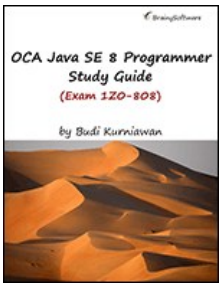

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 7: Interfaces, Abstract Classes and Polymorphism

Overview

Java beginners often get the impression that an interface is simply a class without implementation. While this is not technically incorrect, it obscures the real purpose of having the interface in the first place. The interface is more than that. The interface should be regarded as a contract between a service provider and its clients. This chapter therefore focuses on the concepts before explaining how to write an interface.

The second topic in this chapter is the abstract class. Technically speaking, an abstract class is a class that cannot be instantiated and must be implemented by a subclass. However, the abstract class is important because in some situations it can take the role of the interface. You will learn how to use the abstract class too in this chapter.

Finally, this chapter discusses polymorphism, one of the main pillars in object-oriented programming.

7.1 The Concept of Interface

When learning about the interface for the first time, novices often focus on how to write one, rather than understanding the concept behind it. They would think an interface is something like a Java class declared with the **interface** keyword and whose methods have no body.

While the description is not inaccurate, treating an interface as an implementation-less class misses the big picture. A better definition of an interface is a contract. It is a contract between a service provider (server) and the user of such a service (client). Sometimes the server defines the contract, sometimes the client does.

Consider this real-world example. Microsoft Windows is the most popular operating system today, but Microsoft does not make printers. For printing, you still rely on those people at HP, Canon, Samsung, and the like. Each of these printer makers uses a proprietary technology. However, their products can all be used to print documents from any Windows application. How come?

This is because Microsoft said something to this effect to the printer manufacturers, "If you want your products useable on Windows (and we know you do), you must implement this **Printable** interface."

The interface is as simple as this:

```
interface Printable {

     void print(Document document);
}
```

where *document* is the document to be printed.

Implementing this interface, the printer makers then write printer drivers. Every printer has a different driver, but they all implement **Printable**. A printer driver is an implementation of **Printable**. In this case, these printer drivers are the service provider.

The client of the printing service is all Windows applications. It is easy to print on Windows because an application just needs to call the **print** method and pass a **Document** object. Because the interface is freely available, client applications can be compiled without waiting for an implementation to be available.

The point is, printing to different printers from different applications is possible thanks to the **Printable** interface. This interface is a contract between printing service providers and printing clients.

An interface can define both fields and methods. Prior to JDK 1.8 all methods in an interface were abstract, but starting from JDK 1.8 you can also write default and static methods in an interface. Unless specified otherwise, an interface method refers to an abstract method.

To be useful, an interface has to have an implementing class that actually performs the action.

Figure 7.1 illustrates the **Printable** interface and its implementation in an UML class diagram.

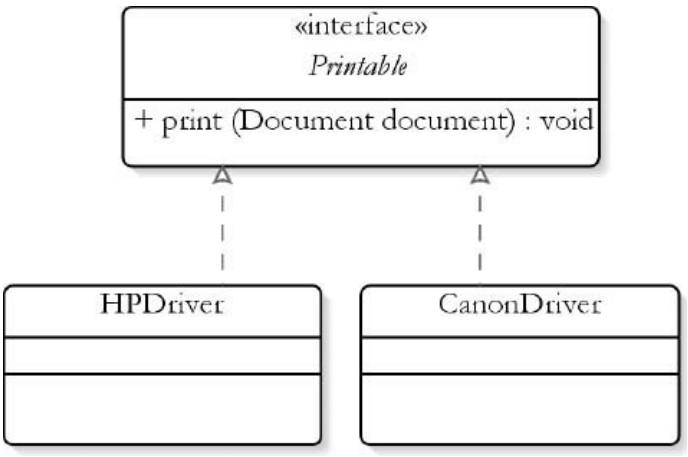

Figure 7.1: An interface and two implementation classes in a class diagram

In the class diagram, an interface has the same shape as a class, however the name is printed in italic and prefixed with <<interface>>. The **HPDriver** and **CanonDriver** classes are classes that implement the **Printable** interface. The implementations are of course different. In the **HPDriver** class, the **print** method contains code that enables printing to a HP printer. In **CanonDriver**, the code enables printing to a Canon driver. In a UML class diagram, a class and an interface are joined by a dash-line with an arrow. This type of relationship is often called realization because the class provides real implementation (code that actually works) of the abstraction provided by the interface.

