Chapter 2

Object Orientation

Business Training

1

Content

1Z0-809 Chapter 2

- Implement Encapsulation
- The Singelton Pattern
- Immutable classes
- Use Static members

2

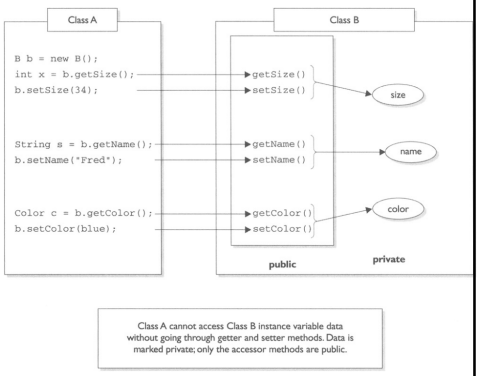
Implement encapsulation

3

1Z0-809 Chapter 2

Encapsulation

- **If you want maintainability, flexibility, and extensibility (and of course, you do), your design must include encapsulation. How do you do that?**
  - Keep instance variables protected (with an access modifier, often private).
  - Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable.
  - For the methods, use the JavaBeans naming convention of set<someProperty>and get<someProperty>.

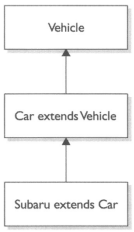


Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

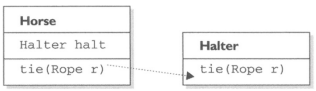
4

Inheritance, Is-A, Has-A

- **Inheritance is everywhere in Java. For the exam you'll need to know that you can create inheritance relationships in Java by extending a class. It's also important to understand that the two most common reasons to use inheritance are**
  - To promote code reuse(create a fairly generic version of a class with the intention of creating more specialized subclasses that inherit from it)
  - To use polymorphism (allow your classes to be accessed polymorphically—a capability provided by interfaces as well)



- **The IS-A Relationship**
  - In OO, the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying, "this thing is a type of that thing."
- **The Has-A Relationship**
  - HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B.



Horse class has a Halter, because Horse declares an instance variable of type Halter. When code invokes `tie()` on a Horse instance, the Horse invokes `tie()` on the Horse object's Halter instance variable.

5

Inheritence in Java 8

- **As the following table shows, it's now possible to inherit concrete methods from interfaces. This is a big change.**
- **you'll notice that as of Java 8 interfaces can contain two types of concrete methods, static and default.**

Elements of Types	Classes	Interfaces
Instance variables	Yes	Not applicable
Static variables	Yes	Only constants
Abstract methods	Yes	Yes
Instance methods	Yes	Java 8, default methods
Static methods	Yes	Java 8, inherited no, accessible yes
Constructors	No	Not applicable
Initialization blocks	No	Not applicable

- **For the exam, you'll need to know that you can create inheritance relationships in Java by *extending* a class or by implementing an interface. It's also important to understand that the two most common reasons to use inheritance are :**
  1. To promote code reuse
  2. To use polymorphism

6

Polymorphism

- **Remember, any Java object that can pass more than one IS-A test can be considered polymorphic.**
  - Other than objects of type object, all Java objects are polymorphic in that they pass the IS-A test for their own type and for class Object.
- **Polymorphic method invocations apply only to instance methods.**
  - You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual object (rather than the reference type) are instance methods. Not static methods. Not variables. Only overridden instance methods are dynamically invoked based on the real object's type.

Overriding / Overloading

- **Invoking a Superclass Version of an Overridden Method**
  - It's easy to do in code using the keyword super.
- **Examples of Legal and Illegal Method Overrides**

```
public class Animal {
  public void eat () { }
}
```

Illegal Override Code	Problem with the Code
`private void eat () { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not defined by superclass version
`public void eat (String food) { }`	A legal overload, not an override, because the argument list changed
`public string eat() { }`	Not an override because of the return type, not an overload either because there's no change in the argument list

- **Overloaded methods ("Le surcharge" en français)**
  - Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).
  - The rules are simple:
    - Overloaded methods MUST change the argument list.
    - Overloaded methods CAN change the return type.
    - Overloaded methods CAN change the access modifier.
    - Overloaded methods CAN declare new or broader checked exceptions.

Overriding / Overloading

```
public class Animal {
   public void eat () {
      System.out.println("Generic Animal Eating Generically"};
   }
}
public class Horse extends Animal {
   public void eat() {
       System.out.println("Horse eating hay ");
   }
   public void eat(String s) {
      System, out .println ("Horse eating " + s) ;
   }
}
```

Method Invocation Code	Result
`Animal a = new Animal(); a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse(); h.eat();`	`Horse eating hay`
`Animal ah = new Horse(); ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (`Horse`), not the reference type (`Animal`), is used to determine which `eat()` is called.
`Horse he = new Horse(); he.eat("Apples");`	`Horse eating Apples` The overloaded `eat(String s)` method is invoked.
`Animal a2 = new Animal(); a2.eat("treats");`	Compiler error! Compiler sees that the `Animal` class doesn't have an `eat()` method that takes a `String`.
`Animal ah2 = new Horse(); ah2.eat("Carrots");`	Compiler error! Compiler still looks only at the reference and sees that `Animal` doesn't have an `eat()` method that takes a `String`. Compiler doesn't care that the actual object might be a `Horse` at runtime.

