3 Object-oriented design principles

Exam objectives covered in this chapter	What you need to know
[3.1] Write code that declares, implements, and/or extends interfaces	The need for interfaces. How to declare, implement, and extend interfaces. Implications of implicit modifiers that are added to an interface and its members.
[3.2] Choose between interface inheritance and class inheritance	The differences and similarities between implementing inheritance by using interfaces and by using abstract or concrete classes. Factors that favor using interface inheritance over class inheritance, and vice versa.
[3.3] Apply cohesion, low-coupling, IS-A, and HAS-A principles	Given a set of IS-A and HAS-A relationships, how to implement them in code. Given code snippets, how to correctly identify the relationships (IS-A or HAS-A) implemented by them. How to identify and promote low coupling and high cohesion.
[3.4] Apply object composition principles (including HAS-A relationships)	Given that an object can be composed of multiple other objects, how to determine the types of compositions—and implement them in code.
[3.5] Design a class using the Singleton design pattern	How to implement the Singleton design pattern. The need for the existence of exactly one copy of a class.
[3.6] Write code to implement the DAO pattern	The usability of the DAO pattern. How this pattern enables separation of data access code in an application.
[3.7] Design and create objects using a Factory pattern	The need for, use of, and benefits of a Factory for creating objects. How this pattern is used in the existing Java API classes.

Have you ever tried to find out the secret(s) behind the most successful people? Almost all agree to follow a set of lifelong principles. So, articles like "Three Common Habits of the Most Successful People" might include points similar to these:

- Never hit the snooze button when the alarm goes off in the morning, so you aren't delaying your actions.
- First things first: prioritize your work.
- Follow your passion and do what you love, because you'll be working almost all your life.

These are the principles that successful people follow (though perhaps in a different manner) to achieve the greatest height of success.

Similarly, *object-oriented design (OOD) principles* enable you to create better application designs, which are manageable and extensible. For example, as a programmer or designer, you know that application requirements typically change. Implementing these modified needs requires changes in the existing code, which usually introduces bugs. Chances are that if the application design implements OOD principles, the modification task will require comparatively less effort. Again, as an example, if your application's design uses the design principles of low coupling and high cohesion, chances are low that changes in a class will affect another class.

Design patterns (for example, the Singleton pattern) also help you design better applications. Unlike inheritance, a design pattern is an example of experience reuse and not code reuse. Building sloping roofs in areas that receive a lot of snowfall can be compared to using a design pattern. A sloping roof was identified as a solution to avoid accumulation of snow on rooftops after multiple people faced issues with flat roofs. Just as a sloping roof is applicable specifically to areas receiving snowfall, a design pattern resolves a *specific* design issue.

Obviously, we can see that the object-oriented design principles are important, but what specifically do you need to know for the exam? Well, the exam will test you on what object-oriented design principles are and how to apply them in your applications. You'll likely find questions on the creation of and preferred use of classes and interfaces to design your application, and how to relate and use Java objects together. In addition, you'll need to know how to make the best use of design patterns in your application. Yes, this is a lot, but I promise to walk you through each piece so you're prepared. This chapter covers

- Interfaces—their declaration, implementation, and application
- Choosing between class inheritance and interface inheritance
- Relationships between Java objects
- Application of object composition principles
- Implementation of IS-A and HAS-A relationships between objects
- Singleton design pattern
- Data Access Object (DAO) design pattern
- Factory design pattern

Interfaces are one of the most powerful concepts in Java. Believe me, not many designers completely understand how to use them *effectively* in their design. To be a good application designer, you *must* know how to do so. Let's start with a quick example: in how many ways can you refer to your father? Apart from being your father, he could also be referred to as a friend, guide, husband, swimmer, orator, teacher, manager, and much more. How can you achieve the same in Java, to refer to the same object by using multiple types? In the next section, you'll learn about the need for using interfaces, followed by declaring, implementing, and extending them. Let's get started.

3.1 Interfaces

[3.1] Write code that declares, implements, and/or extends interfaces

Before we deep-dive into working with interfaces, note that the term *interface* has multiple meanings. First, an interface is a type created by using the keyword `interface`. For example, the following code creates the interface `Movable`:

```
interface Moveable {
    void move();
}
```

An interface in its second meaning is more general. It's how two systems can interact with each other (like your television and the remote control). It's how classes can interact with each other, using their public methods. For class `Person`, its interface (public methods) refers to its methods `eat()` and `work()`:

```
class Person {
    public void eat() {}
    public void work() {}
}
```

Figure 3.1 represents an interface as a type and as a group of public methods of a class.

Note that this exam objective refers to an *interface* as a type, which is created using the keyword `interface`.

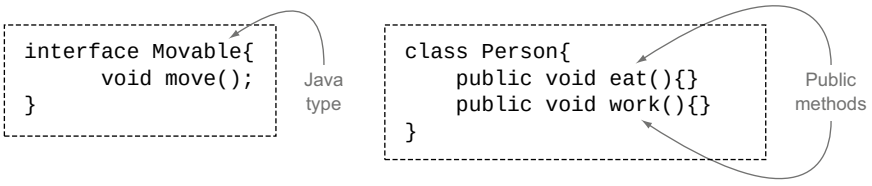


Figure 3.1 The term *interface* has two meanings: a type created using the keyword `interface` and a group of public methods of a class.

3.1.1 Understanding interfaces

We all, quite often, use interfaces in our lives. For example, when you refer to someone as a *runner*, do you care whether that person is also an orator, a parent, or an entrepreneur? You care only that the person is able to *run*. The term *runner* enables you to refer to unrelated individuals, by opening a small window to each person and accessing behavior that's applicable to *only* that person's capacity as a runner. Someone can be referred to as a runner only if that person supports characteristics relevant to running, though the specific behavior can depend on the person.

In the preceding example, you can compare the term *runner* to a Java interface, which defines the required behavior *run*. An interface can define a set of behaviors (methods) and constants. It delegates the implementation of the behavior to the classes that implement it. Interfaces are used to refer to multiple related or unrelated objects that share the same set of behavior. Figure 3.2 compares the interface *runner* with a small *window* to an object, which is concerned only about the running capabilities of that object.

Similarly, when you design your application by using interfaces, you can use similar windows (also referred to as *specifications* or *contracts*) to specify the behavior that you need from an object, without caring about the specific type of objects. Separating the required behavior from its implementation has many benefits. As an application designer, you can use interfaces to *establish* the behavior that's required from objects, promoting flexibility in the design (new classes that implement an interface can be created and used later). Interfaces make an application manageable, extensible, and less prone to propagation of errors due to changes to existing types.

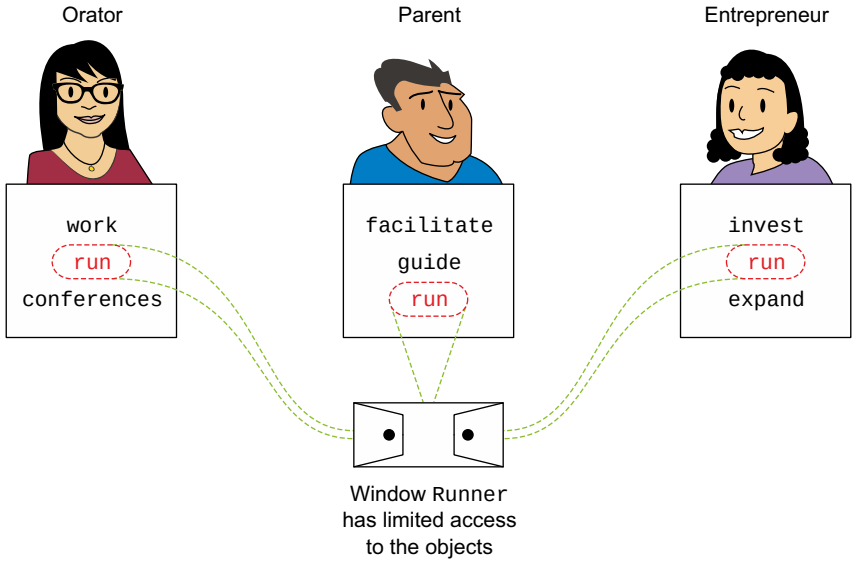


Figure 3.2 You can compare an interface with a window that can connect multiple objects but has limited access to them.

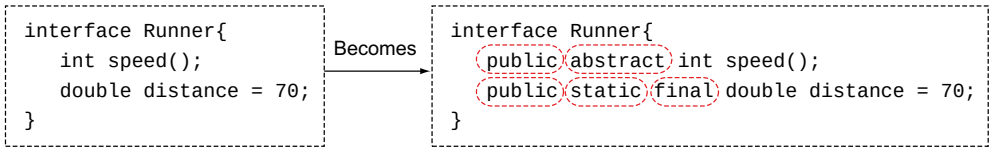


Figure 3.3 All the methods of an interface are implicitly public and abstract. Its variables are implicitly public, static, and final.

3.1.2 Declaring interfaces

You can define methods and constants in an interface. Declaring an interface is simple, but don't let this simplicity take you for a ride. For the exam, it's important to understand the implicit modifiers that are added to the members of an interface. All methods of an interface are implicitly public and abstract, and its variables are implicitly public, static, and final. Let's start with the interface `Runner` that defines a method `speed()` and a variable `distance`. Figure 3.3 shows how implicit modifiers are added to the members of interface `Runner` during the compilation process.

Why do you think these implicit modifiers are added to the interface members? Because an interface is used to define a contract, it doesn't make sense to limit access to its members—and so they are implicitly public. An interface can't be instantiated, and so the value of its variables should be defined and accessible in a static context, which makes them implicitly static. Because an interface is a contract, its implementations shouldn't be able to change it, so the interface variables are implicitly final. Interface methods are implicitly abstract so that it's mandatory for the classes to implement them.

The exam will also test you on the various components of an interface declaration, including access and nonaccess modifiers. Here's the complete list of the components of an interface declaration:

- Access modifiers
- Nonaccess modifiers
- Interface name
- All extended interfaces, if the interface is extending any interfaces
- Interface body (variables and methods), included within a pair of curly braces `{}`

To include all the possible components, let's modify the declaration of the interface `Runner`:

```
public strictfp interface Runner extends Athlete, Walker {}
```

The components of the interface `Runner` are shown in figure 3.4. To declare any interface, you *must* include the keyword `interface`; the name of the interface; and its body, marked by `{}`.

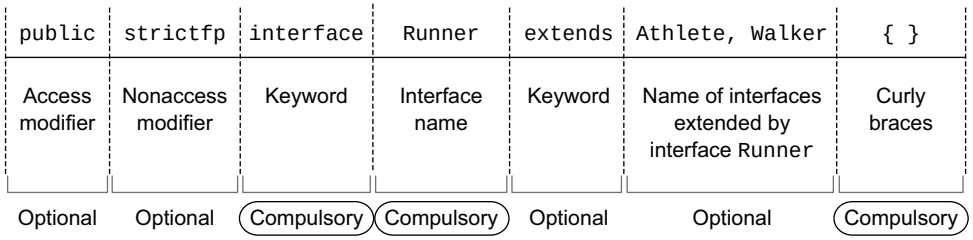


Figure 3.4 Components of an interface declaration

The optional and compulsory components of an interface can be summarized as listed in table 3.1.

Table 3.1 Optional and compulsory components of an interface declaration

Compulsory	Optional
Keyword `interface`	Access modifier
Name of the interface	Nonaccess modifier
Interface body, marked by the opening and closing curly braces `{}`	Keyword `extends`, together with the name of the base interface(s). (Unlike a class, an interface can extend multiple interfaces.)

EXAM TIP The declaration of an interface can't include a class name. An interface can never extend any class.

Can you define a top-level, *protected* interface? No, you can't. For the exam, you must know the answer to such questions about the correct values for each component that can be used with an interface declaration. Let's dive into these nuances.

VALID ACCESS MODIFIERS FOR AN INTERFACE

You can declare a *top-level interface* (the one that isn't declared within any other class or interface), with only the following access levels:

- `public`
- No modifier (default access)

If you try to declare your top-level interfaces by using the other access modifiers (`protected` or `private`), your interface will fail to compile. The following definitions of the interface `MyInterface` won't compile:

```
private interface MyInterface{}
protected interface MyInterface {}
```

Top-level interface can't be defined as private

Top-level interface can't be defined as protected

EXAM TIP All the top-level Java types (classes, enums, and interfaces) can be declared using only two access levels: `public` and default. Inner or nested types can be declared using any access level.

VALID ACCESS MODIFIERS FOR MEMBERS OF AN INTERFACE

All members of an interface—variables, methods, inner interfaces, and inner classes (yes, an interface can define a class within it!)—are public by default. Interfaces support only the `public` access modifier. Using other access modifiers results in compilation errors.

```
interface MyInterface {
    private int number = 10;
    protected void aMethod();
    interface interface2{}
    public interface interface4{}
}
```

❶ **Won't compile**

Won't compile

interface2 is implicitly prefixed with public.

Interface member can be prefixed with public

The code at ❶ fails compilation with the following error message:

```
illegal combination of modifiers: public and private
    private int number = 10;
```

SPECIAL CHARACTERISTICS OF METHODS AND VARIABLES OF AN INTERFACE

Methods in interfaces are public and abstract by default. The following methods defined individually in an interface are equivalent:

```
int getMembers();
public abstract int getMembers();
```

Variables defined in interfaces are public, static, and final by default. The following variables defined individually in an interface are equivalent:

```
int maxMembers = 100;
public static final int maxMembers = 100;
```

Because the interface variables are implicitly final, you can define only *constants* in an interface. Ensure that you initialize these constants, or your code won't compile:

```
interface MyInterface {
    int number;
}
```

Won't compile; variables within interface must be initialized.

VALID NONACCESS MODIFIERS FOR AN INTERFACE

You can declare a top-level interface with only the following nonaccess modifiers:

- `abstract`
- `strictfp`

NOTE The `strictfp` keyword guarantees that results of all floating-point calculations are identical on all platforms.

If you try to declare your top-level interfaces by using the other nonaccess modifiers (`final`, `static`, `transient`, `synchronized`, or `volatile`), the interface will fail to compile. All of the following interface declarations fail to compile:

```
final interface MyInterface {}
static interface MyInterface {}
transient interface MyInterface {}
synchronized interface MyInterface {}
volatile interface MyInterface {}
```

Won't compile; invalid nonaccess modifiers used with interface declaration

A nested interface can be defined using the nonaccess modifier `static` (any other nonaccess modifier isn't allowed):

```
class Outer {
    static interface MyInterface1 {}
}
```

Nested interface

With good coverage of interface declaration, let's start making classes implement interfaces.

3.1.3 Implementing interfaces

You can compare implementing an interface to signing a contract. When a concrete class declares its implementation of an interface, it agrees to implement all its abstract methods. A class can implement multiple interfaces. For example, class `Home` implements `Livable` and `GuestHouse`:

```
interface Livable {
    void live();
}
interface GuestHouse {
    void welcome();
}
class Home implements Livable, GuestHouse {
    public void live() {}
    public void welcome() {}
}
```

abstract method live()

abstract method welcome()

Class uses keyword implements to implement interface

If you don't implement all the methods defined in the implemented interfaces, a class can't compile as a concrete class. Let's modify the code of class `Home`, as follows:

```
class Home implements Livable, GuestHouse {
    public void welcome() {}
}
```

The compiler says it all:

```
House.java:7: error: Home is not abstract and does not override
abstract method live() in Livable
class Home implements Livable, GuestHouse {
^
1 error
```

So a class can choose not to implement all the methods from the implemented interface(s) and still compile successfully, but only if it's defined as an abstract class, as follows:

```
abstract class Home implements Livable, GuestHouse {
    public void welcome() {}
}
```

Abstract class doesn't have to implement all methods from implemented interfaces

EXAM TIP A *concrete* class must implement all the methods from the interfaces that it implements. An *abstract* class can choose not to implement all the methods from the interfaces that it implements.