Note This case study is contrived but the problem and the solution are real. I hope this provides you with more understanding of what the interface really is. It is a contract.

7.2 The Interface, Technically Speaking

Now that you understand what the interface is, let's examine how you can create one. In Java, like the class, the interface is a type. Follow this format to write an interface:

```
accessModifier interface interfaceName {

}
```

Like a class, an interface has either the public or default access level. An interface can have fields and methods. All members of an interface are implicitly public. Listing 7.1 shows an interface named **Printable**.

Listing 7.1: The Printable interface

```
package app07;
public interface Printable {
     void print(Object o);
}
```

The **Printable** interface has a method, **print**. Note that **print** is public even though there is no **public** keyword in front of the method declaration. You are free to use the keyword **public** before the method signature, but it would be redundant.

Just like a class, an interface is a template for creating objects. Unlike an ordinary class, however, an interface cannot be instantiated. It simply defines a set of methods that Java classes can implement.

You compile an interface just you would a class. The compiler creates a .class file for each interface compiled successfully.

To implement an interface, you use the **implements** keyword after the class declaration. A class can implement multiple interfaces. For example, Listing 7.2 shows the **CanonDriver** class that implements **Printable**.

Listing 7.2: An implementation of the Printable interface

```
package app07;
public class CanonDriver implements Printable {
    @Override
    public void print(Object obj) {
        // code that does the printing
    }
```

```
}
```

Note that a method implementation should also be annotated with **@Override**.

Unless specified otherwise, all interface methods are abstract. An implementing class has to override all abstract methods in an interface. The relationship between an interface and its implementing class can be likened to a parent class and a subclass. An instance of the class is also an instance of the interface. For example, the following **if** statement evaluates to **true**.

```
CanonDriver driver = new CanonDriver();
if (driver instanceof Printable)     // evaluates to true
```

Some interfaces have neither fields nor methods, and are known as marker interfaces. Classes implement them as a marker. For example, the **java.io.Serializable** interface, has no fields nor methods. Classes implement it so that their instances can be serialized, i.e. saved to a file or to memory.

Fields in an Interface

Fields in an interface must be initialized and are implicitly public, static and final. However, you may redundantly use the modifiers **public, static**, and **final**. These lines of code have the same effect.

```
public int STATUS = 1;
int STATUS = 1;
public static final STATUS = 1;
```

Note that by convention field names in an interface are written in upper case.

It is a compile error to have two fields with the same name in an interface. However, an interface might inherit more than one field with the same name from its superinterfaces.

Abstract Methods

You declare abstract methods in an interface just as you would declare a method in a class. However, abstract methods in an interface do not have a body, they are immediately terminated by a semicolon. All abstract methods are implicitly public and abstract, even though it is legal to have the **public** and **abstract** modifiers in front of a method declaration.

The syntax of an abstract method in an interface is

```
[methodModifiers] ReturnType MethodName(listOfArgument)
        [ThrowClause];
```

where *methodModifiers* is **abstract** and **public**.

Extending An Interface

The interface supports inheritance. An interface can extend another interface. If interface **A** extends interface **B, A** is said to be a subinterface of **B. B** is the superinterface of **A**. Because **A** directly extends **B, B** is the direct superinterface of **A**. Any interfaces that extend **B** are indirect subinterfaces of **A**. Figure 7.2 shows an interface that extends another interface. Note that the type of the line connecting both interfaces is the same as the one used for extending a class.

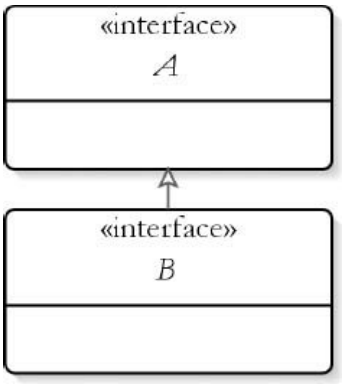

Figure 7.2: Extending an interface

What is the purpose of extending an interface? To safely add functionality to an interface without breaking existing code. This is so because you cannot add a new method to an interface once the interface has been published. Suppose an imaginary interface **XYZ** in JDK 1.7 was a popular interface with millions of implementation classes. Now, the designers of Java wanted to add a new method in **XYZ** in JDK 1.8. What would happen if a class that implemented the old **XYZ** and was compiled with a pre-JDK 1.8 compiler was deployed on JDK 1.8 (which would have shipped with the new version of **XYZ)?** It would break because the class had not provided the implementation for the new method.

