an example? Here's a complicated way to multiply two numbers:

```
1:  public class PrintNumbers {
2:     private int length = 5;
3:     public void calculate() {
4:        final int width = 20;
5:        class MyLocalClass {
6:           public void multiply() {
7:              System.out.print(length * width);
8:           }
9:        }
10:       MyLocalClass local = new MyLocalClass();
11:       local.multiply();
12:    }
13:    public static void main(String[] args) {
14:       PrintNumbers outer = new PrintNumbers();
15:       outer.calculate();
16:    }
17: }
```

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. Let's talk about that now. The compiler is generating a `.class` file from your local class. A separate class has no way to refer to local variables. If the local variable is `final`, Java can handle it by passing it to the constructor of the local class or by storing it in the `.class` file. If it weren't effectively final, these tricks wouldn't work because the value could change after the copy was made.

An *anonymous class* is a specialized form of a local class that does not have a name. It is declared and instantiated all in one statement using the `new` keyword, a type name with parentheses, and a set of braces `{}`. **Anonymous classes are required to extend an existing**

class or implement an existing interface. They are useful when you have a short implementation that will not be used anywhere else. Here's an example:

```
1:  public class ZooGiftShop {
2:     abstract class SaleTodayOnly {
3:        abstract int dollarsOff();
4:     }
5:     public int admission(int basePrice) {
6:        SaleTodayOnly sale = new SaleTodayOnly() {
7:           int dollarsOff() { return 3; }
8:        };  // Don't forget the semicolon!
9:        return basePrice - sale.dollarsOff();
10: } }
```

Pay special attention to the **semicolon on line 8**. We are declaring a local variable on these lines. Local variable declarations are required to end with semicolons, just like other Java statements—even if they are long and happen to contain an anonymous class.

But what if we want to implement both an `interface` and extend a class? You can't with an anonymous class, unless the class to extend is `java.lang.Object`.

There is one more thing that you can do with anonymous classes. You can define them right where they are needed, even if that is an argument to another method.

```
1:  public class ZooGiftShop {
2:     interface SaleTodayOnly {
3:        int dollarsOff();
4:     }
5:     public int pay() {
6:        return admission(5, new SaleTodayOnly() {
7:           public int dollarsOff() { return 3; }
8:        });
9:     }
10:    public int admission(int basePrice, SaleTodayOnly sale) {
11:       return basePrice - sale.dollarsOff();
12: }}
```

You can even define anonymous classes outside a method body. The following may look like we are instantiating an interface as an instance variable, but the `{}` after the interface name indicates that this is an anonymous inner class implementing the interface.

```
public class Gorilla {
   interface Climb {}
   Climb climbing = new Climb() {};
}
```

Reviewing Nested Classes

For the exam, make sure that you know the information in Table 12.1 and Table 12.2 about which syntax rules are permitted in Java.

TABLE 12.1 Modifiers in nested classes

Permitted Modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`Final`	Yes	Yes	Yes	No

TABLE 12.2 Members in nested classes

Permitted Members	Inner class	`static` nested class	Local class	Anonymous class
Instance methods	Yes	Yes	Yes	Yes
Instance variables	Yes	Yes	Yes	Yes
`static` methods	No	Yes	No	No
`static` variables	Yes (if `final`)	Yes	Yes (if `final`)	Yes (if `final`)

	Inner class	`static` nested class	Local class	Anonymous class
Can extend any class or implement any number of interfaces	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class without a reference	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Understanding Interface Members

When Java was first released, there were only two types of members an interface declaration could include: abstract methods and constant (`static final`) variables. Since Java 8 and 9 were released, four new method types have been added that we will cover in this section. Keep Table 12.4 handy as we discuss the various interface types in this section.

TABLE 12.4 Interface member types

	Since Java version	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	1.0	Class	—	`public` `static` `final`	Yes
Abstract method	1.0	Instance	—	`public` `abstract`	No
Default method	`8`	Instance	`default`	`public`	Yes
Static method	`8`	Class	`static`	`public`	Yes
Private method	`9`	Instance	`private`	—	Yes
Private static method	`9`	Class	`private` `static`	—	Yes

Relying on a *default* Interface Method

A *default method* is a method defined in an interface with the `default` keyword and includes a method body. Contrast `default` methods with abstract methods in an interface, which do not define a method body.