DEFINING AND ACCESSING VARIABLES WITH THE SAME NAME

A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. In the following class, the interface `Livable` defines variables `status` and `ratings`. Class `Home` implements `Livable` and defines instance variable `status` and static variable `ratings`, with a default access level:

```
interface Livable {
    boolean status = true;
    int ratings = 10;
}
class Home implements Livable {
    boolean status;
    static int ratings = 7;
    Home() {
        System.out.println(status);
        System.out.println(Livable.status);

        System.out.println(ratings);
        System.out.println(Livable.ratings);
    }
}
```

public variables

Variables with default access

Prints "false"

Prints "true"

Prints "7"

Prints "11"

EXAM TIP A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.

FOLLOWING METHOD OVERRIDING RULES FOR IMPLEMENTING INTERFACE METHODS

The methods in an interface are public, by default. So, trying to assign weaker access to the implemented method in a class won't allow it to compile:

```
interface Livable {
    void live();
}
class Home implements Livable {
    void live() {}
}
```

public method

Won't compile; method implemented using weaker access

The compilation error message says it all:

```
House.java:8: error: live() in Home cannot implement live() in Livable
    void live() {}
         ^
  attempting to assign weaker access privileges; was public
1 error
```

EXAM TIP Because interface methods are implicitly public, the implementing class must implement them as public methods, or else the class will fail to compile.

IMPLEMENTING MULTIPLE INTERFACES THAT DEFINE METHODS WITH THE SAME NAME

Methods in the interfaces don't define any implementation; they come without any baggage. But what happens if a class implements multiple interfaces that define methods with the same name? Let's add a method `live()` to interface `GuestHouse` (modifications in bold):

```
interface Livable {
    void live();
}
interface GuestHouse {
    void welcome();
    void live();
}
```

Interface Livable defines method live().

Interface GuestHouse also defines method live().

Class `Home` implements two interfaces, `Livable` and `GuestHouse`, both of which define method `live()`:

```
class Home implements Livable, GuestHouse {
    public void live() {
        System.out.println("live");
    }
    public void welcome() {
        System.out.println("welcome");
    }
}
```

Method live() in Home has only one implementation.

Both the Java compiler and Java Runtime Environment are good with the preceding code. Because the signature of method `live()` is the same in both interfaces, `Livable` and `GuestHouse`, class `Home` needs to define only one implementation for method `live()` to fulfill both contracts (interface implementations).

OVERLAPPING METHOD IMPLEMENTATIONS WITH THEIR OVERLOAD VERSIONS

A class can try to implement multiple interfaces that define methods with the same name. But in doing so, you can have a not-so-pleasant cocktail of overlapping method implementations and their overloaded versions. We have two scenarios here:

- Correctly overloaded methods
- Invalid overloaded methods

Overloaded methods are defined by using the same name but a different parameter list. For example, when implemented in class `Home`, method `live()` defined in the interface `Livable` overloads method `live()` defined in the interface `GuestHouse`. Class `Home` must implement both these methods:

```
interface Livable {
    void live();
}
interface GuestHouse {
    void live(int days);
}
class Home implements Livable, GuestHouse {
    public void live() {
        System.out.println("live");
    }
    public void live(int days) {
        System.out.println("live for " + days);
    }
}
```

live() doesn't accept any method parameters.

live() accepts a method parameter.

Correctly overloaded method live() from Livable

Correctly overloaded method live() from GuestHouse

You can't define overloaded methods by changing only the return type of methods. What happens if method `live()` in the interfaces `Livable` and `GuestHouse` returns different types? In this case, class `Home` needs to implement both versions of method `live()`, which can't be qualified as overloaded methods. So class `Home` doesn't compile in this case:

```
interface Livable {
    String live();
}
interface GuestHouse {
    void live();
}
class Home implements Livable, GuestHouse {
    public String live() {
        return null;
    }
    public void live() {
        System.out.println("live" );
    }
}
```

live() returns String.

live() returns nothing—void.

Class Home won't compile.

When implemented in class Home, both versions of live() qualify as incorrectly overloaded methods.

Here's the compiler error for class `Home`:

```
Home.java:11: error: method live() is already defined in class Home
    public void live() {
                ^
Home.java:7: error: Home is not abstract and does not override abstract
 method live() in GuestHouse
class Home implements Livable, GuestHouse {
^
```

```
Home.java:8: error: live() in Home cannot implement live() in GuestHouse
    public String live() {
                  ^
  return type String is not compatible with void
3 errors
```

EXAM TIP A class can implement methods with the same name from multiple interfaces. But these must qualify as correctly overloaded methods.

3.1.4 Extending interfaces

An interface can inherit multiple interfaces. Because all the members of an interface are implicitly `public`, a derived interface inherits all the methods of its super interface(s). An interface uses the keyword `extends` to inherit an interface, as shown in the following example:

```
interface GuestHouse {
    void welcome();
}
interface PayingGuestHouse extends GuestHouse {
    void paidBreakfast();
}
interface StudentPGHouse extends PayingGuestHouse {
    void laundry();
}
interface ChildFriendly {
    void toys();
}
interface FamilyPGHouse extends ChildFriendly, PayingGuestHouse {
    void kitchen();
}
```

The preceding code is shown in figure 3.5 as a UML diagram.

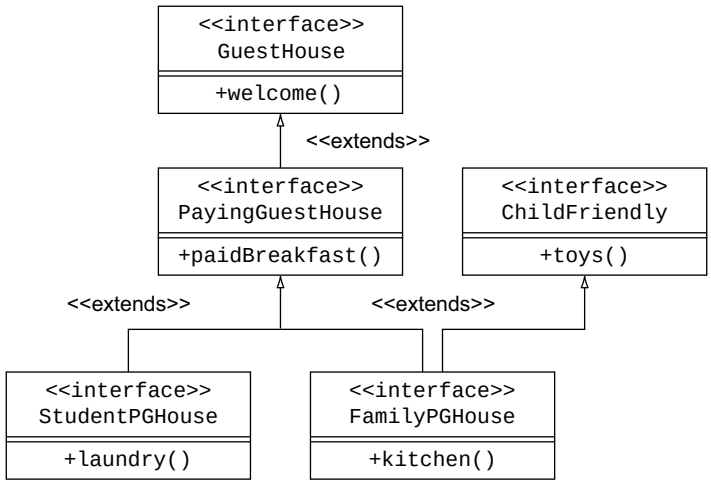


Figure 3.5 UML relationships between interfaces that extend other interfaces

By extending interfaces, you can combine methods of multiple interfaces. In the previous example, the interface `FamilyPGHouse` combines the methods of the interfaces `ChildFriendly`, `PayingGuestHouse`, and `GuestHouse`.

When a class implements an interface, it should implement all the methods defined in the interface and its base interfaces, unless it's declared as abstract. If, for example, a class implements the interface `PayingGuestHouse`, that class must implement method `paidBreakfast()` defined in the interface `PayingGuestHouse`, and method `welcome()` defined in the interface `GuestHouse`. Let's work with a concrete example of a class implementing interfaces in the next section.

Rules to remember about interfaces

- An interface is abstract by definition.
- An interface can define only public, abstract methods and public, static, final variables.
- An interface uses the keyword `extends` to inherit other interfaces.
- A class can *implement* multiple interfaces. An interface can *extend* multiple interfaces.
- An interface can define inner interfaces and (surprisingly) inner classes too.
- If a class doesn't implement all the methods of the interface that it implements, the class *must* be defined as an abstract class.
- A class uses the keyword `implements` to implement an interface.
- If a class implements multiple interfaces that define methods with the same name, the interface methods must either qualify as correctly overloaded or overridden methods, or else the class won't compile.

3.2 Class inheritance versus interface inheritance

[3.2] Choose between interface inheritance and class inheritance

A class can implement interface(s) and a class can also extend a class and override its methods. So the big question is, while designing classes and interfaces in your application, how do you implement inheritance to reuse existing code? Would you prefer that your class inherit another (abstract or concrete) class or implement an interface? There is no straight answer to this question. Depending on the requirements, you might need to extend a class or implement an interface, because each offers a different set of benefits. To make this informed decision, let's focus on the similarities and differences in both approaches.

3.2.1 Comparing class inheritance and interface inheritance

An interface doesn't include implementation details, whereas a class does. This basic distinction has introduced differences in inheriting a class and implementing an interface. These differences are listed in table 3.2.

Table 3.2 Differences between class inheritance and interface inheritance

	Class inheritance	Interface inheritance
Instantiation of derived class	Instantiation of a derived class instantiates its base class.	Interfaces can't be instantiated.
How many?	A class can extend only one base class.	A class can implement multiple interfaces.
Reusing implementation details	A class can reuse the implementation details of its base class.	An interface doesn't include implementation details.
Modification to base class implementation details	With the modified base class, a derived class might cease to offer the functionality it was originally created for; it may also fail to compile.	Interfaces don't include implementation details.

Now for the similarities between class and interface inheritance. In both cases, you can refer to a derived class or implementing class by using a variable of the base class or implemented interface.

3.2.2 Preferring class inheritance over interface inheritance

Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class. Let's examine both of these in detail.

Reusing the implementation from the base class

When we create any class, we extend and reuse class `java.lang.Object`. The class `Object` defines code to take care of all the threading and object-locking issues, together with providing default implementation for methods like `toString()`, `hashCode()`, and `equals()`. Method `toString()` returns a textual description (`String`) of an instance. Methods like `hashCode()` and `equals()` enable objects to be stored and retrieved efficiently in hash-based collection classes like `HashMap`. What do you think would happen if class `java.lang.Object` was defined as an interface? In this case, you'd need to implement all these methods for every class that you created.

But it's not useful to replicate this type of boilerplate code across many implementation classes. So class inheritance comes in handy here.

Adding new behavior in all derived classes

Imagine you created a set of entities (`Lion`, `Elephant`), identified their common behavior, and moved the common behavior to their common base class (`Animal`). Because you control the definition of all these classes, you might add new behavior to your base class and make it available to all the derived classes. Examine the following definition of the abstract class `Animal` and nonabstract class `Lion`, which extends class `Animal`:

```
public abstract class Animal {
    public abstract void move();
    public abstract void live();
}
```

```
public class Lion extends Animal {
    public void move(){/*...*/}
    public void live(){/*...*/}
}
```

Let's add another method to `Animal` (modifications in bold):

```
public abstract class Animal {
    public abstract void move();
    public abstract void live();
    public void eat() {/*...*/}
}
public class Lion extends Animal {
    public void move(){/*...*/}
    public void live(){/*...*/}
}
```

Addition of new method eat

The addition of public method `eat()` in class `Animal` makes it available to all subclasses of `Lion`, implicitly. But adding or modifying behavior in a base class is not always a bed of roses, as you'll see in the next section.

3.2.3 Preferring interface inheritance over class inheritance

You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.

IMPLEMENTING MULTIPLE INTERFACES

Imagine you need to use a class that can be executed in a separate `thread` *and* can be attached as an `ActionListener` to a GUI component. You can achieve this by making your class implement multiple interfaces that support these functionalities—interfaces `Runnable` and `ActionListener`:

```
class MyClass implements Runnable, ActionListener {
    //..code to implement methods from interface
    //..Runnable and ActionListener
}
```

Interface implementation has one major advantage: a class can implement multiple interfaces, to support multiple functionality. For the preceding example, you can pass instances of class `MyClass` to all methods that define parameters of type `Runnable` or `ActionListener`.

DEFINING A NEW CONTRACT FOR EXISTING CLASSES TO ABIDE BY

Starting with Java version 7, a new language feature has been added to the exception handling: auto-closing resources by using a `try`-with-resources statement. The intent is to define a `try` statement that can use streams that can be *auto-closed*, releasing any system resources associated with them. This prevents Java objects from using resources that are no longer required. These unused and unclosed resources can lead to resource leakage. Though a `try` statement provides a `finally` clause that can be used by programmers to close streams, at times it isn't being used to do so. So to manage resources automatically, Java designers introduced the `try`-with-resources

statement. The objects that can be used with this statement need to define a `close` method, so this method can be called to automatically close and release used resources. To apply this constraint, Java designers at Oracle started by defining interface `java.lang.AutoCloseable`, as follows:

```
package java.lang;
public interface AutoCloseable {
    void close() throws Exception;
}
```

The `try`-with-resources statement can declare only objects that implement the interface `java.lang.AutoCloseable`. Prior to Java 7 (starting with Java 5), many input and output streams from the Java IO API implemented the interface `java.io.Closeable`:

```
public abstract class Reader implements Readable, Closeable {
public abstract class Writer implements Appendable, Closeable, Flushable {
public abstract class InputStream implements Closeable {
public abstract class OutputStream implements Closeable, Flushable {
```

To accommodate the use of class instances mentioned in the preceding code, the existing interface `java.io.Closeable` was tricked (read: modified) into extending `java.lang.AutoCloseable`:

```
package java.io;
import java.io.IOException;
   public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

Also, any user-defined class that implements the interface `java.lang.AutoCloseable` or any of its subinterfaces can be used with a `try`-with-resources statement. Here's a quick example of using a `try`-with-resources statement:

```
void openFile(String filename) throws Exception {
    try (FileInputStream fis = new FileInputStream(new File(filename))) {
            /* ... */
    }
}
```

Object of FileInputStream can be declared in try-with-resources because FileInputStream implements Closeable (which extends AutoCloseable).

As you'll see in chapter 6, a `try` block in `try`-with-resources can exist without any companion `catch` or `finally` block.

Interface inheritance added new behavior to classes like `Reader` and `Writer` without breaking their existing code. Inheritance of the interface `AutoCloseable` by `Closeable` defines multiple contracts for instances of these classes. They can now be assigned to a reference variable of type `AutoCloseable`, enabling them to be used with a `try`-with-resources statement.

Several other classes and interfaces implement or extend `AutoCloseable`, among the main Java Database Connectivity (JDBC) interfaces (`Connection`, `Statement`, `ResultSet`) and several Java Sound API interfaces.

FRAGILE DERIVED CLASSES

Adding to or modifying a base class can affect its derived classes. Adding new methods to a base class can result in breaking the code of a derived class. Consider this initial arrangement, which works well:

```
public abstract class Animal {
    void move(){}
}
class Lion extends Animal {
    void live(){}
}
```

Now consider a modified arrangement: a new method `live()` is added to base class `Animal`. Because `live()` clashes (because of an incorrectly overridden method) with the existing method `live()` in its derived class `Lion`, `Lion` will no longer compile:

```
public abstract class Animal {
    void move(){}
    String live(){                          // New method added to Animal
        return "live";
    }
}
class Lion extends Animal {                 // live() in Lion neither overloads nor overrides live() in Animal.
    void live(){}
}
```

EXAM TIP Class inheritance isn't always a good choice because derived classes are fragile. If any changes are made to a base class, a derived class might break. Extending classes that are from another package or are poorly documented aren't good candidates for base classes.

If a base class chooses to modify the implementation details of its methods, the derived classes might not be able to offer the functionality they were supposed to, or they might respond differently. Consider this initial arrangement:

```
public abstract class Animal {
    String currentPosition;
    public void move(String newPosition){
        currentPosition = newPosition;
    }
}
class Lion extends Animal {
    void changePosition(String newPosition) {
        super.move(newPosition);
        System.out.println("New Position:" + newPosition);
    }
}
class Test{
    public static void main(String args[]) {
        new Lion().changePosition("Forest");    // Prints "New Position:Forest" once
    }
}
```

Imagine that `Animal` adds another line of code to method `move()`. Let's see how it changes the code output of class `Test` (modification in bold):

```
public abstract class Animal {
    String currentPosition;
    public void move(String newPosition){
        currentPosition = newPosition;
        System.out.println("New Position:" + newPosition);
    }
}
class Lion extends Animal {
    void changePosition(String newPosition) {
        super.move(newPosition);
        System.out.println("New Position:" + newPosition);
    }

    public static void main(String args[]) {
        new Lion().changePosition("Forest");
    }
}
```

Implementation details modified; new code line added

Prints "New Position:Forest" twice

EXAM TIP There isn't any clear winner when it comes to selecting the better option from class inheritance and interface inheritance. Analyze the given conditions or situations carefully to answer questions on this topic.

Imagine the thought process required to modify the core Java classes when its new version is planned or executed. As you witnessed in the preceding example, changes to a base class can break the code of its derived classes.

In the next "Twist in the Tale" exercise, let's try to figure out how an already-defined class implements the interface `AutoCloseable`, or any of its subinterfaces, so it can be used with a `try`-with-resources statement.

Twist in the Tale 3.1

As shown in the preceding examples, a `try`-with-resources statement can declare resources (objects) that implement the interface `java.lang.AutoCloseable` or any of its subinterfaces. A programmer has defined a class `MyLaptop` as follows, and wants to use it with a `try`-with-resources statement. Which option will enable the programmer to achieve this goal?