The safe way would be to provide a new interface that extends **XYZ** so old software would still work and new applications can choose to implement the extension interface instead of **XYZ**.

Default Methods

Extending an interface is a safe way of adding functionality to the interface. However, you end up with two interfaces with similar functionality. This is acceptable if you only need to extend one or two interfaces. If you need to add features to hundred of interfaces, this has certainly become a serious issue.

This is exactly what the Java language designers faced when they were trying to add lambda expressions to Java 8 and add support for lambda in dozens of interfaces in the Collection Framework. Extending all the interfaces would double the number of interfaces and some would probably end up with ugly names such as **List2** or **ExtendedSet**.

Instead, they chose to add default methods. In other words, from JDK 1.8 onward, an interface can have default methods.

A default method in an interface is a method with implementation. A class implementing the interface does not have to implement the default method, which means you can add new methods to an interface without breaking backward compatibility.

To make a method in an interface a default method, add the keyword **default** in front of the method signature. Additionally, instead of terminating the signature with a semicolon, add a pair of brackets and write code in the brackets. Here is an example.

```
default java.lang.String getDescription() {
     return "This is a default method";
}
```

As you will learn later, a lot of Java interfaces in JDK 1.8 now have default methods.

When extending an interface with default methods, you have these options.

- Ignore the default methods, in effect inheriting them,
- Re-declare the default methods, which makes them abstract,
- Override the default methods.

Remember that the main reason Java now support default methods is for backward compatibility. By no means should you start writing programs without classes.

Static Methods

A static method in a class is shared by all instances of the class. In Java 8 and later you can have static methods in an interface so that all static methods associated with an interface can be written in the interface, rather than in a helper class.

The signature of a static method is similar to that of a default method. Instead of the keyword **default**, however, you use the keyword **static**. Static methods in an interface are public by default.

Static methods in an interface are rare. Of the close to 30 interfaces in the **java.util** package, only two contain static methods.

7.3 Abstract Classes

With the interface, you have to write an implementation class that perform the actual action. If there are many abstract methods in the interface, you risk wasting time overriding methods that you don't use. An abstract class has a similar role to an interface, i.e. provide a contract between a service provider and its clients, but at the same time an abstract class can provide partial implementation. Methods that must be explicitly overridden can be declared abstract. You still need to create an implementation class because you cannot instantiate an abstract class, but you don't need to override methods you don't want to use or change.

You create an abstract class by using the **abstract** modifier in the class declaration. To make an abstract method, use the **abstract** modifier in front of the method declaration. Listing 7.3 shows an abstract **DefaultPrinter** class as an example.

Listing 7.3: The DefaultPrinter class

```
package app07;
public abstract class DefaultPrinter {
    @Override
    public String toString() {
        return "Use this to print documents.";
    }
    public abstract void print(Object document);
}
```

There are two methods in **DefaultPrinter, toString** and **print**. The **toString** method has an implementation, so you do not need to override this method in an implementation class, unless you want to change its return value. The **print** method is declared abstract and does not have a body. Listing 7.4 presents a **MyPrinterClass** class that is the implementation class of **DefaultPrinter**.

Listing 7.4: An implementation of DefaultPrinter

```
package app07;
public class MyPrinter extends DefaultPrinter {
    @Override
    public void print(Object document) {
        System.out.println("Printing document");
        // some code here
    }
}
```

A concrete implementation class such as **MyPrinter** must override all abstract methods. Otherwise, it itself must be declared abstract.

Declaring a class abstract is a way to tell the class user that you want them to extend the class. You can still declare a class abstract even if it does not have an abstract method.

In UML class diagrams, an abstract class looks similar to a concrete class, except that the name is italicized. Figure 7.3 shows an abstract class.

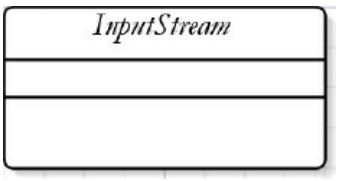

Figure 7.3: An abstract class

7.4 Polymorphism

In Java and other OOP languages, it is legal to assign to a reference variable an object whose type is different from the variable type, if certain conditions are met. In essence, if you have a reference variable **a** whose type is **A**, it is legal to assign an object of type **B**, like this

```
A a = new B();
```

provided one of the following conditions is met.

- **A** is a class and **B** is a subclass of **A**.
- **A** is an interface and **B** or one of its parents implements **A**.

As you have learned in Chapter 6, "Inheritance," this is called upcasting.