A `default` method may be overridden by a class implementing the interface. The name *default* comes from the concept that it is viewed as an abstract interface method with a default implementation. The class has the option of overriding the `default` method, but if it does not, then the default implementation will be used.

The following is an example of a `default` method defined in an interface:

```
public interface IsWarmBlooded {
   boolean hasScales();
   default double getTemperature() {
      return 10.0;
   }
}
```

Default Interface Method Definition Rules

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is assumed to be `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

Inheriting Duplicate default Methods

We have one last rule for `default` methods that warrants some discussion. You may have realized that by allowing `default` methods in interfaces, coupled with the fact that a class may implement multiple interfaces, Java has essentially opened the door to multiple inheritance problems. For example, what value would the following code output?

```java
public interface Walk {
   public default int getSpeed() { return 5; }
}

public interface Run {
   public default int getSpeed() { return 10; }
}

public class Cat implements Walk, Run {  // DOES NOT COMPILE
   public static void main(String[] args) {
      System.out.println(new Cat().getSpeed());
   }
}
```

In this example, `Cat` inherits the two `default` methods for `getSpeed()`, so which does it use? Since `Walk` and `Run` are considered siblings in terms of how they are used in the `Cat` class, it is not clear whether the code should output `5` or `10`. In this case, Java throws up its hands and says “Too hard, I give up!” and fails to compile.

If a class implements two interfaces that have `default` methods with the same method signature, the compiler will report an error. This rule holds true even for abstract classes because the duplicate method could be called within a concrete method within the abstract class. All is not lost, though. If the class implementing the interfaces *overrides* the duplicate `default` method, then the code will compile without issue.

```java
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   }

   public static void main(String[] args) {
      System.out.println(new Cat().getWalkSpeed());
   }
}
```

In this example, we first use the interface name, followed by the `super` keyword, followed by the `default` method we want to call. We also put the call to the inherited `default` method inside the instance method `getWalkSpeed()`, as `super` is not accessible in the `main()` method.

Using *static* Interface Methods

If you've been using an older version of Java, you might not be aware that Java now supports `static` interface methods. These methods are defined explicitly with the `static` keyword and for the most part behave just like `static` methods defined in classes.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is assumed to be `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

Java “solved” the multiple inheritance problem of `static` interface methods by not allowing them to be inherited. For example, a class that implements two interfaces containing `static` methods with the same signature will still compile. Contrast this with the behavior you saw for `default` interface methods in the previous section.

Introducing *private* Interface Methods

`private` interface methods can be used to reduce code duplication.

Private Interface Method Definition Rules

1. A `private` interface method must be marked with the `private` modifier and include a method body.
2. A `private` interface method may be called only by `default` and `private` (non- `static`) methods within the interface definition.

By introducing six different interface member types, Java has certainly blurred the lines between an abstract class and an interface. A key distinction, though, is that interfaces do not implement constructors and are not part of the class hierarchy. While a class can implement multiple interfaces, it can only directly extend a single class.

Introducing Functional Programming

Functional interfaces are used as the basis for lambda expressions in functional programming. A *functional interface* is an interface that contains a single abstract method. Your friend Sam can help you remember this because it is officially known as a *single abstract method (SAM)* rule.

A *lambda expression* is a block of code that gets passed around, sort of like an anonymous class that defines one method. As you'll see in this section, it can be written in a variety of short or long forms.

Declaring a Functional Interface with *Object* Methods

As you may remember from your previous studies, all classes inherit certain methods from `Object`. For the exam, you should be familiar with the following `Object` method declarations:

- `String toString()`
- `boolean equals(Object)`
- `int hashCode()`

We bring this up now because there is one exception to the single abstract method rule that you should be familiar with. If a functional interface includes an abstract method with the same signature as a `public` method found in `Object`, then those methods do not count toward the single abstract method test. The motivation behind this rule is that any class that implements the interface will inherit from `Object`, as all classes do, and therefore always implement these methods.