```
class MyLaptop {
    public int open()    {
        /* some code */
        return 0;
    }
    public void charge() {
        /* some code */
    }
```

```
    public int close()   {
        /* some code */
        return 1;
    }
}
```

a Make class `MyLaptop` implement interface `java.lang.AutoCloseable`.
b Make class `MyLaptop` implement interface `java.io.Closeable`, which extends interface `java.lang.AutoCloseable`.
c Create a new interface `MyCloseable` that extends `java.lang.AutoCloseable`, and make class `MyLaptop` implement it.
d Class `MyLaptop` can't implement interface `java.lang.AutoCloseable` or any of its subinterfaces because of the definition of its method `close()`.

In the next section, you'll work with how to identify and implement IS-A and HAS-A principles in code.

3.3 IS-A and HAS-A relationships in code

[3.3] Apply cohesion, low-coupling, IS-A, and HAS-A principles

You'll be amazed at how easily you can identify and implement IS-A and HAS-A relationships between classes and objects, if you remember one simple rule—follow the literal meaning of these terms:

- *IS-A*—This relationship is implemented when
  - A class extends another class (derived class IS-A base class)
  - An interface extends another interface (derived interface IS-A base interface)
  - A class implements an interface (class IS-A implemented interface)
- *HAS-A*—This relationship is implemented by defining an instance variable. If a class—say, `MyClass`—defines an instance variable of another class—say, `YourClass`—`MyClass` HAS-A `YourClass`. If `MyClass` defines an instance variable of an interface—say, `YourInterface`—`YourClass` HAS-A `YourInterface`.

EXAM TIP Representing IS-A and HAS-A relationships by using (quick) UML diagrams can help you on the exam. Though you may not see UML diagrams on the exam, creating quick UML diagrams on an erasable board (or something similar) provided to you during the exam will help you answer these questions.

The exam will test whether you can identify and implement these relationships in your code, so let's start with an example of an IS-A relationship.

3.3.1 Identifying and implementing an IS-A relationship

An *IS-A relationship* is implemented by extending classes or interfaces and implementing interfaces. Traverse the inheritance tree *up* the hierarchy to identify this relationship. A derived class IS-A *type* of its base class and its implemented interfaces. A derived interface IS-A type of its base interface. The reverse isn't true; a base class or interface *isn't a* type of its derived class or interface.

IDENTIFYING AN IS-A RELATIONSHIP

Here's a simple but long example for you to read and comprehend:

```
interface Movable {}
interface Hunter extends Movable {}

class Animal implements Movable {}
class Herbivore extends Animal {}
class Carnivore extends Animal implements Hunter {}

class Cow extends Herbivore {}
class Goat extends Herbivore {}

class Lion extends Carnivore {}
class Tiger extends Carnivore {}
```

Which of the following options do you think are correct?

- `Cow` IS-A `Hunter`.
- `Tiger` IS-A `Herbivore`.
- `Cow` IS-A `Movable`.
- `Animal` IS-A `Herbivore`.

To answer this question, refer to the preceding code, and you'll notice that the interface `Hunter` is implemented only by class `Carnivore`. Class `Cow` doesn't extend class `Carnivore`. So, `Cow` IS-A `Hunter` is incorrect.

Similarly, you can refer to the preceding code to answer all the other options. Option 2 is incorrect because class `Tiger` doesn't extend class `Herbivore`. Option 3 is correct because the interface `Movable` is implemented by class `Animal`, which is the base class of `Herbivore`, extended by class `Cow`.

Option 4 is incorrect because you can't traverse the hierarchy tree *down* to determine an IS-A relationship. Evaluate it like this: An `Herbivore` IS-A type of `Animal` with some additions or modifications because an `Herbivore` can modify (override) methods of class `Animal` and add new ones. But `Animal` IS-NOT-A `Herbivore`. `Animal` *can also be* a `Carnivore`.

Phew! So we had to refer to the code multiple times to answer each option. How about representing the relationships between these classes and interfaces by using UML notation, as shown in figure 3.6?

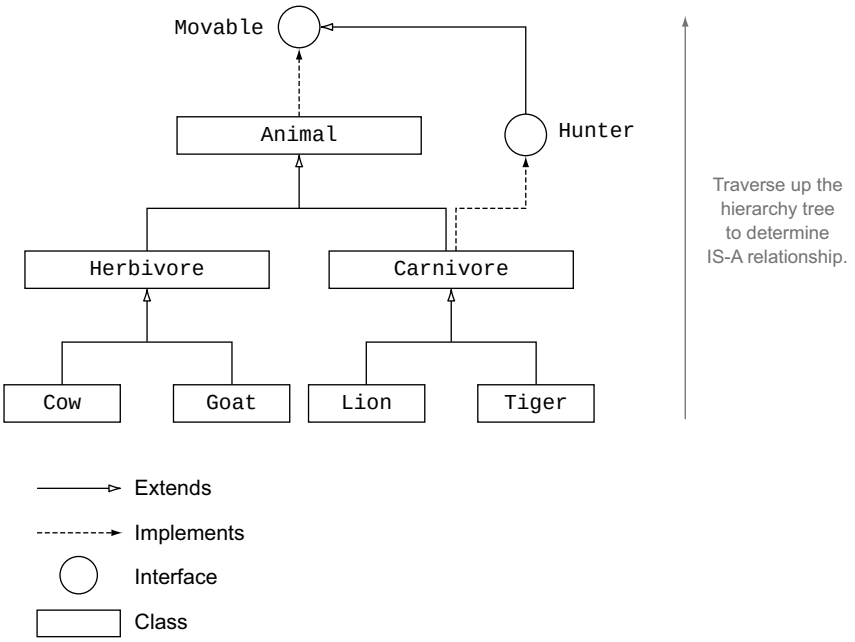


Figure 3.6 A UML representation can help answer questions about IS-A relationships between classes and interfaces.

If you can traverse up, from a derived class or interface to a base class or interface, following the connecting arrows (lined or dashed), the derived entity shares an IS-A relationship with the base entity. If you think that the preceding figure seems to depict a rather polished form of a class-and-interface relationship, look at figure 3.7, which shows the same relationship in a rather raw form.

I understand that you may not have the time or patience to draw neat diagrams during the exam because of time constraints or space available to you on an erasable board. The main point to remember is to use correct connecting lines to connect two types. Use an arrow to show an IS-A relationship and a line to show a HAS-A relationship.

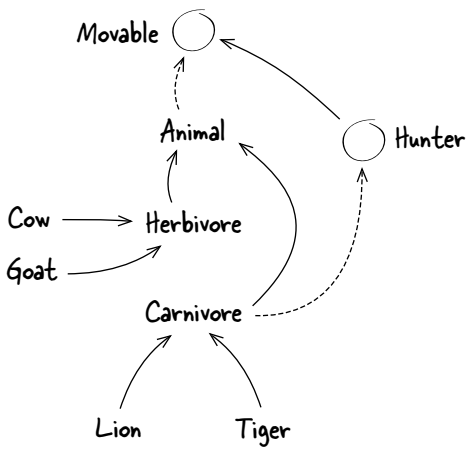


Figure 3.7 A rather raw form of the UML diagram that you may draw on an erasable board while taking your exam to represent an IS-A relationship between classes and interfaces

When I attempted this exam, I drew similar not-so-good-looking diagrams. Believe me, they helped me answer questions quickly, without referring to the code again and again. Also, the questions on the exam may not use names that indicate an obvious relationship between classes and interfaces. The next "Twist in the Tale" exercise will ensure that you get the hang of this point.

Twist in the Tale 3.2

Using the following code

```
interface InterH {}
interface SameY extends InterH {}

class JamD implements InterH {}
class SunP extends JamD {}
class BreaU extends JamD implements SameY {}
```

your task is to identify which of the following statements are true:

a `SunP` IS-A `InterH`.
b `JamD` IS-A `SameY`.
c `InterH` IS-A `InterH`.
d `SameY` IS-A `JamD`.

First attempt the exercise without drawing a UML diagram, and then by drawing and using a UML diagram. Do you think using the UML diagram helps you answer the questions more quickly?

EXAM TIP The key to finding the types that participate in an IS-A relationship is to find your way, up the hierarchy tree, in the direction of the arrows. This technique will not only help you with the exam, but also take you a long way in your professional career.

IMPLEMENTING AN IS-A RELATIONSHIP

You can implement an IS-A relationship by extending classes or interfaces, or by implementing interfaces. Here is a quick set of rules for implementing inheritance between classes and interfaces in code:

- A class inherits another class by using the keyword `extends`.
- A class implements an interface by using the keyword `implements`.
- An interface inherits another interface by using the keyword `extends`.

How will you implement the following relationship in code?

```
Herbivore IS-A Animal
```

Because you don't know whether either `Herbivore` or `Animal` refers to a class or an interface, you have the following possibilities:

- `Herbivore` and `Animal` are classes. `Herbivore` extends `Animal`.
- `Herbivore` and `Animal` are interfaces. `Herbivore` extends `Animal`.
- `Herbivore` is a class, and `Animal` is an interface. `Herbivore` implements `Animal`.

Figure 3.8 shows these three possible implementations.

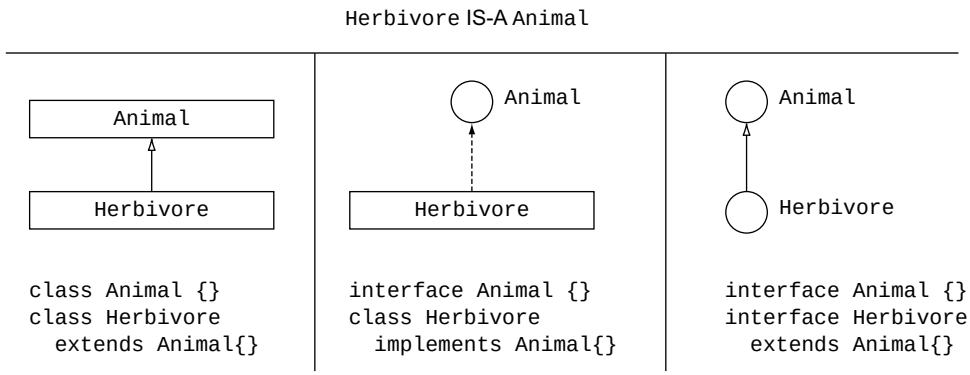


Figure 3.8 How to implement an IS-A relationship, if you don't know whether the relationship is between classes, interfaces, or both

Now, let's add another relationship to the previous one. How would you implement the following relationship and rules in code?

- `Herbivore` IS-A `Animal`.
- `Carnivore` IS-A `Animal`.
- `Animal` can define only abstract methods and constants.

The third rule makes it clear that `Animal` is an interface. But you still don't know whether `Herbivore` and `Carnivore` are classes or interfaces. So you can have the following possibilities:

- `Herbivore` and `Carnivore` are interfaces that extend the interface `Animal`.
- `Herbivore` and `Carnivore` are classes that implement the interface `Animal`.
- `Herbivore` is a class that implements the interface `Animal`. `Carnivore` is an interface that extends the interface `Animal`.
- `Herbivore` is an interface that extends the interface `Animal`. `Carnivore` is a class that implements the interface `Animal`.

These relationships can be implemented as shown in figure 3.9.

The exam may specify a similar set of rules and ask you to choose the code that you think correctly implements the specified conditions. Let's work through another set

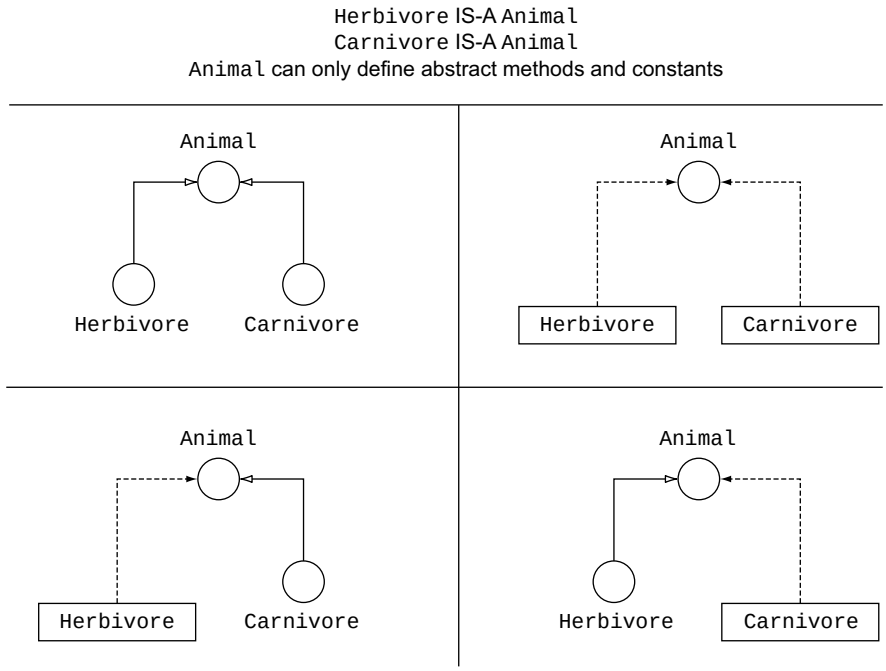


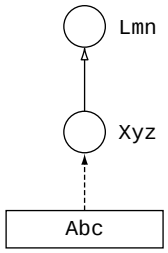
Figure 3.9 How to implement an IS-A relationship between three entities, one of which is an interface

of rules and implement the relationships in code. How would you implement the following relationships and rules in code?

1. `Abc` IS-A `Xyz`.
2. `Abc` defines methods and instance variables.
3. `Xyz` can declare only abstract methods.
4. `Xyz` IS-A `Lmn`.

Rule 2 states that `Abc` is a class, because an interface can't define instance variables. Rule 3 states that `Xyz` is an interface, because a class can declare both abstract and nonabstract methods. When you go up the hierarchy tree of an interface, you can find only another interface. In other words, `Lmn` is also an interface. The preceding rules evaluate to the following:

- `Abc` is a class.
- `Xyz` and `Lmn` are interfaces.
- `Abc` implements `Xyz`.
- `Xyz` extends `Lmn`.




```
interface Lmn{}
interface Xyz extends Lmn{}
class Abc implements Xyz{}
```

Figure 3.10 Implementing a set of rules and an IS-A relationship between three entities: two interfaces, and a class

After the evaluation, these rules seem simple to implement. Figure 3.10 shows the relationships in UML notation and in code.

When a class defines an instance variable of another class, they share a HAS-A relationship, covered in the next section.

3.3.2 Identifying and implementing a HAS-A relationship

As compared to an IS-A relationship, a *HAS-A relationship* is easy to identify and implement. I hope this statement relieves you! Consider this definition of the bare-bones class `Engine`:

```
class Engine {}
```

Which of the following classes (`Statistics`, `Car`, `PartsFactory`, `TestCar`) do you think shares a HAS-A relationship with class `Engine`?