When you assign **a** an instance of **B** like in the code above, **a** is of type **A**. This means, you cannot call a method in **B** that is

not defined in **A**. However, if you print the value of **a.getClass().getName()**, you'll get "B" and not "A." So, what does this mean? At compile time, the type of **a** is **A**, so the compiler will not allow you to call a method in **B** that is not defined in **A**. On the other hand, at runtime the type of **a** is **B**, as proven by the return value of **a.getClass().getName()**.

Now, here comes the essence of polymorphism. If **B** overrides a method (say, one named **play)** in **A**, calling **a.play()** will cause the implementation of **play** in **B** (and not in **A)** to be invoked. Polymorphism enables an object (in this example, the one referenced by **a)** to determine which method implementation to choose (either the one in **A** or the one in **B)** when a method is called. Polymorphism dictates that the implementation in the runtime object be invoked. But, polymorphism does not stop here.

What if you call another method in **a** (say, a method called **stop)** and the method is not implemented in **B?** The JVM will be smart enough to know this and look into the inheritance hierarchy of **B. B**, as it happens, must be a subclass of **A** or, if **A** is an interface, a subclass of another class that implements **A**. Otherwise, the code would not have compiled. Having figured this out, the JVM will climb up the class hierarchy and find the implementation of **stop** and run it.

Now, there is more sense in the definition of polymorphism: Polymorphism is an OOP feature that enables an object to determine which method implementation to invoke upon receiving a method call.

Technically, though, how does Java achieve this? The Java compiler, as it turns out, upon encountering a method call such as **a.play()**, checks if the class/interface represented by **a** defines such a method (a **play** method) and if the correct set of parameters are passed to the method. But, that is the farthest the compiler goes. With the exception of static and final methods, it does not connect (or bind) a method call with a method body. The JVM determines how to bind a method call with the method body at runtime. In other words, except for static and final methods, method binding in Java happens at runtime and not at compile time. Runtime binding is also called late binding or dynamic binding. The opposite is early binding, in which binding occurs at compile time or link time. Early binding occurs in other languages, such as C.

Therefore, polymorphism is made possible by the late binding mechanism in Java. Because of this, polymorphism is rather inaccurately also called late binding, dynamic binding or runtime binding in other languages.

Consider the Java code in Listing 7.5.

Listing 7.5: An example of polymorphism

```
package app07;
class Employee {
    public void work() {
        System.out.println("I am an employee.");
    }
}

class Manager extends Employee {
    public void work() {
        System.out.println("I am a manager.");
    }

    public void manage() {
        System.out.println("Managing ...");
    }
}

public class PolymorphismDemo1 {
    public static void main(String[] args) {
        Employee employee;
        employee = new Manager();
        System.out.println(employee.getClass().getName());
        employee.work();
        Manager manager = (Manager) employee;
        manager.manage();
    }
}
```

Listing 7.5 defines two non-public classes: **Employee** and **Manager. Employee** has a method called **work**, and **Manager** extends **Employee** and adds a new method called **manage**.

The **main** method in the **PolymorphismDemo1** class defines an object variable called **employee** of type **Employee:**

```
Employee employee;
```

However, **employee** is assigned an instance of **Manager**, as in:

```
employee = new Manager();
```

This is legal because **Manager** is a subclass of **Employee**, so a **Manager** "is an" **Employee**. Because **employee** is assigned an instance of **Manager**, what is the outcome of **employee.getClass().getName()?** You're right. It's "Manager," not "Employee."

Then, the **work** method is called.

```
employee.work();
```

Guess what is written on the console?

```
I am a manager.
```

This means that it is the **work** method in the **Manager** class that got called, which was polymorphism in action.

Note Polymorphism does not work with static methods because they are early-bound. For example, if the **work** method in both the **Employee** and **Manager** classes were static, a call to **employee.work()** would print "I am an employee." Also, since you cannot extend final methods, polymorphism will not work with final methods either.

Now, because the runtime type of **a** is **Manager**, you can downcast **a** to **Manager**, as the code shows:

```
Manager manager = (Manager) employee;
manager.manage();
```

After seeing the code, you might ask, why would you declare **employee** as **Employee** in the first place? Why didn't you declare **employee** as type **Manager**, such as this?

```
Manager employee;
employee = new Manager();
```

You do this to ensure flexibility in cases where you don't know whether the object reference **(employee)** will be assigned an instance of **Manager** or something else.

7.5 Polymorphism in Action

Suppose you have a **Greeting** interface that defines an abstract method named **greet**. This simple interface is given in Listing 7.6.