*Since Java assumes all classes extend from `Object`, you also cannot declare an interface method that is incompatible with `Object`. For example, declaring an abstract method `int toString()` in an interface would not compile since `Object`'s version of the method returns a `String`.

OVERRIDING TOSTRING(), EQUALS(OBJECT), AND HASHCODE()

While knowing how to properly override `toString()`, `equals(Object)`, and `hashCode()` was part of Java certification exams prior to Java 11, this requirement was removed on all of the Java 11 exams. As a professional Java developer, it is important for you to know at least the basic rules for overriding each of these methods.

- `toString()`: The `toString()` method is called when you try to print an object or concatenate the object with a `String`. It is commonly overridden with a version that prints a unique description of the instance using its instance fields.
- `equals(Object)`: The `equals(Object)` method is used to compare objects, with the default implementation just using the `==` operator. You should override the `equals(Object)` method anytime you want to conveniently compare elements for equality, especially if this requires checking numerous fields.
- `hashCode()`: Any time you override `equals(Object)`, you must override `hashCode()` to be consistent. This means that for any two objects, if `a.equals(b)` is `true`, then `a.hashCode()==b.hashCode()` must also be `true`. If they are not consistent, then this could lead to invalid data and side effects in hash-based collections such as `HashMap` and `HashSet`.

All of these methods provide a default implementation in `Object`, but if you want to make intelligent use out of them, then you should override them.

Implementing Functional Interfaces with Lambdas

In addition to functional interfaces you write yourself, Java provides a number of predefined ones. You'll learn about many of these in Chapter 15, “Functional Programming.” For now, let's work with the `Predicate` interface. Excluding any `static` or `default` methods defined in the interface, we have the following:

```java
public interface Predicate<T> {
   boolean test(T t);
}
```

We'll review generics in Chapter 14, “Generics and Collections,” but for now you just need to know that `<T>` allows the interface to take an object of a specified type. Now that we have a functional interface, we'll show you how to implement it using a lambda expression. The relationship between functional interfaces and lambda expressions is as follows: *any functional interface can be implemented as a lambda expression*.

Even older Java interfaces that pass the single abstract method test are functional interfaces, which can be implemented with lambda expressions.

Let's try an illustrative example. Our goal is to print out all the animals in a list according to some criteria. We start out with the `Animal` class.

```java
public class Animal {
   private String species;
   private boolean canHop;
   private boolean canSwim;
   public Animal(String speciesName, boolean hopper, boolean swimmer) {
      species = speciesName;
      canHop = hopper;
      canSwim = swimmer;
   }
   public boolean canHop()  { return canHop; }
   public boolean canSwim() { return canSwim; }
   public String toString() { return species; }
}
```

The `Animal` class has three instance variables, which are set in the constructor. It has two methods that get the state of whether the animal can hop or swim. It also has a `toString()` method so we can easily identify the `Animal` in programs.

Now we have everything that we need to write our code to find each `Animal` that hops.

```
1:  import java.util.*;
2:  import java.util.function.Predicate;
3:  public class TraditionalSearch {
4:     public static void main(String[] args) {
5:
6:        // list of animals
7:        var animals = new ArrayList<Animal>();
8:        animals.add(new Animal("fish",     false, true));
9:        animals.add(new Animal("kangaroo", true,  true));
10:       animals.add(new Animal("rabbit",   true,  false));
11:       animals.add(new Animal("turtle",   false, true));
12:
13:       // Pass lambda that does check
14:       print(animals, a -> a.canHop());
15:    }
16:    private static void print(List<Animal> animals,
17:       Predicate<Animal> checker) {
18:       for (Animal animal : animals) {
19:          if (checker.test(animal))
20:             System.out.print(animal + " ");
21:       }
22:    }
23: }
```

This program compiles and prints `kangaroo rabbit` at runtime. The `print()` method on line 14 method is very general—it can check for any trait. This is good design. It shouldn't need to know what specifically we are searching for in order to print a list of animals.

Now what happens if we want to print the `Animal`s that swim? We only have to add one line of code—no need for an extra class to do something simple. Here's that other line:

```
14:       print(animals, a -> a.canSwim());
```

This prints `fish kangaroo turtle` at runtime. How about `Animal`s that cannot swim?

```
14:       print(animals, a -> !a.canSwim());

```

This prints `rabbit` by itself. The point here is that it is really easy to write code that uses lambdas once you get the basics in place.