```
class Statistics {
    static Engine engine;
}
class Car {
    Engine engine;
}
class PartsFactory {
    Object createEngine() {
        Engine engine = new Engine();
        //.. code
        return engine;
    }
}
class TestCar {
    boolean testEngine(Engine engine) {
        //.. code
    }
}
```

Statistics defines class variable of type Engine.

Car defines instance variable of type Engine.

PartsFactory defines local variable of type Engine.

TestCar defines method parameter of type Engine.

Of all the preceding classes—`Statistics`, `Car`, `PartsFactory`, and `TestCar`—only `Car` shares a HAS-A relationship with the class `Engine` because `Car` defines an instance variable of type `Engine`. Note that it doesn't matter whether the instance variable `engine` in class `Car` is initialized with an object. The HAS-A relationship is shared by the classes.

EXAM TIP Classes and interfaces can share a HAS-A relationship with each other. If a class or interface—say, `Type1`—defines an instance variable of a class or interface—say, `Type2`, `Type1` HAS-A `Type2` is correct. The reverse isn't correct. Also, the HAS-A relationship is shared by classes, and so the relationship isn't affected, whether the instance variable is initialized or not.

The exam doesn't stop at the IS-A and HAS-A relationships. Let's see how *high cohesion* and *low coupling* can improve your application design.

3.4 Cohesion and low coupling

[3.3] Apply cohesion, low-coupling, IS-A, and HAS-A principles

Focused teams and team members are known to deliver better results. On the other hand, highly dependent departments, teams, or team members might perform poorly. The same principles can be applied to application design. Focused classes and modules (cohesion) that aren't highly dependent (or coupled) on other classes or modules are generally easy to work with, reusable, and maintainable. Let's start with the design principle cohesion, which supports creation of focused modules and classes.

3.4.1 Cohesion

Cohesion refers to how focused a class or a module is. *High cohesion* refers to a well-focused class or module, whereas *low cohesion* refers to a class or module that doesn't have a well-defined responsibility. Such modules or classes might perform multiple actions, which could have been assigned to separate classes.

Imagine a book editor who is supposed to edit book content, manage the book-printing process, and reach out to new authors for new book ideas. Let's define this editor by using a class, say, `Editor`:

```
class Editor{
    public void editBooks() {}
    public void manageBookPrinting() {}
    public void reachOutToNewAuthors() {}
}
```

Low cohesion; Editor is performing diverse set of unrelated tasks

Because this editor is managing multiple tasks over a period of time, managing all these processes might become difficult. Also, working with multiple responsibilities can prevent the editor from specializing in *all* these processes. Let's limit the tasks to the book-editing process:

```
class Editor{
    public void useEditTools() {}
    public void editFirstDraft() {}
    public void clearEditingDoubts() {}
}
```

High cohesion; Editor is performing multiple but related tasks.

The preceding example creates a highly cohesive class, `Editor`. Highly cohesive classes are easy to use. In the preceding example, class `Editor` provides a one-stop solution for all editing tasks. Highly cohesive classes are also easy to maintain and reuse; whenever you need to add or modify any editing-related process, you know which class you need to refer to: class `Editor`.

EXAM TIP Well-designed applications aim for highly cohesive classes and modules.

Classes and modules also perform better if they are least affected by the changes made to other classes or modules. Let's work with this aspect in detail in the next section on coupling.

3.4.2 Coupling

Coupling refers to how much a class or module knows about other classes or modules. If a class—say, `Editor`—interacts with another class—say, `Author`—by using its interface (public methods), then classes `Editor` and `Author` are *loosely coupled*. But if class `Editor` can access and manipulate `Author` by using its nonpublic members, these classes are *tightly coupled*.

Let's code the class `Author`:

```
class Author {
    String name;
    String skypeID;
    public String getSkypeID() {
        return skypeID;
    }
}
```

NOTE The terms *low coupling* and *loose coupling* refer to the same concept. They are often used interchangeably.

The modified class `Editor` is tightly coupled with class `Author`. The method `clearEditingDoubts` in class `Editor` accesses the nonpublic member `skypeId` of class `Author`:

```
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.skypeID);
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Tight coupling; nonpublic variable skypeID is referred to outside its class Author.

What happens, say, if a programmer changes the name of the variable `skypeID` in class `Author` to `skypeName`? The code of class `Editor` won't compile. As long as the public interface of a class remains the same, it's free to change its implementation details. In

this case, the name of instance variable `skypeID` forms part of `Author`'s implementation details. One suggested solution is to use the public method `getSkypeID()` in class `Editor` (changes in bold):

```
class Author {
    String name;
    String skypeName;
    public String getSkypeID() {
        return skypeName;
    }
}
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.getSkypeID());
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Change in instance variable name won't affect classes that access this method

Loose coupling; public method getSkypeID() accesses Author's skypeName.

Interfaces also promote loose coupling across classes and modules. Assume that the entity `Author` is defined as an interface, which can be implemented by specialized authors such as `TechnicalAuthor`. Here's the new arrangement:

```
interface Author {
    String getSkypeID();
}
class TechnicalAuthor implements Author{
    String name;
    String skypeName;
    public String getSkypeID() {
        return skypeName;
    }
}
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.getSkypeID());
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

❶ **interface Author**

❷ **Class TechnicalAuthor implements Author.**

❸ **Loose coupling; method clearEditingDoubts() uses interface to access concrete implementations**

The code at ❶ defines the entity `Author` as an interface. At ❷, class `TechnicalAuthor` implements `Author`. At ❸, the type of parameter passed to method `clearEditingDoubts()` is the interface `Author`. So method `clearEditingDoubts()` is guaranteed to access only public members of instances of `Author`. Also, because method `clearEditingDoubts()` can be passed objects of classes that implement `Author`, it can also accept instances of classes that are created later, such as `FictionWriter`, which implement `Author`.

EXAM TIP Well-designed applications aim for loosely coupled classes and modules.

The tips for creating well-designed applications don't end here. The next section covers object composition principles.

3.5 Object composition principles

[3.4] Apply object composition principles (including HAS-A relationships)

How can you *use* the existing functionality of a class? Inexperienced programmers or newcomers to the Java programming language and OOP often answer this question by saying, "inheritance." They shouldn't be completely blamed for this incorrect answer. Many books, articles, and programmers overemphasize inheritance—which is correct in a way, because inheritance is an important concept. But this might leave a lot of newcomers with the wrong impression that inheriting a class is the best way to use another class. Most of the time, you can use another class by *composing* your own class with an object of another class. Let's start with a quick example:

```
class Engine { /* code */ }
class Wheel { /* code */ }
class Car {
    Engine engine;
    Wheel[] wheels = new Wheel[5];
}
```

Car is composed of Engine and Wheel.

EXAM TIP Object composition enables you to use the existing functionality of classes without extending them. The approach is simple: create and use objects of other classes in your own class.

Look around for examples of classes defined by your peers, in books or articles, or in the Java API. You'll be amazed to notice that composition is *the* way to use a class, when you want to use the functionality of any other class. You should inherit a class only when you think that the derived class is a type of its base class. For example, it's correct to say that `RacingCar` is a type of `Car`. But it's incorrect to say that `Engine` is a type of `Car`.

There's another reason to favor object composition over inheritance: a base class is *fragile* (refer to the subsection "Fragile Derived Classes" in section 3.2.3 for an example). A change to a base class can have major effects on all its derived classes. For example, changing the method signature of a public method in a base class can lead to broken code of all its derived classes. A change in the nonpublic variables or methods of a base class can affect its derived classes, *if* the variables or methods are used by the derived classes.

The remaining sections cover the design patterns on the exam. Before you dive into the details of the design patterns, let's look at what they are and why we need them.

3.6 Introduction to design patterns

People who live in regions that experience snowfall build sloping roofs so that snow and ice don't accumulate on the rooftops. This "pattern" of designing sloping roofs was identified after multiple persons faced *similar* difficulties and found *similar* solutions. Now this is an established practice. Being ignorant about the design pattern of building a sloping roof can cause you a lot of rework later. Similarly, in the computing domain, multiple design patterns have been documented by observing recurring programming, behavioral, or implementation issues.

3.6.1 What is a design pattern?

A *design pattern* identifies a specific problem and suggests a solution to it. It's neither ready-made code that you can drop in your projects nor a framework to use. For example, you *might* document the *sloping-roof* design pattern as

- *Design pattern name:* Sloping roof
- *Problem:* Accumulation of snow and ice on rooftops
- *Suggested solution:* Build sloping roofs for all houses, offices, and buildings in areas that receive snowfall during any time of the year. This enables snow or ice from rooftops to slide and fall to the ground.

Notice the design pattern doesn't include actual materials or tools to build a house.

NOTE No formal format of documentation of a design pattern exists. You can document it the way you like.

3.6.2 Why do you need a design pattern?

Design patterns offer *experience reuse* and not *code reuse*. Design patterns help you reuse the *experience* of application designers and developers in terms of the guidelines that you can follow to implement commonly occurring programming scenarios. By using design patterns for known issues in your application, you'll benefit from the experience of others and be less likely to reinvent the wheel.

NOTE The Singleton and Factory design patterns are creational patterns initially described in the Gang of Four (GoF) book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et al. (Addison-Wesley, 1995). DAO is an integration-tier core J2EE pattern; see *Core J2EE Patterns: Best Practices and Design Strategies*, Second Edition, by Deepak Alur, John Crupi, and Dan Malks (Prentice Hall, 2003).

In the next section, we'll cover the first design pattern that is covered on this exam: the Singleton design pattern.

3.7 Singleton pattern

[3.5] Design a class using the Singleton design pattern

Singleton is a creational design pattern that ensures that a class is instantiated only once. The class also provides a global point of access to it.

So under what conditions would you want to have only one object of a class? Wouldn't the object feel lonely because there's only one of its kind?

3.7.1 Why do you need this pattern?

Imagine the issues that can be caused by multiple browser caches or multiple thread pools. In these scenarios, you might need only one object of a class to encapsulate all operations for managing a pool of resources, and to also serve as a global point of reference. Other common examples include a single instance of Device Manager to manage all the devices on your system, and a single instance of a print spooler to manage all the printing jobs.

3.7.2 Implementing the Singleton pattern

Implementation of the `Singleton` class involves a single class. But don't let this simplicity dismiss the finer details that you should get right. Let's move on to the basics of implementing the Singleton pattern:

1. Define a private constructor for the class that implements the Singleton pattern. To prevent any other class from creating an object of this class, mark the constructor of this class as a `private` member:

```
class Singleton {
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

   Now, no class can execute `new Singleton()` to create an instance of this class. But if no other class can create objects of this class, how will they use it? The class that implements the Singleton pattern creates and manages its sole instance by defining a static variable to store this instance.

2. Define a private static variable to refer to the *only* instance of the `Singleton` class. A static variable ensures that the class stores and accesses the same instance. In the following code, the variable `anInstance` is a class variable:

```
class Singleton {
    private static Singleton anInstance = null;
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

A well-encapsulated class should enable access to its members by using well-defined interfaces. So let's create a method to access the private variable `anInstance`.

3 Define a public static method to *access* the only instance of the `Singleton` class. Before you access the variable `anInstance`, you should create it. The creation and return of this variable is usually defined as follows (additions to previous code in bold):

```
class Singleton {
    private static Singleton anInstance = null;
    public static Singleton getInstance() {
        if (anInstance == null)
            anInstance = new Singleton();
        return anInstance;
    }
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

If anInstance hasn't been initialized

Initialize anInstance

Return anInstance

A class can request an object of class `Singleton` by calling the static method `getInstance()`:

```
class UseSingleton {
    public static void main(String args[]) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
        System.out.println(singleton1 == singleton2);
    }
}
```

New instance of class Singleton created and returned when accessed first time

Previously created instance returned for subsequent calls to method getInstance()

Prints "true"

The output of this code confirms that an object of class `Singleton` is created only once:

```
Private Constructor
true
```

These steps ensure that only one object of class `Singleton` ever exists. But what happens if multiple classes request an object of class `Singleton` at exactly the same time? This *may* lead to the creation of more than one object of class `Singleton`. Don't worry; we have ways to fix this one too, as discussed in the next section.

3.7.3 Ensuring creation of only one object in the Singleton pattern

Though the previous code *seems* to guarantee that only one instance of `Singleton` will be created, concurrent access of method `getInstance()` may result in creation of multiple instances. This can be a problem in multithreaded environments, such as application servers and servlet engines. Before you fix the issue of concurrent creation

of an object in a Singleton pattern, you need to ensure that you understand the finer details of this issue. So let's get started.

UNDERSTANDING THE PROBLEM OF CONCURRENT ACCESS

Imagine that two objects request class `Singleton` to return its instance at *exactly the same time*, by calling method `getInstance()`:

```
class Singleton {
    private static Singleton anInstance = null;
    public static Singleton getInstance() {
        if (anInstance == null)
            anInstance = new Singleton();
        return anInstance;
    }
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

Because each call will discover that the variable `anInstance` hasn't been initialized, each method call will create a new object and assign it to the variable `anInstance`. Method `getInstance()` may also return separate objects for each call. This is shown in figure 3.11.

If you think that it doesn't make a difference if you create multiple objects of a class that implements the Singleton pattern, think again. The definition of the previously defined class `Singleton` is oversimplified. Real classes that implement the Singleton pattern define much more meaningful code in their constructors—for example, initializing their own resources, starting threads, or creating database or network connections. Obviously, a class won't like to do all these again, when it's not supposed to.

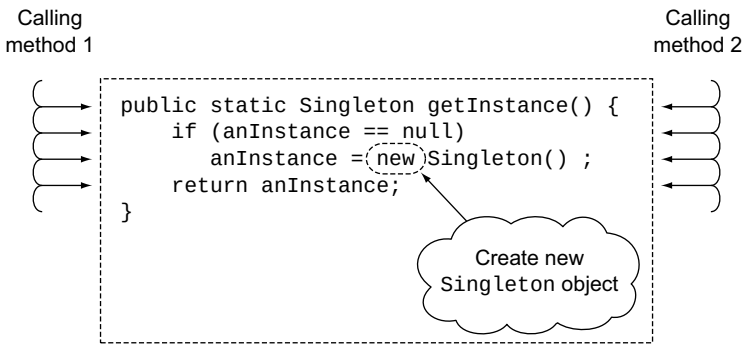


Figure 3.11 Multiple concurrent calls to method `getInstance()` can create multiple objects of class `Singleton`.

FIXING CONCURRENT CREATION: EAGER INITIALIZATION

There are multiple ways to ensure that an object of a class that implements the Singleton pattern is initialized only once. To begin with, *eager initialization* will enable you to initialize the static variable as soon as the class is loaded:

```
class Singleton {
    private static final Singleton anInstance = new Singleton();
    public static Singleton getInstance() {
        return anInstance;
    }
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

Eager initialization; anInstance is initialized as soon as class loaded ❶

Simply returns anInstance ❷

The code at ❶ executes when the class is loaded by the Java class loaders. So an object of class `Singleton` is created before any class requests it. When any other object requests an object of class `Singleton`, using method `getInstance()`, the code at ❷ simply returns the Singleton instance `anInstance`. The preceding code ensures that multiple objects of class `Singleton` aren't created.

FIXING CONCURRENT CREATION: SYNCHRONIZED LAZY INITIALIZATION

Though this seems to be the perfect solution, eager initialization creates an object of class `Singleton`, even if it's never used. Don't worry; every problem has a solution. Let's not employ eager initialization and synchronize method `getInstance()`:

```
class Singleton {
    private static Singleton anInstance;
    synchronized public static Singleton getInstance() {
        if (anInstance == null)
            anInstance = new Singleton();
        return anInstance;
    }
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

No eager initialization

Method getInstance() defined as synchronized method ❶

Method `getInstance()` is defined as a synchronized method at ❶. This means that multiple threads or objects can't execute this method concurrently. So this again saves us from multiple-object creation of a class implementing the Singleton pattern. If you're thinking that this is the last way to fix multiple-object creation issues for a `Singleton` class, take a deep breath, my friend, because there's more to it.