Listing 7.6: The Greeting interface

```
package app07;
public interface Greeting {
    public void greet();
}
```

The **Greeting** interface can be implemented to print a greeting in different languages. For example, the **EnglishGreeting** class in Listing 7.7 and the **FrenchGreeting** class in Listing 7.8 implement **Greeting** to greet the user in English and French, respectively.

Listing 7.7: The EnglishGreeting class

```
package app07;
public class EnglishGreeting implements Greeting {

    @Override
    public void greet() {
        System.out.println("Good Day!");
    }
}
```

Listing 7.8: The FrenchGreeting class

```
package app07;
public class FrenchGreeting implements Greeting {
```

```
    @Override
    public void greet() {
        System.out.println("Bonjour!");
    }
}
```

The **PolymorphismDemo2** class in Listing 7.9 shows polymorphism in action. It asks the user in what language they want to be greeted. If the user chooses English, then the **EnglishGreeting** class will be instantiated. If French is selected, **FrenchGreeting** will be instantiated. This is polymorphism because the class to be instantiated is only known at runtime, after the user types in a selection.

Listing 7.9: The PolymorphismDemo2 class

```
package app07;
import java.util.Scanner;

public class PolymorphismDemo2 {

    public static void main(String[] args) {
        String instruction = "What is your chosen language?" +
                "\nType 'English' or 'French'.";
        Greeting greeting = null;
        Scanner scanner = new Scanner(System.in);
        System.out.println(instruction);
        while (true) {
            String input = scanner.next();
            if (input.equalsIgnoreCase("english")) {
                greeting = new EnglishGreeting();
                break;
            } else if (input.equalsIgnoreCase("french")) {
                greeting = new FrenchGreeting();
                break;
            } else {
                System.out.println(instruction);
            }
        }
        scanner.close();
        greeting.greet();
    }
}
```

Self Test

1 Question

Which statements are true with regard to the interface?

A. All methods in an interface are abstract

B. All methods in an interface are public

C. You can use the public and protected modifier for a method in an interface

D. You can use the abstract modifier for a method in an interface

2 Question

Which statement is true with regard to the abstract class?

A. All fields and methods in an abstract class must be abstract

B. An abstract class may contain concrete methods and fields

C. You can create an instance of an abstract class

D. An abstract class can also be final

3 Question

The **java.util.ArrayList** is a class implementing the **java.util.List** interface. Which of these statements demonstrates polymorphism?

A. ArrayList<Integer> numbers = new ArrayList<>();

B. List<String> names = new ArrayList<>();

C. List list = new List();

D. ArrayList<Object> employees = new List<>();

4 Question ?

Given

```
package com.example.test;
interface Downloadable {
    Object download();
    void printDescription();
}
class Document implements Downloadable {
    @Override
    public Object download() {
        return null;
    }
}
```

The code fragment above will not compile. How do you fix it?

A. Remove the @Override annotation in class **Document**

B. Make the **Document** class abstract

C. Provide an implementation of printDescription in **Document**

D. None of the above

5 Question ?

Given the following class:

```
package test;
abstract class Paint {
    void changeColor(int colorCode) {}
    @Override
    public String toString() {
        return "Paint";
    }
}
```

Which is the correct statement?

A. The **Paint** class must not be abstract since it has no abstract methods/fields

B. The **Paint** class will not compile unless the **abstract** modifier is removed

C. The **Paint** class compiles and is an abstract class

D. The **Paint** class will not compile because an abstract class must not override a method from a parent class

6 Question ?

Consider the following code snippet:

```
package papermatter;
class PaperCollection {
    public static void print() {
        System.out.print("PaperCollection.print()");
    }

    @Override
    public String toString() {
        return "PaperCollection";
    }
}

class Book extends PaperCollection {
    public static void print() {
        System.out.print("Book.print()");
    }

    @Override
```

```
    public String toString() {
        return "Book";
    }
}

public class Printer {
    public static void main(String[] args) {
        PaperCollection paper = new Book();
        paper.print();
        System.out.println(" | " + paper.toString());
    }
}
```

What is printed in the standard out when the code is executed?

A. Book.print() | Book

B. PaperCollection.print() | Book

C. Book.print() | PaperCollection

D. PaperCollection.print() | PaperCollection

7 Question ?

Which access modifiers can be used for an interface?