Lambda expressions rely on the notion of deferred execution. Deferred execution means that code is specified now but runs later. In this case, later is when the print() method calls it. Even though the execution is deferred, the compiler will still validate that the code syntax is correct.

Writing Lambda Expressions

The syntax of lambda expressions is tricky because many parts are optional. Despite this, the overall structure is the same. The left side of the lambda expression lists the variables. It must be compatible with the type and number of input parameters of the functional interface's single abstract method.

The right side of the lambda expression represents the body of the expression. It must be compatible with the return type of the functional interface's abstract method. For example, if the abstract method returns `int`, then the lambda expression must return an `int`, a value that can be implicitly cast to an `int`, or throw an exception.

Let's take a look at a functional interface in both its short and long forms. Figure 12.1 shows the short form of this functional interface and has three parts:

- A single parameter specified with the name `a`
- The arrow operator to separate the parameter and body
- A body that calls a single method and returns the result of that method

FIGURE 12.1 Lambda syntax omitting optional parts

Now let's look at a more verbose version of this lambda expression, shown in Figure 12.2. It also contains three parts.

- A single parameter specified with the name `a` and stating the type is `Animal`
- The arrow operator to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a `return` statement

FIGURE 12.2 Lambda syntax, including optional parts

The parentheses can be omitted only if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way so they can do as little typing as possible.

It shouldn't be news to you that we can omit braces when we have only a single statement. We did this with `if` statements and loops already. What is different here is that the rules change when you omit the braces. Java doesn't require you to type `return` or use a semicolon when no braces are used. This special shortcut doesn't work when we have two or more statements. At least this is consistent with using `{}` to create blocks of code elsewhere.

As a fun fact, s -> {} is a valid lambda. If the return type of the functional interface method is void, then you don't need the semicolon or return statement.

Let's take a look at some examples. The following are all valid lambda expressions, assuming that there are functional interfaces that can consume them:

```
() -> new Duck()
d -> {return d.quack();}
(Duck d) -> d.quack()
(Animal a, Duck d) -> d.quack()
```

The first lambda expression could be used by a functional interface containing a method that takes no arguments and returns a `Duck` object. The second and third lambda expressions both can be used by a functional interface that takes a `Duck` as input and returns whatever the return type of `quack()` is. The last lambda expression can be used by a functional interface that takes as input `Animal` and `Duck` objects and returns whatever the return type of `quack()` is.

Now let's make sure you can identify invalid syntax. Let's assume we needed a lambda that returns a `boolean` value. Do you see what's wrong with each of these?

```
3: a, b -> a.startsWith("test")         // DOES NOT COMPILE
4: Duck d -> d.canQuack();              // DOES NOT COMPILE
5: a -> { a.startsWith("test"); }       // DOES NOT COMPILE
6: a -> { return a.startsWith("test") } // DOES NOT COMPILE
7: (Swan s, t) -> s.compareTo(t) != 0   // DOES NOT COMPILE
```

Lines 3 and 4 require parentheses around each parameter list. Remember that the parentheses are optional *only* when there is one parameter and it doesn't have a type declared. Line 5 is missing the `return` keyword, which is required since we said the lambda must return a `boolean`. Line 6 is missing the semicolon inside of the braces, `{}`. Finally, line 7 is missing the parameter type for `t`. If the parameter type is specified for one of the parameters, then it must be specified for all of them.

Working with Lambda Variables

Variables can appear in three places with respect to lambdas: the parameter list, local variables declared inside the lambda body, and variables referenced from the lambda body. All three of these are opportunities for the exam to trick you.

Parameter List

Earlier you learned that specifying the type of parameters is optional. Now `var` can be used in a lambda parameter list. That means that all three of these statements are interchangeable:

```
Predicate<String> p = x -> true;
Predicate<String> p = (var x) -> true;
Predicate<String> p = (String x) -> true;
```

Restrictions on Using var in the Parameter List

While you can use `var` inside a lambda parameter list, there is a rule you need to be aware of. If `var` is used for one of the types in the parameter list, then it must be used for all parameters in the list.