On the exam, you might also see a variation of the previously defined synchronized method `getInstance()`. Because synchronized methods don't allow concurrent execution, your application may feel a performance hit if a lot of classes in your application call method `getInstance()`. Java can rescue you this time, by synchronizing

method `getInstance()` partially (if you're new to threading and concurrency and can't understand the following code, don't worry. Just refer to chapters 10 and 11 on threading and concurrency):

```
public static Singleton getInstance() {
    if (anInstance == null) {
        synchronized (Singleton.class) {
            if (anInstance == null)
                anInstance = new Singleton();
        }
    }
    return anInstance;
}
```

Don't synchronize complete method

Synchronize code block that creates new object

After the thread acquires a lock on `Singleton.class` and enters the synchronized block, the code checks whether `anInstance` is `null` (again), before creating a new object. This is to ensure that after the lock is acquired, the condition hasn't changed and `anInstance` is still `null`.

EXAM TIP On the exam, all of these approaches (eager initialization, synchronization of the complete method `getInstance()`, and partial synchronization of method `getInstance()`) may be presented, and you may be questioned about the right approach for implementing the Singleton pattern. All these approaches are good. Beware of modified code that tries to synchronize a partial `getInstance()` method, which doesn't synchronize the code that creates an object of `Singleton`.

USING ENUMS

By using enums, you can implement the Singleton pattern in a thread-safe manner. Here's a simple implementation:

```
public enum Singleton {
    INSTANCE;

    public void initCache(){
        //..code
    }
}
```

Because enum instances can't be created by any other class, the enum `Singleton` will ensure the existence of only *one* of its instances, `Singleton.INSTANCE`.

NOTE Even though using a single-element enum is the best way to implement the Singleton pattern, you must know all the previously discussed approaches to answer questions on this topic on the exam.

After making yourself aware of the multiple rules that you need to follow to apply the Singleton pattern, test yourself on it with the next "Twist in the Tale" exercise.

Twist in the Tale 3.3

Does the class in the following code apply the Singleton pattern correctly?

```
class Singleton {
    private Singleton anInstance;
    synchronized public Singleton getInstance() {
        if (anInstance == null)
            anInstance = new Singleton();
        return anInstance;
    }
}
```

NOTE The Singleton pattern is also referred to as an anti-pattern. It has been overused by developers and designers, who make a lot of assumptions about the applications that use it. It also makes testing difficult.

Even before the Singleton pattern was officially recognized and used, *single-object instances with global access* have been implemented using static variables. But this has its own set of disadvantages.

3.7.4 Comparing Singleton with global data

Programmers have been creating and using single instances of a class by defining them as static variables for quite a long time. Some of them do that even now. But doing so requires the following serious considerations:

- *Possibility of creating multiple objects of the same type*—Using a static variable doesn't stop you (or any other user) from creating another object of the class and referring to it by another name. Limiting creation of only one object is the responsibility of the application developer and isn't included as part of the class design in this case. This, as you know, can introduce issues when multiple (unwanted) objects are created.
- *Eager initialization*—Static variables are usually initialized before any class uses them. This risks allocation of resources and other processing that may never have been required or used (for example, initializing resources of the class used as a global variable) and other tasks that it may define in its constructor (for example, starting threads, or creating database or network connections).
- *Pollution of namespace*—Using multiple static variables within an application is sure to pollute the namespace, which is, again, not a preferred approach.

The API of a language, product, or service can be huge, and it isn't possible for users to know about all its classes. It makes a lot of sense to be able to create and use objects of a class by specifying a set of requirements. The Factory pattern makes this feasible. Apart from hiding the implementation details of object creation, it enables developers to extend an API and users to use the newer classes.

3.8 Factory pattern

[3.7] Design and create objects using a factory pattern

Imagine you need to open files, say, Hello.doc and Hello.xml, programmatically using your Java application. To do so, you'd need instances of classes, say, `WordProcessor` and `TextEditor`, that can open these files. One of the obvious approaches is to use the operator `new` to create an instance of `WordProcessor` and `TextEditor` to open files. But this would result in tight coupling between the application that opens files and the classes that are used to open the files. What happens if you need to use another class, say, `QuickProcessor`, to open .doc files in the future?

In this section, you'll work with how to use the Factory pattern to prevent tight coupling between classes. This pattern also eliminates direct constructor calls in favor of invoking a method. One of the most frequently used design patterns, multiple Factory patterns exist:

- Simple Factory or Static Factory pattern
- Factory Method pattern
- Abstract Factory pattern

On the exam, most of the questions on the Factory pattern refer to the Simple Factory pattern.

3.8.1 Simple Factory pattern (or Static Factory pattern)

This pattern creates and returns objects of classes that extend a common parent class or implement a common interface. The objects are created without exposing the instantiation logic to the client. The calling class is decoupled from knowing the exact name of the instantiated class.

Figure 3.12 shows the UML class diagram for classes used in the sample code to exhibit the Simple Factory pattern.

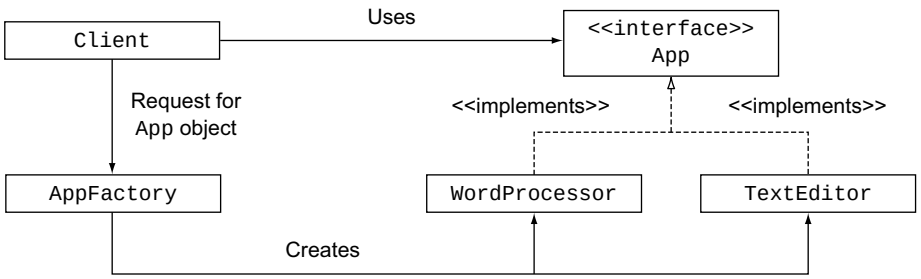


Figure 3.12 UML class diagram for interfaces and classes implementing the Simple Factory pattern

In the following sample code, class `Client` is decoupled from knowing the exact subclass, either `WordProcessor` or `TextEditor`, that it needs to open a file. It calls `AppFactory`'s method `getAppInstance()`, passing it the file extension, which returns an appropriate `App` object. It gives you the flexibility of modifying class `AppFactory` without impacting class `Client`. You might want to modify method `getAppInstance()` to return another `App` instance, say, `XMLEditor`, to open a .txt file. For instance

```
interface App {
    void open(String filename);
}
class WordProcessor implements App {
    public void open(String filename) {
        System.out.println("Launch WordProcessor using " + filename);
    }
}
class TextEditor implements App {
    public void open(String filename) {
        System.out.println("Launch TextEditor using " + filename);
    }
}
class AppFactory {
    public static App getAppInstance(String fileExtn) {
        App appln = null;
        if (fileExtn.equals(".doc")) {
            appln = new WordProcessor();
        }
        else if (fileExtn.equals(".txt") ||
            fileExtn.equals(".xml")) {
            appln = new TextEditor();
        }
        return appln;
    }
}
class Client{
    public static void main(String args[]) {
        App app = AppFactory.getAppInstance(".doc");
        app.open("Hello.doc");
        App app2 = AppFactory.getAppInstance(".xml");
        app2.open("Hello.xml");
    }
}
```

Interface App implemented by classes WordProessor and TextEditor

Implements Simple Factory pattern by returning App object according to parameter value

Client is decoupled from classes TextEditor and WordProcessor; it calls AppFactory.getApp-Instance() to get App object.

NOTE Because method `getAppInstance()` in class `AppFactory` is a static method, this pattern is also referred to as the Static Factory pattern or Factory Class pattern.

Method `getAppInstance()` is manageable with just a few comparisons (`if-else`) statements. What happens if method `getAppInstance()` is supposed to return `App` instances for a wide variety of file extensions? Because it can become unmanageable, let's work with a variation of the Simple Factory pattern—that is, the Factory Method pattern.

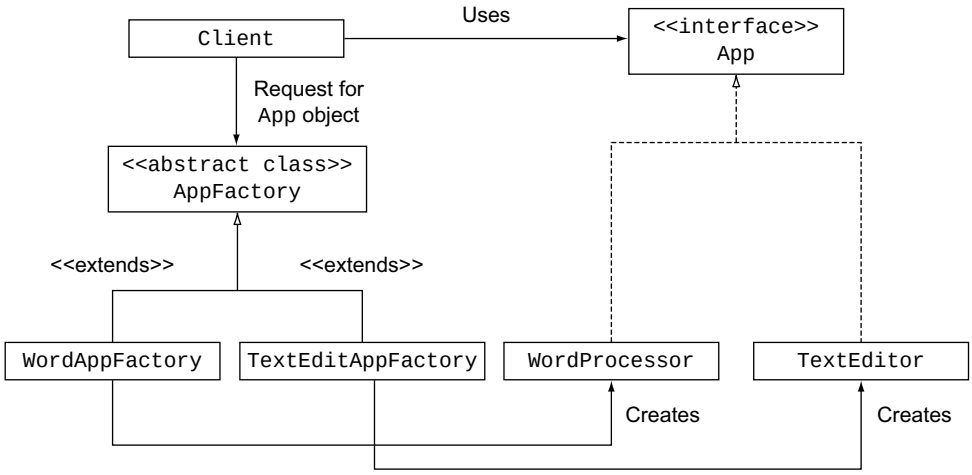


Figure 3.13 UML class diagram for interfaces and classes implementing the Factory Method pattern

3.8.2 Factory Method pattern

The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses.

Figure 3.13 shows the UML class diagram for classes used in the sample code to exhibit the Factory Method pattern.

In the following example code, to get the required `App` object, class `Client` uses one of the subclasses of abstract class `AppFactory` and calls its method `getAppInstance()`. Method `getAppInstance()` calls method `getApp()`, which is a factory method defined as an abstract method in class `AppFactory`. It is implemented by its concrete factory classes, `WordAppFactory` and `TextEditAppFactory`. The subclasses `WordAppFactory` and `TextEditAppFactory` implement `getApp()` to return a specific `App` object. It allows `Client` to use `WordProcessor` and `TextEditor`, but decouples it from knowing their names. This arrangement promotes flexibility to change the `App` object returned from concrete factory classes (`WordAppFactory` and `TextEditAppFactory`):

```
interface App {
    void open(String filename);
}
class WordProcessor implements App {
    public void open(String filename) {
        System.out.println("Launch WordProcessor using " + filename);
    }
}
class TextEditor implements App {
    public void open(String filename) {
        System.out.println("Launch TextEditor using " + filename);
    }
}
```

Interface App implemented by classes Word-Processor and TextEditor

```
abstract class AppFactory {
    public App getAppInstance() {
        App appln = getApp();
        return appln;
    }
    public abstract App getApp();
}
class WordAppFactory extends AppFactory {
    public App getApp() {
        return new WordProcessor();
    }
}
class TextEditAppFactory extends AppFactory {
    public App getApp() {
        return new TextEditor();
    }
}
class Client {
    public static void main(String args[]) {
        AppFactory factory = new WordAppFactory();
        App app = factory.getAppInstance();
        app.open("Hello.doc");

        app = new TextEditAppFactory().getAppInstance();
        app.open("Hello.xml");
    }
}
```

Abstract Factory class

Factory Method

Concrete Factory classes

Implement Factory Method to return a specific App object.

Client class uses variable of Abstract Factory pattern to refer to a concrete factory object

Because factory refers to WordAppFactory object, call to getAppInstance() returns WordProcessor object

Now, what happens if you were required to create *families* of related classes, such as applications that can open rich format files for editing in Windows and Mac systems? Because these systems might use separate applications for similar purposes, let's modify the example used in the previous section to use the Abstract Factory pattern.

3.8.3 Abstract Factory pattern

The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object). This pattern also defines an interface for creating objects, but it lets subclasses decide which class to instantiate.

Figure 3.14 shows the UML class diagram for classes used in the sample code to exhibit the Abstract Factory pattern.

In the following example code, to get the required `App` and `Font` objects, class `Client` uses one of the subclasses of abstract class `AppFactory` and calls its methods `getAppInstance()` or `getFontInstance()`. The Concrete Factory pattern classes `WordAppFactory` and `TextEditAppFactory` return an appropriate concrete object for the interfaces `App` and `Font`. This pattern also allows `Client` to use `WordProcessor`, `TextEditor`, `RichFont`, and `RegularFont`, but decouples it from knowing their names. This arrangement also promotes flexibility to change the `App` or `Font` object

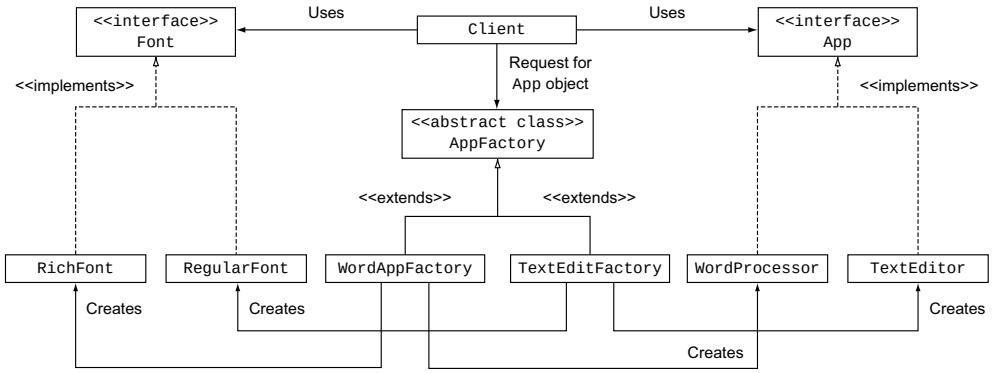


Figure 3.14 UML class diagram for interfaces and classes implementing the Abstract Factory pattern

returned from Concrete Factory pattern classes (`WordAppFactory` and `TextEditAppFactory`).

```
interface App { /* code */ }
class WordProcessor implements App { /* code */ }
class TextEditor implements App { /* code */ }
```

interface App implemented by classes WordProcessor and TextEditor

```
interface Font { /* code */ }
class RichFont implements Font { /* code */ }
class RegularFont implements Font { /* code */ }
```

interface Font implemented by classes RichFont and RegularFont

```
abstract class AppFactory {
    protected abstract App getApp();
    protected abstract Font getFont();

    public App getAppInstance() {
        App appln = getApp();
        return appln;
    }
    public Font getFontInstance() {
        Font font = getFont();
        return font;
    }
}
```

Abstract Factory class

```
class WordAppFactory extends AppFactory {
    protected App getApp() {
        return new WordProcessor();
    }
    protected Font getFont() {
        return new RichFont();
    }
}

class TextEditAppFactory extends AppFactory {
    protected App getApp() {
        return new TextEditor();
    }
```

Concrete Factory classes

```
    protected Font getFont() {
        return new RegularFont();
    }
}

class ClientAbstractFactoryMethod {
    public static void main(String args[]) {
        AppFactory factory1 = new WordAppFactory();
        App app1 = factory1.getAppInstance();
        Font font1 = factory1.getFontInstance();
        System.out.println(app1 + ":" + font1);

        AppFactory factory2 = new TextEditAppFactory();
        App app2 = factory2.getAppInstance();
        Font font2 = factory2.getFontInstance();
        System.out.println(app2 + ":" + font2);
    }
}
```

Concrete Factory class used to create objects of App and Font

NOTE The sample code used in the sections on Factory Method and Abstract Factory patterns can use `App` or `Font` as an interface or abstract or concrete class.

In the next section, you'll learn the terms and phrases that the exam uses to test you on the benefits of using the Factory pattern.

3.8.4 Benefits of the Factory pattern

The exam won't ask you to select the benefit of a specific Factory pattern—that is, Simple Factory, Factory Method, or Abstract Factory. Here are the benefits that apply to all Factory patterns:

- Prefer method invocation over direct constructor calls
- Prevent tight coupling between a class implementation and your application
- Promote creation of cohesive classes
- Promote programming to an interface
- Promote flexibility. Object instantiation logic can be changed without affecting the clients that use objects. They also allow addition of new concrete classes.

Here's a list of what doesn't apply or isn't related to the Factory pattern:

- It doesn't eliminate the need of overloading constructors in class implementations.
- It doesn't encourage the use of any particular access modifier. It isn't compulsory to define private members to use this pattern.
- It won't slow your application.
- It isn't related to how to monitor objects for change.