A. public

B. protected

C. default

D. private

8 Question ?

Given

```
package demo.oop;
class Display {
    public void display() {
        System.out.print("Display.display()");
    }
}
class Monitor extends Display {
    public long pixelCount() {
        return 1024 * 768;
    }
}
public class Printer {
    public static void main(String[] args) {
        Display display = new Monitor();
        // some code
        long pixelCount = display.pixelCount();
    }
}
```

Which of the following statements are true?

A. The code will compile

B. The code will not compile because the **pixelCount** method is not part of **Display**.

C. The code will compile but will thrown a **NullPointerException**

D. The code is a perfect example of polymorphism

9 Question ?

Given two interfaces and a class:

```
interface Swimmable {
    void swim();
}

interface Walkable {
    void walk();
}
```

```
class Animal {
}
```

You need to create a class named **Dog** that extends Animal and implements **Swimmable** and **Walkable**. Which statements about the **Dog** class are correct?

A. Java does not support multiple inheritance, so the **Dog** class cannot be written because a class can only implement one interface

B. The **Dog** class would have the following declaration: class Dog extends Animal implements Walkable, Swimmable { }

C. The **Dog** class would have the following declaration: class Dog extends Animal, Walkable, Swimmable { }

D. The **Dog** class would have the following declaration: class Dog implements Walkable, Swimmable extends Animal { }

E. The **Dog** class would have the following declaration: class Dog extends Animal implements Swimmable, Walkable { }

10 Question ?

The **Readable** and **Writable** interfaces shown below happen to have methods with the same signature

```
interface Readable {
    void perform(int howManyTimes);
}

interface Writable {
    void perform(int howManyTimes);
}
```

You need to write a class that implements both interfaces. Which statements are true?

A. You cannot implement interfaces with the same name

B. You can implement both interfaces and you need to provide two implementations of perform()

C. You can implement both interfaces and you need to provide one implementation of perform()

D. It is not necessary to implement both Readable and Writable since they have exactly the same set of methods. You can just implement either one and have the same effect as implementing both interfaces

Answers

1 **B, D**.

A is incorrect because Java 8 interfaces can have default and static methods as well as abstract methods. It would have been correct if we were talking about Java 7 and previous versions.

B is correct because all methods in an interface are public even if defined without the public modifier.

C is incorrect because a method in an interface cannot be protected.

D is correct because you can use the keyword **abstract** for abstract methods in an interface. However, the use of the keyword is redundant as all methods that declare no method body are implicitly abstract.

2 **B**.

An abstract class may contain concrete fields and methods and it cannot be instantiated. An abstract class must be extended and therefore an abstract class cannot also be a final class.

3 **B**.

Polymorphism allows an instance of class B to be assigned to a reference variable of type A if

- **A** is a class and **B** is a subclass of **A**
- **A** is an interface and **B** or one of its parents implements **A**

A will not cause a compile error but it is not polymorphism. B is polymorphism as **List** is an interface and **ArrayList** is a class implementing **List**. C is invalid because you cannot instantiate an interface. D is also incorrect and will generate a compile error because you cannot instantiate an interface and assigning an instance of a **List** to a reference variable of type **ArrayList** is not permitted.

4 **B, C**.

A class that implements an interface must implement all its methods. The code will not compile because the **Document** class is missing an implementation of **printDescription**. You can fix it by either making **Document** abstract or provide an implementation of **printDesription** in

Document.

5 **C**.

An abstract class does not have to have an abstract member. Since **Paint** is abstract, it may not be instantiated.

6 **B**.

Polymorphism does not work on static methods or static fields because they are early-bound. The **paper** variable references an instance of **Book**, however calling a static method on **paper** will invoke the method on **PaperCollection**, not **Book**.

7 **A, C**.

An interface, like a class, can have the public or default access modifier.

8 **B**.

Since **display** is of type **Display**, you cannot call the **pixelCount** method on it. You can upcast **display** to **Monitor** like so to fix the compiler error:

```
long pixelCount = ((Monitor) display).pixelCount();
```

9 **B, E**.

Java does not support multiple inheritance so you can only extend one class. However, you can implement multiple interfaces. The syntax of a class that extends another class and implements interfaces is as follows.

```
[public] class className extends parentClass implements interface1,
        interface2, ...
```

The implemented interfaces can be in any order, so B and E are both correct.

10 **C**.

You need to implement both interfaces so you can instantiate your class and assign it to a variable of type **Readable** or **Writable**. Since the perform methods have the same signature, you just need one implementation. In fact, it would generate a compile error if you tried to provide two implementations since both methods would have the same signature.