The exam will question you on the classes from the Java API that use this pattern. Let's cover them in the next section.

3.8.5 Using the Factory pattern from the Java API

You'll find this important design pattern in multiple classes in the Java API.

Some of these classes are listed in table 3.3.

Table 3.3 Classes and methods from the Java API that use the Factory pattern

Class	Method	Description
`java.util.Calendar`	`getInstance()`	Gets a calendar using the default time zone and locale.
`java.util.Arrays`	`asList()`	Returns a fixed-size list backed by the specified array.
`java.util.ResourceBundle`	`getBundle()`	Overloaded versions of this method return a resource bundle using the specified base name, target locale, class loader, and control.
`java.sql.DriverManager`	`getConnection()`	Establishes and returns a connection to the given database URL.
`java.sql.DriverManager`	`getDriver()`	Attempts to locate and return a driver that understands the given URL.
`java.sql.Connection`	`createStatement()`	Overloaded version of this method creates a statement object for sending SQL statements to the database and generates `ResultSet` objects with the given type, concurrency, and holdability.
`java.sql.Statement`	`executeQuery()`	Executes the given SQL statement, which returns a single `ResultSet` object.
`java.text.NumberFormat`	`getInstance()` `getNumberFormat()`	Returns a general-purpose number format for the current default locale.
`java.text.NumberFormat`	`getCurrencyInstance()`	Returns a currency format for the current default locale.

Table 3.3 Classes and methods from the Java API that use the Factory pattern

Class	Method	Description
`java.text.NumberFormat`	`getIntegerInstance()`	Returns an integer format for the current default locale.

`java.util.concurrent.Executors`	`newFixedThreadPool()`
	`newCachedThreadPool()`
	`newSingleThreadExecutor()`
		Creates a thread pool.

NOTE Refer to chapter 12 for detailed coverage of the Factory method `getInstance()` defined in class `NumberFormat`.

Almost all applications need to store data to a persistent medium in one form or another. Data persistence can range from using simple text files to full-fledged database management systems. In the next section, we'll cover how the Data Access Object (DAO) pattern enables you to separate code that communicates with the data source from the classes that use the data.

3.9 DAO pattern

[3.6] Write code to implement the DAO pattern

Imagine your employee application needs to read its data from and write to multiple sources like flat files, relational databases, XML, or JSON. Add to this the differences in accessing the data for different vendor implementations. How would your application manage to work with data stored in a different format, with different data management systems, offering separate features, using separate APIs? This section shows you how the DAO pattern helps in a similar situation.

3.9.1 What is the DAO pattern?

The DAO pattern abstracts and encapsulates all access to a data store (flat files, relational databases, XML, JSON, or any other data source). It manages the connection with the data source to access and store the data. It shields a client from knowing how to retrieve or store data and lets it specify what data to retrieve and store. So it makes the client code flexible to work with multiple data sources.

EXAM TIP The DAO pattern decouples classes that define business or presentation logic from the data persistence details.

3.9.2 Implementing the DAO pattern

Identify the data that you need to store to or retrieve from a data store (say, `Emp`). Define an interface, a DAO, say, `EmpDAO`, to expose the data's CRUD operations. The

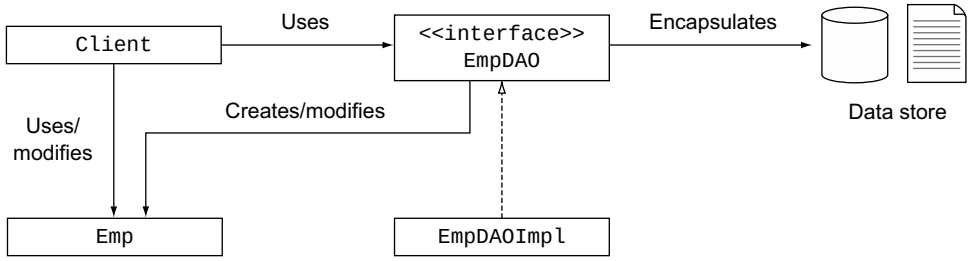


Figure 3.15 UML class diagram of classes and interfaces implementing the DAO pattern

implementation details are hidden from clients and defined in a class (say, `EmpDAOImpl`). If the implementation details to access data in the data source change, it doesn't affect a client. This pattern allows an application to adapt to different data stores or its version without affecting a client. Figure 3.15 shows the UML class diagram of classes implementing the DAO pattern.

In the following sample code, class `Emp` encapsulates the employee data that can be read from and stored to multiple types of data stores. The interface `EmpDAO` exposes the operations that can be performed with `Emp` objects in a data store. Class `EmpDAOImpl` implements `EmpDAO`, connecting to a data store and retrieving `Emp` from it and updating it in the data store. Note how class `Client` is decoupled from the data storage and retrieval details. Class `Client` works with `EmpDAO` and not with its specific implementation.

```
class Emp {
    int id;
    String name;
    int age;
    String address;
}
interface EmpDAO {
    public int create(Emp e);
    public Emp get(int id);
    public boolean delete(Emp e);
    public boolean update(Emp e);
    public Collection<Emp> getAll();
}
class EmpDAOImpl implements EmpDAO {
    public int create(Emp e) {
    /* connect to datastore, insert data for employee e */
    }
    public Emp get(int id) {
    /* connect to datastore, retrieve and return data for employee-id id */
    }
    public boolean delete(Emp e) {
    /* connect to datastore and delete data for employee-id e.id */
    }
```

```
    public boolean update(Emp e) {
    /* connect to datastore and update employee data */
    }
    public Collection<Emp> getAll() {
    /* connect to datastore, retrieve emp data, return as Collection */
    }
}
class Client {
    public static void main(String args[]) {
        Emp emp = new Emp();
        emp.id = 10; emp.name = "Harry";
        emp.age = 39; emp.address = "UK";

        EmpDAO dao = new EmpDAOImpl();
        dao.create(emp);
        emp.name = "Harry M"; emp.age = 40;
        dao.update(emp);

        Emp emp2 = dao.get(11);
        if (emp2 != null) dao.delete(emp2);
    }
}
```

Create Emp object.

Initialize emp object.

Create instance of EmpDAO.

Insert emp data in data store using method create().

Update emp object in data store.

Modify existing values of emp object.

Retrieve object from data store.

Delete data corresponding to retrieved data from data store.

EXAM TIP The CRUD operations form the basis of the DAO pattern.

The preceding example uses only one implementation of the DAO interface. How would an application manage working with multiple DAO implementations? You can use the Factory pattern—that is, Simple Factory, Factory Method, or Abstract Factory—with the DAO pattern to work with multiple DAO implementations, as shown in the next section.

EXAM TIP The exam might ask you whether it's common to use the Factory pattern with the DAO pattern. The answer is yes (as shown in the next section). But it isn't mandatory to use the Factory pattern with the DAO pattern (as shown in this section).

3.9.3 Using the Simple Factory pattern with the DAO pattern

You can use a Factory pattern to work with multiple DAO pattern implementations. The following example uses the Simple Factory pattern. Method `getInstance()` in class `DAOFactory` returns an instance of the `EmpDAO` implementation, which can be used by a client class (`Client`). I haven't repeated the class details deliberately for `Emp` and `EmpDAO` to keep the code short. They're the same as used in the example in the preceding section. The code for classes `EmpDAOOracleImpl` and `EmpDAOMySQLImpl` can be assumed to be the same as the code for class `EmpDAOImpl` used in the preceding example.

```
class Emp { /* code */ }
interface EmpDAO { /* code */ }
```

```
class EmpDAOOracleImpl implements EmpDAO { /* code */ }
class EmpDAOMySQLImpl implements EmpDAO { /* code */ }

abstract class DAOFactory {
    public static int ORACLE = 1;
    public static int MYSQL = 2;
    public static EmpDAO getEmpDAOInstance(int DBtype) {
        if (DBtype == ORACLE)
            return new EmpDAOOracleImpl();
        else if (DBtype == MYSQL)
            return new EmpDAOMySQLImpl();
        else
            return null;
    }
}

class Client {
    public static void main(String args[]) {
        EmpDAO empDAO = DAOFactory.getEmpDAOInstance(DAOFactory.ORACLE);
        Emp emp = new Emp();
        emp.id = 10; emp.name = "Harry";
        emp.age = 39; emp.address = "UK";
        empDAO.create(emp);
    }
}
```

EmpDAO implementation for Oracle Database

EmpDAO implementation for MySQL Database

DAOFactory uses Simple Factory pattern.

Static Factory pattern to return implementation of EmpDAO

Get EmpDAO implementation for Oracle DB.

Insert emp data in data store.

NOTE To keep the code simple, this and the next section use DAO implementation classes for only two different data stores, Oracle and MySQL.

In the next section, you'll see how you can decouple data storage and retrieval for multiple type of objects (`Emp`, `Dept`) from multiple data stores (Oracle, MySQL) using the Factory Method or Abstract Factory patterns with the DAO pattern.

3.9.4 Using the Factory Method or Abstract Factory pattern with the DAO pattern

In the following example, class `Client` needs to store and retrieve objects of `Emp` and `Dept` to and from a data store. To decouple `Client` from the persistence details, the interfaces `EmpDAO` and `DeptDAO` define all data store operations with `Emp` and `Dept` objects.

The example code defines implementation classes for Oracle and MySQL data stores. Classes `EmpDAOOracleImpl` and `DeptDAOOracleImpl` define implementation details for an Oracle data store and classes `EmpDAOMySQLImpl` and `DeptDAOMySQLImpl` define implementation details for a MySQL data store. The abstract class `DAOFactory` defines abstract methods `getEmpDAO()` and `getDeptDAO()`, which are implemented by its subclasses `OracleDAOFactory` and `MySQLDAOFactory`.

To store `Emp` and `Dept` objects to an Oracle database, class `Client` can use `OracleDAOFactory`, and to store them to a MySQL database, class `Client` can use `MySQLDAOFactory`:

```
class Emp { /* code */ }
class Dept { /* code */ }
```

Data objects to persist

```
interface EmpDAO { /* code */ }
interface DeptDAO { /* code */ }

class EmpDAOOracleImpl implements EmpDAO { /* code */ }
class DeptDAOOracleImpl implements DeptDAO { /* code */ }

class EmpDAOMySQLImpl implements EmpDAO { /* code */ }
class DeptDAOMySQLImpl implements DeptDAO { /* code */ }

abstract class DAOFactory {
    protected abstract EmpDAO getEmpDAO();
    protected abstract DeptDAO getDeptDAO();
    public EmpDAO getEmpDAOInstance() {
        return getEmpDAO();
    }
    public DeptDAO getDeptDAOInstance() {
        return getDeptDAO();
    }
}

class OracleDAOFactory extends DAOFactory {
    protected EmpDAO getEmpDAO() {
        return new EmpDAOOracleImpl();
    }
    protected DeptDAO getDeptDAO() {
        return new DeptDAOOracleImpl();
    }
}
class MySQLDAOFactory extends DAOFactory {
    protected EmpDAO getEmpDAO() {
        return new EmpDAOMySQLImpl();
    }
    protected DeptDAO getDeptDAO() {
        return new DeptDAOMySQLImpl();
    }
}

class Client {
    public static void main(String args[]) {
        DAOFactory factory = new OracleDAOFactory();
        EmpDAO empDAO = factory.getEmpDAOInstance();
        DeptDAO deptDAO = factory.getDeptDAOInstance();

        Emp emp = new Emp();
        empDAO.create(emp);

        Dept dept = new Dept();
        deptDAO.update(dept);
    }
}
```

DAO pattern

DAO pattern implementation for Oracle database

DAO pattern implementation for MySQL database

Abstract Factory pattern class

Factory to return Oracle DAO implementations

Factory to return MySQL DAO implementations

Create OracleDAOFactory.

Access EmpDAO and DeptDAO implementations.

Insert emp data in database.

Update dept data in database.

In the next section, you'll see what terms and phrases the exam might use to test you on the benefits of using the DAO pattern.

3.9.5 Benefits of the DAO pattern

The benefits of the DAO pattern are

- It abstracts and encapsulates all access to a data source. It manages the connection to the data source to obtain and store data.
- It promotes programming to an interface. It completely hides the data access implementation from its clients.
- It decouples the business logic layer and persistence layer. It makes the code independent of any changes to a data source or its vendor (for example, plain-text, XML, LDAP, MySQL, Oracle, or DB2).
- It promotes flexibility. Because the interfaces accessible to client classes don't change, new implementation classes can be added.
- The DAO pattern might also include Factory pattern classes.
- It prevents tight coupling between client classes and DAO implementation classes. It promotes the creation of cohesive classes.

Design patterns help you to reuse the experience of other programmers to create robust application designs. When you work with real-life projects, identify recurrent issues across projects and their probable solutions. You never know, you might identify and pen your own design patterns. Good luck to you!

3.10 Summary

In this chapter, you covered interfaces and how to use them in class design. Given the task of designing an application, API, or framework, you need to define multiple interfaces and abstract classes. But the choice of using abstract classes and interfaces isn't straightforward.

An interface can define only constants and abstract methods. The methods of an interface are implicitly abstract and can't define any implementation details. The interfaces are used to define a *contract*, which all the classes should adhere to. Interfaces only specify the behavior that should be supported by their implementing classes; the implementation details are left for the classes.

You can subclass an abstract class to create concrete classes only if you implement all of the class's abstract methods. Similarly, a concrete class should implement all the methods of an interface.

For the exam, you also need to know when to use interface inheritance and when to use class inheritance. Class inheritance helps you reuse implementation details provided in the base classes, so the derived classes don't have to write all the code themselves. Class inheritance also scores better when you want to add new behavior to an existing base class. You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes. Interface implementation has one major advantage of allowing a class to implement multiple interfaces, so an object of the class can be assigned to variables of multiple interface types.

Java objects share multiple relationships with other objects. IS-A and HAS-A are two important relationships shared by Java objects. The IS-A relationship is implemented using inheritance. You can implement the IS-A relationship by extending classes, extending interfaces, and implementing interfaces. When a class has an instance variable of a certain type, the class HAS-A <certain-type>.

Both inheritance and composition enable you to reuse the functionality of a class, but with a difference. Most often, newcomers to programming or OOP aren't sure whether to use inheritance or composition, to use another object. So they inherit a class when they want to use it in another class. You should use inheritance when the extended class is a specialized type of the base class. You should use composition when you simply want to use the functionality being offered by a class in another class.

Design patterns help you reuse the experience of other application designers and developers, in terms of the guidelines and suggested solutions for implementing an application's commonly occurring logic. A design pattern enables you to reuse experience and not code. We discussed all three design patterns that are on the exam: Singleton, DAO, and Factory method.

Singleton is a class design pattern that ensures that a class is instantiated only once. The class also provides a global point of access to it. This pattern is usually applied to only one class. Common examples include a single instance of Device Manager to manage all the devices on your system, or a single instance of print spooler to manage all printing jobs. To apply the Singleton pattern, you should mark the constructor of a class as `private` so that no other class can call it. Make the class itself create the sole instance, referred by a static variable. You can define a static method to access this sole instance.

The Factory pattern prevents tight coupling between the classes it uses from their concrete class implementation. It also eliminates direct constructor calls in favor of invoking a method. Multiple variations of this pattern exist: Simple Factory, Factory Method, and Abstract Factory.

The Simple Factory pattern creates and returns objects of classes that extend a common parent class or implement a common interface. The objects are created without exposing the instantiation logic to the client. The calling class is decoupled from knowing the exact name of the instantiated class. The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses. The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object). This pattern also defines an interface for creating objects, but it lets subclasses decide which class to instantiate.

You learned what the DAO pattern is and how to implement it in code. This pattern encapsulates all access to the persistent store to access and manipulate the data. The DAO pattern also manages the connection to the data store to retrieve and store the data. Usually, a DAO class accesses and manipulates a separate data object. An

application usually defines a separate DAO for separate data objects that should be persisted. You need this design pattern so you can decouple data access code from the business logic. This eases the transition from using various data storage formats and vendors and creates more cohesive classes. The DAO pattern is frequently used with the Factory pattern.

REVIEW NOTES

This section lists the main points covered in this chapter.

Interfaces

- An interface is an example of separating the behavior that an object should support from its implementation. An interface is used to define behavior by defining a group of abstract methods.
- All members (variables and methods) of an interface are implicitly public.
- You declare an interface using the keyword `interface`. An interface can define only public, final, static variables and public, abstract methods.
- The methods of an interface are implicitly abstract and public.
- The variables of an interface are implicitly public, static, and final.
- You can declare a top-level interface only with public and default access. Valid nonaccess modifiers that can be applied to an interface are `abstract` and `strictfp`.
- An interface that's defined within another interface can be defined with any access modifier.
- An interface can't extend a class.
- An interface can extend multiple interfaces. It can't implement another interface.
- An interface can define inner interfaces and (surprisingly) inner classes too.
- Because all the members of an interface are implicitly public, a derived interface inherits all the methods of its base interface.
- You can compare interface implementation to the signing of a contract. When a concrete class declares an implementation of an interface, it agrees to and must implement all its abstract methods.
- If you don't implement all the methods defined in the implemented interfaces, a class can't compile as a concrete class. A concrete class must implement all the methods from the interfaces that it implements. An abstract class might not implement all the methods from the interfaces that it implements.
- A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.
- Because the methods in an interface are implicitly public, if you try to assign a weaker access to the implemented method in a class, it won't compile.

- A class can inherit methods with the same name from multiple interfaces. There are no compilation issues if these methods have exactly the same method signature or if these methods can coexist in the implemented class as overloaded methods. The class won't compile if these methods coexist as incorrectly overloaded or overridden methods.

Class inheritance versus interface inheritance

- Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class.
- You can add new behavior to an abstract or nonabstract base class, and you may not break all the classes that subclass it.
- You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.
- Interface implementation has one major advantage of allowing a class to implement multiple interfaces, so an object of the class can be assigned to variables of multiple interface types.

IS-A and HAS-A relationships in code

- An IS-A relationship is implemented using inheritance.
- You can traverse the inheritance tree *up* the hierarchy to identify an IS-A relationship. A derived class IS-A *type* of its base class and its implemented interfaces. A derived interface IS-A type of its base interface. A base class or interface *is not a* type of its derived class or interface.
- The key to finding the entities that participate in an IS-A relationship is to find your way, up the hierarchy tree, in the direction of the arrows. This technique not only will help you with the exam, but also will take you a long way in your professional career.
- You can implement an IS-A relationship by extending classes, extending interfaces, or implementing interfaces.
- A HAS-A relationship is implemented using association.
- The relationship `MyClass` HAS-A `YourClass` is implemented by defining an instance variable of type `YourClass` in `MyClass`. Defining an instance variable of type `MyClass` in `YourClass` will implement the relationship `YourClass` HAS-A `MyClass`.

Cohesion and low coupling

- Cohesion refers to how focused a class or a module is.
- High cohesion refers to a well-focused class or module, whereas low cohesion refers to a class or module that doesn't have a well-defined responsibility.
- Well-designed applications aim for highly cohesive classes and modules.

- Coupling refers to how much a class or module knows about other classes or modules.
- Loosely coupled classes interact with each other by using their interface (public methods).
- Low coupling and loose coupling refer to the same concept and are often used interchangeably.
- Well-designed applications aim for loosely coupled classes and modules.

Object composition principles

- Newcomers to programming often extend a class when they want to use a class in another class. They use inheritance in place of composition.
- You should extend a class (inheritance) when you want the objects of the derived classes to reuse the interface of their base class.
- You should define an object of another class (composition) when you want to use the functionality offered by the class.

Singleton pattern

- Singleton is a creational design pattern that ensures that a class is instantiated only once. The class also provides a global point of access to it.
- It is used in scenarios when you might need only one object of a class.
- Implementation of the Singleton pattern involves a single class.
- A class that implements the Singleton pattern must define its constructor as `private`.
- A Singleton class uses a static private reference variable to refer to its sole instance.
- A Singleton class defines a static method to access its sole instance.
- To avoid threading issues with the creation of the sole instance of the Singleton class, you might use either of the following to create its sole instance:
  - Eager initialization—instantiate the object with its declaration
  - Synchronized lazy initialization—create the instance using a synchronized method or code block
- You can also use enums to implement the Singleton pattern because enum instances can't be created by any other class.
- On the exam, all of these approaches (eager initialization, synchronization of the complete method `getInstance()`, and partial synchronization of method `getInstance()`) may be presented, and you may be questioned about the right approach for implementing the Singleton pattern. All these approaches are good. Beware of modified code that tries to synchronize a partial method `getInstance()`, which doesn't synchronize the code that creates an object of `Singleton`.

Factory pattern

- One of the most frequently used design patterns, multiple flavors of this pattern exist: Simple Factory, Factory Method, and Abstract Factory.
- The Simple Factory pattern creates and returns objects of classes that extend a common parent class or implement a common interface. The objects are created without exposing the instantiation logic to the client. The calling class is decoupled from knowing the exact name of the instantiated class.
- The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses.
- The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object). This pattern also defines an interface for creating objects but it lets subclasses decide which class to instantiate.
- The benefits of the Factory pattern are
  - Prefers method invocation over direct constructor calls
  - Prevents tight coupling between a class implementation and your application
  - Promotes creation of cohesive classes
  - Promotes programming to an interface
  - Promotes flexibility. Object instantiation logic can be changed without affecting the clients that use objects. It also allows the addition of new concrete classes.
- The following don't apply to the Factory pattern:
  - It doesn't eliminate the need of overloading constructors in class implementations.
  - It doesn't encourage the use of any particular access modifier. It isn't compulsory to define private members to use this pattern.
  - It won't slow your application.
  - It isn't related to how to monitor objects for change.
- The Java API uses the Factory pattern in many of its classes, including
  - `Calendar.getInstance()`
  - `Arrays.asList()`
  - `ResourceBundle.getBundle()`
  - `DriverManager.getConnectionEstablish()`, `DriverManager.getDriver()`
  - `Connection.createStatement()`
  - `Statement.executeQuery()`
  - `NumberFormat.getInstance()`, `NumberFormat.getNumberFormat()`, `NumberFormat.getCurrencyInstance()`, `NumberFormat.getIntegerInstance()`
  - `Executors.newFixedThreadPool()`, `Executors.newCachedThreadPool()`, `Executors.newSingleThreadExecutor()`

DAO pattern

- The DAO pattern encapsulates all communication with a persistent store to access and manipulate the stored data.
- The DAO pattern also manages the connection to the data store to retrieve and store the data.
- An application usually defines separate DAO classes for each type of data object that should be persisted.
- The CRUD operations form the basis of the DAO pattern.
- The DAO pattern removes the direct dependency between an application and the data persistence implementation.
- The DAO pattern is frequently used with the Factory pattern.

SAMPLE EXAM QUESTIONS

Q 3-1. What is the output? Choose the best answer.

```
interface Online {
    String course = "OCP";
    int duration = 2;
}
class EJavaGuru implements Online {
    String course = "OCA";
    public static void main(String args[]) {
        EJavaGuru ejg = new EJavaGuru();
        System.out.print(ejg.course);              // n1
        System.out.print(EJavaGuru.duration);      // n2
    }
}
```

a Compilation fails at line n1.
b Compilation fails at line n2.
c Compilations fails at both lines n1 and n2.
d Code prints "OCA2".
e Code prints "OCP2".
f Code throws a runtime exception.

Q 3-2. In the next Java version, designers are planning to create a new `switch` statement. This statement should be able to accept an object of type `SwitchArgument` and be able to call method `defaultValue()` on it. Which of the following options describe feasible (workable) options?

a Define class `SwitchArgument` and make class `java.lang.Object` extend class `SwitchArgument`.
b Define method `defaultValue` in class `java.lang.Object`.
c Define interface `SwitchArgument` with no methods. The classes that need to be used in this `switch` statement can implement interface `SwitchArgument`.

d Define interface `SwitchArgument` with method `defaultValue()`. The classes that need to be used in this `switch` statement can implement interface `SwitchArgument`.

Q 3-3. Given the following code, select the correct options:

```
class AbX {}
class Sunny extends AbX {}
interface Moon {}
class Sun implements Moon {
    Sunny AbX;
}
```

a `Sunny` IS-A `Moon`.
b `Sunny` HAS-A `AbX`.
c `Sun` HAS-A `AbX`.
d `Sun` HAS-A `Sunny`.
e `AbX` HAS-A `Moon`.
f `Sun` IS-A `Abx`.
g `Sunny` IS-A `Abx`.

Q 3-4. Given the following statements, chose the options that are correct individually:

- `ABCD` can't define instance variables.
- `XYZ` can only define public methods.
- `ABCD` can extend `XYZ`.
- `XYZ` can't implement `ABCD`.
- `LMN` can define instance variables.
- `LMN` can't extend `ABCD`.

a `ABCD` is a class.
b `ABCD` is an interface.
c `XYZ` is a class.
d `XYZ` is an interface.
e `LMN` is a class.
f `LMN` is an interface.

Q 3-5. Which of the following options are correct?

a If you add a method to your interface, you'll break all the classes that implement it.
b If you add a nonabstract method to a base abstract class, its subclasses might not always succeed to compile.

c When you work with an interface type, you decouple from its implementation.
d Code that works with a reference variable of an abstract base class works with any object of its subclasses.

Q 3-6. Given the following statements, choose the corresponding code implementation:

- `Apple` HAS-A `Ball`.
- `Ball` IS-A `Cone`.
- `Cone` HAS-A `Apple`.
- `Dot` IS-A `Ball`.

a
```
class Apple { String Ball; }
class Cone { String Apple; }
class Ball extends Cone {}
class Dot extends Ball {}
```
b
```
class Apple {Ball Ball;}
class Cone {Apple Apple;}
class Ball extends Cone {}
class Dot extends Ball {}
```
c
```
class Apple {Ball aVar;}
class Cone {Apple age;}
class Ball extends Cone {}
class Dot extends Ball {}
```
d
```
class Apple {Ball var;}
interface Cone {Apple a;}
interface Ball implements Cone {}
interface Dot implements Ball {}
```

Q 3-7. What is true about interfaces? (Choose all that apply.)

a They force an implementing class to provide its own specific functionality.
b An object of a class implementing an interface can be referred to by its own type.
c An interface can define constructors to initialize its final variables.
d An interface can define a static initializer block to initialize its variables.

Q 3-8. Select all incorrect statements.

a An abstract class may define abstract methods.
b An abstract class forces all its concrete derived classes to implement all its abstract methods.
c An abstract class does provide enough details for its objects to be created.
d An abstract class is used to group the common behaviors of a set of similar objects, but which itself may be incomplete.
e An abstract class may not be used to create a new type.
f You can create an instance of an abstract class.

Q 3-9. Assuming that the names of the classes used in the following code represent the actual objects, select the correct options.

```
class Ray {}
class Satellite {}
class Sun { Ray rays; }
class Moon extends Satellite {}
class Earth {}
class SolarSystem {
    Earth a;
    Moon b;
}
```

a `Sun` is associated with `Ray`.

b `Moon` is composed of `Satellite`.

c `SolarSystem` is composed of `Earth` and `Moon`.

d `SolarSystem` is associated with `Earth`.

e `SolarSystem` is associated with `Moon`.

f `Ray` is composed of `Sun`.

Q 3-10. Given the following code, which options correctly declare, implement, and extend these interfaces?

```
interface Coverable {}
interface Package {}
interface Ship extends Coverable, Package {}
```

a
```
class Book implements Ship {}
```

b
```
class Container implements Coverable {}
class Bottle extends Container{}
```

c
```
interface Voyage implements Ship {}
class Fan implements Voyage {}
```

d
```
interface Delivery extends Ship{}
interface Payment extends Package, Delivery{}
class Product extends Payment {}
```

Q 3-11. Which of the following code options implements the Singleton pattern correctly?

a
```
class King {
      private static King king = null;
      private King() {}
      public static King getInstance() {
          king = new King();
          return king;
      }
  }
```

b
```
class King {
    private static King king = new King();
    public static King getInstance() {
        return king;
    }
}
```

c
```
class King {
    private static King king = new King();
    private King() {}
    private static King getInstance() {
        return king;
    }
}
```

d
```
class King {
    private static King king;
    public static King getInstance() {
        if (king == null)
            king = new King();
        return king;
    }
}
```

e None of the above

Q 3-12. Given the following definition of class `King`, which option, when replacing `//INSERT CODE HERE//`, implements the Singleton pattern correctly? (Choose all that apply.)

```
class King {
    private static String name;
    private static King king = new King();
    // INSERT CODE HERE //
    public static King getInstance() {
           return king;
    }
}
```

a
```
private King() {}
```

b
```
private King() {
    name = null;
}
```

c
```
private King() {
    name = new String("King");
}
```

d
```
private King() {
    if (name != null)
        name = new String("King");
}
```

e None of the above

Q 3-13. Given the following definition of class `King`, which option, when replacing `//INSERT CODE HERE//`, implements the Singleton pattern correctly with no concurrent creation of objects of class `King`? (Choose all that apply.)

```
class Jungle {}
class King {
    private static King king = null;
    private King() {}
    //INSERT CODE HERE//
}
```

a
```
public static synchronized King getInstance() {
    if (king == null)
            king = new King();
    return king;
}
```

b
```
public static King getInstance() {
    if (king == null) {
            synchronized (Jungle.class){
                    king = new King();
            }
    }
    return king;
}
```

c
```
public static King getInstance() {
    synchronized (Jungle.class){
            if (king == null) {
                    king = new King();
            }
    }
    return king;
}
```

d
```
synchronized static public King getInstance() {
    synchronized (Jungle.class){
            if (king == null) {
                    king = new King();
            }
    }
    return king;
}
```

Q 3-14. Given the following statements, select all options that are correct individually:

- Class `Queen` implements the Singleton pattern.
- Class `King` implements the Singleton pattern.
- Class `Prince` doesn't implement the Singleton pattern.
- Class `Princess` doesn't implement the Singleton pattern.

a Only class `Queen` can create an object of class `King`.

b Either class `King` or class `Queen` can create an object of class `King`.

c Only class `King` can create its object.

d Both classes `King` and `Queen` can create objects of `Prince` and `Princess`.

e All classes (`King`, `Queen`, and `Princess`) can create objects of `Prince`.

Q 3-15. Given the definition of class `Person` as follows, which options do you think are correct implementations of a class that implements the DAO pattern for class `Person` (there are no compilation issues with this code)?

```
class Person {
    int id;
    String name;
    int age;
}
```

a
```
class PersonDAO {
    class DAO {
        Person person;
    }
}
```

b
```
class PersonDAO {
    Person findPerson(int id) { /* code */ }
    Person seekPerson(int id) { /* code */ }
}
```

c
```
class PersonDAO {
    static Person findPerson(int id) { /* code */ }
    static int create(Person p) { /* code */ }
    static int update(Person p) { /* code */ }
    static int delete(Person p) { /* code */ }
}
```

d
```
class PersonDAO {
    Person findPerson(int id) { /* code */ }
    int create(Person p) { /* code */ }
    int update(Person p) { /* code */ }
    int delete(Person p) { /* code */ }
}
```

Q 3-16. Select the correct statements:

a The DAO pattern helps decouple code that inserts data in persistence storage from code that deletes data in persistence storage.

b The DAO pattern helps encapsulate persistence data logic.

c The DAO eases migration of persistent data from one vendor to another.

d The DAO promotes low coupling and high cohesion.

Q 3-17. Given the following definition of class `Person`, which of its methods would you need to move to another class, say, `PersonDAO`, to implement the DAO pattern?

```
class Person {
    int id;
    String name;
    int age;
    int getId() {return id;}
    void setId(int id) {this.id = id;}
    String getName() {return name;}
    void setName(String name) {this.name = name;}
    int getAge() {return age;}
    void setAge(int age) {this.age = age;}
    void find() { /* code to find Person with this id in DB */ }
    void insert() { /* code to insert Person with its details in DB */ }
    void modify() { /* code to update Person with this id in DB*/ }
    void remove() { /* code to remove Person with this id in DB */ }
}
```

a Methods `getId()`, `setId()`, `find()`, `insert()`, `modify()`, `remove()`
b Methods `find()`, `insert()`, `modify()`, `remove()`
c Methods `getId()`, `setId()`, `getName()`, `setName()`, `getAge()`, `setAge()`
d Methods `getId()`, `setId()`

Q 3-18. Given

```
class Animal {}
class Herbivore extends Animal {}
class Carnivore extends Animal {}
class Cow extends Herbivore {}
class Tiger extends Carnivore {}
class Client{
    public void createAnimal(String eatingHabits) {
        Animal foo = null;
        if (eatingHabits.equals("grass"))
            foo = new Cow();
        else if (eatingHabits.equals("deer"))
            foo = new Tiger();
    }
}
```

What are the benefits of moving creation of `Animal` instances from class `Client` to a separate class, say, `Animals`?

a To enable class `Client` to use `Animal` instances without the need to know its instance creation logic
b To promote extensibility—specific `Animal` classes can be added later, which might be returned by class `Animals`
c To implement the Singleton pattern
d To implement DAO
e To enable low coupling and high cohesion

Q 3-19. Given

```
interface Animal {}
class Cat implements Animal {}
class Tiger implements Animal {}
class Factory {
    static Animal getInstance(String type) {
        if (type.equals("Tiger"))
            return new Tiger();
        else if (type.equals("Cat"))
            return new Cat();
        else
            return getAnimal();
    }
    private static Animal getAnimal() {
        return new Cat();
    }
}
```

Select code that initializes an `Animal` reference using a Factory:

a. `Animal animal = Factory.getInstance();`
b. `Animal animal = Factory.getAnimal();`
c. `Animal animal = Factory.getInstance("Animal");`
d. `Animal animal = Factory.getAnimal("Tiger");`
e. `Animal animal = new Factory().getInstance("Cat");`

Q 3-20. Which of the following use the Factory pattern? (Choose all that apply.)

a. `Object.equals();`
b. `Calendar.getInstance()`
c. `DriverManager.getDriver();`
d. `Object.wait();`
e. `NumberFormat.getDateInstance();`

ANSWERS TO SAMPLE EXAM QUESTIONS

A 3-1. d

[3.1] Write code that declares, implements, and/or extends interfaces

Explanation: Class `EJavaGuru` defines an instance variable `course`. Interface `Online` also defines a variable with the same name—`course` (which is implicitly static). Class `EJavaGuru` implements `Online`. Using `EJavaGuru`'s `instanceName.course` will refer to its instance variable. Using `Online.course` will refer to the variable `course` from `Online`. Using `EJavaGuru.course` will result in a compilation error. Code on line `n1` compiles successfully and prints `OCA`.

Because the variables defined in an interface are implicitly static and final, the variable `duration` can be accessed as `EJavaGuru.duration`. Code on line `n2` compiles successfully and prints `2`.

However, a class can't define static and instance variables with the same name. The following class won't compile:

```
class EJavaGuru {
    String course;
    static String course;
}
```

A 3-2. d

[3.2] Choose between interface inheritance and class inheritance

Explanation: Option (a) is incorrect. `java.lang.Object` is the base class of all classes in Java. Making class `java.lang.Object` extend another class can be extremely risky. Adding a method with a particular signature can break code of some other class, if it has defined a method with the same name (`defaultValue()`) but a different signature that isn't compatible, forming invalid overloaded methods.

Option (b) is incorrect. Because the requirement expects an object of `SwitchArgument`, adding just method `defaultValue()` to class `java.lang.Object` won't serve the purpose. To define a new type, we need to define `SwitchArgument` as either a class or an interface.

Option (c) is incorrect. The requirement mentions that the object of `SwitchArgument`, passed to a `switch` statement, should define method `defaultValue()`. Defining this method in interface `SwitchArgument` ensures that all classes that implement interface `SwitchArgument` define method `defaultValue()`.

Option (d) is correct. Creation of type `SwitchArgument` as an interface with method `defaultValue()` provides a convenient option for all existing classes that want to be passed as an argument to the `switch` statement. When the classes implement the interface `SwitchArgument`, they'll be responsible for implementing method `defaultValue()`.

A 3-3. d, g

[3.3] Apply cohesion, low-coupling, IS-A, and HAS-A principles

Explanation Option (a) is incorrect. Classes `Sunny` and `Moon` are unrelated.

Option (b) is incorrect. Class `Sunny` extends class `AbX`; it doesn't define a variable of type `AbX`. The correct relationship here would be `Sunny` IS-A `AbX`.

Option (c) is incorrect. Class `Sun` defines a variable of type `Sunny`. So the correct relation would be `Sun` HAS-A `Sunny`. The IS-A and HAS-A relationships don't reflect the names of the variables.

Option (d) is correct. Class `Sun` defines a variable of type `Sunny`, so the relationship `Sun` HAS-A `Sunny` is correct.

In option (e), class `AbX` doesn't define any variable of type `Moon`, so this relationship is incorrect.

In option (f), class `Sun` doesn't extend class `AbX`, so this relationship is incorrect.

In option (g), class `Sunny` extends class `AbX`, so this relationship is correct.

A 3-4. b, d, e

[2.1] Identify when and how to apply abstract classes
[2.2] Construct abstract Java classes and subclasses
[3.1] Write code that declares, implements, and/or extends interfaces

Explanation: Option (a) is incorrect. As specified, `ABCD` can't define an instance variable, but a class can define instance variables.

Option (b) is correct. All the variables defined in an interface are implicitly public, final, and static. The static variables can't exist as instance variables.

Option (c) is incorrect. `XYZ` can't exist as a class. As specified, `XYZ` can define only public methods, whereas a class can define nonpublic methods.

Option (d) is correct. `XYZ` can exist as an interface because all the methods in an interface are implicitly public. All methods in a class aren't implicitly public.

Option (e) is correct, and (f) is incorrect. `LMN` is a class because it can define instance methods and can't extend `ABCD`, an interface.

A 3-5. a, b, c, d

[2.1] Identify when and how to apply abstract classes
[2.2] Construct abstract Java classes and subclasses
[3.1] Write code that declares, implements, and/or extends interfaces

Explanation: Option (a) is correct. If you add a method to your interface, all the classes that implement the interface will fall short on the definition of the newly added method and will no longer compile.

Option (b) is correct. If you add a nonabstract method to your base class, you *can* break its subclasses. If a subclass method has the same name as the newly added method in the base class, which doesn't qualify as a valid overloaded or overriding method, the subclass won't compile.

Option (c) is correct. When you work with an interface type, you're free to work with any object that implements the interface.

Option (d) is correct. Objects of all subclasses can be assigned to a reference variable of its abstract or nonabstract base class. So code that works with the abstract base class will work with objects of any of its subclasses.

A 3-6. b, c

[3.3] Apply cohesion, low-coupling, IS-A, and HAS-A principles

Explanation: An IS-A or a HAS-A relationship is defined between the types of the variables, and not their names.

Option (a) is incorrect. Class `Apple` HAS-A `String` and not `Ball`.

Option (b) is correct. Class `Apple` defines a variable `Ball` of type `Ball`. So `Apple` HAS-A `Ball`. It's acceptable to define a variable with the name of its class. Class `Cone` defines a variable `Apple` of type `Apple`. So it satisfies the relationship `Cone` HAS-A `Apple`. Class `Ball` extends class `Cone`, so it satisfies `Ball` IS-A `Cone`. Class `Dot` extends class `Ball`. So it satisfies `Dot` IS-A `Ball`.

Option (c) is also correct. Class `Apple` defines a variable `aVar` of type `Ball`. So `Apple` HAS-A `Ball`. Class `Cone` defines a variable `age` of type `Apple`. So it satisfies the relationship `Cone` HAS-A `Apple`. Class `Ball` extends class `Cone`, so it satisfies `Ball` IS-A `Cone`. Class `Dot` extends class `Ball`. So it satisfies `Dot` IS-A `Ball`.

Option (d) is incorrect. An interface can't implement another interface. It can only extend it.

A 3-7. a, b

[3.1] Write code that declares, implements, and/or extends interfaces

Explanation: Options (c) and (d) are incorrect. An interface can neither define a constructor nor a static initializer block.

A 3-8. c, e, f

[2.1] Identify when and how to apply abstract classes

Explanation: Option (c) is an incorrect statement, because objects of an abstract class can't be created, even if the class doesn't define any abstract method.

Option (e) is an incorrect statement. An abstract class defines a new type. This type can be used to define variables in multiple scopes (instance variables, static variables, method parameters, and local variables).

Option (f) is an incorrect statement, because you can't create an object of an abstract class.

A 3-9. a, c, d, e

[3.4] Apply object composition principles (including HAS-A relationships)

Explanation: Option (a) is correct. `Sun` gives out rays, so `Sun` is associated with `Ray`. Option (b) is incorrect. `Moon` is a type of `Satellite`. Composition is a whole-part relationship; if the enclosing object goes out of scope, the part also goes out of scope.

Option (c) is correct. If `SolarSystem` goes out of scope, all the objects that it's composed of, including `Earth` and `Moon`, will go out of scope.

Options (d) and (e) are correct. Composition is a special type of association, and objects in this relationship are associated with each other.

Option (f) is incorrect. `Ray` isn't composed of `Sun`. It's the other way around: `Sun` is composed of `Ray`. If `Sun` goes out of scope, `Ray` also goes out of scope (is no longer visible).

A 3-10. a, b

[3.1] Write code that declares, implements, and/or extends interfaces

Explanation: Option (c) is incorrect, because an interface (`Voyage`) can't implement another interface (`Ship`).

Option (d) is incorrect, because a class (`Product`) can't extend an interface (`Payment`).

A 3-11. e

[3.5] Design a class using the Singleton design pattern

Explanation: Option (a) is incorrect. It creates a new object of class `King`, whenever method `getInstance()` is called. On the contrary, the Singleton pattern creates only one instance of a class.

Option (b) is incorrect. A Singleton should define a `private` constructor so that no other class can create its objects. The class defined in this option doesn't define any constructor. In the absence of a constructor, the Java compiler creates a default constructor for a class with the access modifier as that of the class itself. Because the class in this option is defined with default or package access, a constructor with default access will be created for it by the Java compiler. Because other classes can use its constructor to create new objects of this class, it doesn't qualify as a Singleton.

Option (c) is incorrect. There is no way to access an object of class `King` outside the class itself. Variable `king` is a static private variable, so it can't be accessed directly. The constructor of the class is marked `private`, so it can't be used to create objects of this class. Method `getInstance()` is also private, so no other class can call it.

Option (d) is incorrect. Though the variable `king` is private, and method `getInstance` creates and returns an object of class `King`, the catch here is that this class doesn't define a constructor. As mentioned in the explanation of option (b), in the absence of a constructor, the Java compiler creates a default constructor. Because other classes can use its constructor to create new objects of this class, it doesn't qualify as a Singleton.

A 3-12. a, b, c, d

[3.5] Design a class using the Singleton design pattern

Explanation: All the options are trying to confuse you with the correct implementation of method `getInstance()` of a class that uses the Singleton pattern with its constructor. A class that implements the Singleton pattern should have a private constructor, so no other class can create its objects. The implementation of the constructor isn't detailed by the Singleton pattern. The class may choose to include or exclude whatever it feels is good for it.

A 3-13. a, c, d

[3.5] Design a class using the Singleton design pattern

Explanation: In option (a), the complete method `getInstance()` is synchronized, which ensures that only one `Thread` executes this method and creates an instance of class `King` (assigning it to the static variable `king`), if it's `null`.

Option (b) is incorrect. Method `getInstance()` synchronizes only the code `king = new King();`. So multiple methods can still execute method `getInstance()` concurrently and query whether the variable `king` is `null`. If, say, two threads find it `null`, they both will execute the following code (though not at the same time):

```
king = new King();
```

When the second `Thread` executes the preceding code, it creates and assigns another object of class `King` to variable `king`. This method fails to prevent multiple creations of objects of class `King`.

Option (c) is correct. Method `getInstance()` synchronizes the part of the method that creates an object of class `King`. When the control is within the `synchronized` block, the code checks again to confirm that variable `king` is still `null`. If true, it creates an object of class `King` and assigns it to the variable `king`.

Option (d) is correct. It defines the same code as option (c), but with a difference: this option applies the `synchronized` keyword to the method also. Though synchronizing the code block and the complete method isn't required, it isn't incorrect to do so. Because this method prevents creation of multiple objects of class `King`, it qualifies as a correct implementation of method `getInstance()`.

A 3-14. c, d, e

[3.5] Design a class using the Singleton design pattern

Explanation: Options (a) and (b) are incorrect and (c) is correct. Only a class that implements the Singleton pattern can create its object, because its constructor is marked `private`. No other class can. Only class `King` can create its own object by calling its constructor from its other method.

Options (d) and (e) are correct. If a class doesn't implement the Singleton pattern, we can assume that creation of its multiple objects is allowed. So another class can also create its objects.

A 3-15. c, d

[3.6] Write code to implement the DAO pattern

Explanation: Options (a) and (b) are incorrect. A class that implements the DAO pattern should define methods for CRUD operations (create, retrieve, update, and delete). Options (a) and (b) don't define all these methods.

Options (c) and (d) are correct. Both these options define methods for CRUD operations. You can implement these methods as static or nonstatic.

A 3-16. b, c, d

[3.6] Write code to implement the DAO pattern

Explanation: Option (a) is incorrect. The DAO pattern helps separate and decouple application logic from persistence storage logic. It isn't used to decouple different data manipulation operations.

A 3-17. b

[3.6] Write code to implement the DAO pattern

Explanation: To implement the DAO pattern, you should move the methods that interact with the persistent data storage to a separate class. In class `Person`, the getter and setter methods are for assigning and retrieving object fields. They don't work with data in persistence storage. The rest of the methods (`find()`, `insert()`, `modify()`, and `remove()`) work with persistent data and should be moved to another class to implement the DAO pattern. The DAO pattern doesn't specify any rules for conventions on naming these methods and the type that they return.

A 3-18. a, b, e

[3.7] Design and create objects using a Factory pattern

Explanation: Here's one of the ways you can move `Animal` instance creation logic to class `Animals`:

```
class Amimals{
    public static Animal createAnimal(String eatingHabits) {
        Animal foo = null;
        if (eatingHabits.equals("grass"))
            foo = new Cow();
```

```
        else if (eatingHabits.equals("deer"))
            foo = new Tiger();
        return foo;
    }
}
```

Option (a) is correct. Moving method `createAnimal()` to a separate class frees class `Client` from knowing the logic of creating `Animal` instances. It can call `Animals.createAnimal()`, passing it a `String` value to get an appropriate `Animal` instance.

Option (b) is correct. Method `createAnimal()` in class `Animals` can be modified to include instantiation of other specific `Animal` instances without modifying its API.

Options (c) and (d) are incorrect. The stated modification is neither related to data persistence nor to creating just one instance of a class.

Option (e) is correct. With the modification, class `Client` doesn't need to know about the specific implementations of class `Animal`. Class `Client` can concentrate on using `Animal` instances rather than knowing how to create them.

A 3-19. c, e

[3.7] Design and create objects using a Factory pattern

Explanation: Class `Factory` doesn't expose the object creation logic of `Animal` objects and uses the Factory pattern to create and return its instances.

Option (a) won't compile. Though you might dismiss it as a trivial or tricky option, note that it's easy to find similar options on the exam.

Options (b) and (d) won't compile because `getAnimal()` is a private method and it doesn't define the method parameters.

Option (e) is correct. A static method can be accessed using both the class name and an instance.

A 3-20. b, c, e

[3.7] Design and create objects using a Factory pattern

Explanation: Methods `equals()` and `wait()` in class `Object` don't use the Factory pattern.