Extending, Implementing

```
class Foo { }                                  // OK
class Bar implements Foo  { }                  // No! Can't implement a class
interface Baz { }                              // OK
interface Fi { }                               // OK
interface Fee implements Baz { }               // No! an interface can't
                                               // implement an interface
interface Zee implements Foo { }               // No! an interface can't
                                               // implement a class
interface Zoo extends Foo { }                  // No! an interface can't
                                               // extend a class
interface Boo extends Fi { }                   // OK. An interface can extend
                                               // an interface
class Toon extends Foo, Button { }             // No! a class can't extend
                                               // multiple classes
class Zoom implements Fi, Baz { }              // OK. A class can implement
                                               // multiple interfaces
interface Vroom extends Fi, Baz { }            // OK. An interface can extend
                                               // multiple interfaces
class Yow extends Foo implements Fi { }        // OK. A class can do both
                                               // (extends must be 1st)
class Yow extends Foo implements Fi, Baz { } // OK. A class can do all three
                                               // (extends must be 1st)
```

Overriding / Overloading

- **Overridden Methods**
  - Any time you have a class that inherits a method from a superclass, you have the opportunity to override the method (unless, as you learned earlier, the method is marked final).
    - **The key benefit of overriding is the ability to define behavior that's specific to a particular subclass type.**
  - The rules for overriding a method are as follows:
    - **The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.**
    - **The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. (covariant returns.)**
    - **The access level CAN be less restrictive than that of the overridden method.**
    - **The access level can't be more restrictive than the overridden method's.**
    - **Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass).**
    - **The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.**
    - **The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method.**
    - **The overriding method can throw narrower or fewer exceptions. Just because an overridden method "takes risks" doesn't mean that the overriding subclass' exception takes the same risks. Bottom line: an overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.**
    - **You cannot override a method marked final.**
    - **You cannot override a method marked static.**
    - **If a method can't be inherited, you cannot override it. Remember that overriding implies that you're reimplementing a method you inherited!**

11

Reference Variable Casting

- **Downcasting vs. Upcasting**
  - Down casting : casting a class to a sub-class
  - Up Casting : casting a sub-class to a super-class
- **Unlike downcasting, upcasting works implicitly (i.e. you don't have to type in the cast)**
  - This because when you upcast you're implicitly restricting the number of methods you can invoke, as opposed to downcasting, which implies that later on, you might want to invoke a more specific method.

12

Is the Cast necessary ?

- **Given**

```
class Animal {
  void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
  void makeNoise() {System.out.println("bark"); }
  void playDead() { System.out.println("roll over"); }
}

class CastTest2 {
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal() };
    for(Animal animal : a) {
      animal.makeNoise();
      if(animal instanceof Dog) {
        animal.playDead();        // try to do a Dog behavior?
      }
    }
  }
}
```

  - When we try to compile this code, the compiler says something like this:
    - **cannot find symbol**
  - The compiler is saying, “Hey, class Animal doesn’t have **playDead()** method.” Let’s modify the if code block:

```
if(animal instanceof Dog) {
  Dog d = (Dog) animal;      // casting the ref. var.
  d.playDead();
}
```

    - Now the compiler is happy !

Legal Return Types

- **Return Types on Overloaded Methods**
  - Remember that method overloading is not much more than name reuse.
  - What you can't do is change **only** the return type. To overload a method, remember, you must change the argument list.
- **Overriding and Return Types, and Covariant Returns**
  - When a subclass wants to change the method implementation of an inherited method (an override), the subclass must define a method that matches the inherited version exactly.
    - **Or, as of Java 5, you're allowed to change the return type in the overriding method as long as the new return type is a subtype of the declared return type of the overridden (superclass) method.**

Legal Return Types

- **Returning a Value**
  - You have to remember only six rules for returning a value:
    1. You can return null in a method with an object reference return type.

       ```
       public Button doStuff() { return null;}
       ```

    2. An array is a perfectly legal return type.

       ```
       public String[] go () { return new String[] {"Fred", "Barney", "wilma"};}
       ```

    3. In a method with a primitive return type, you can return any value or variable that can be implicitly converted to the declared return type.

       ```
       public int foo () {  char c = 'c'; return c; } // char is compatible with int
       ```

    4. In a method with a primitive return type, you can return any value or variable that can be explicitly cast to the declared return type.

       ```
       public int foo () {  float f = 32.5f;  return (int) f;}
       ```

    5. You must not return anything from a method with a void return type.

       ```
       public void bar() {  return "this is it"; } // Not legal!!
       ```

    6. In a method with an object reference return type, you can return any object type that can be implicitly cast to the declared return type.

       ```
       public Animal getAnimal() {  return new Horse(); }
       ```


15

Constructor Basics

- **Constructor Basics**
  - Every class, including abstract classes, MUST have a constructor. Burn that into your brain. But just because a class must have one, doesn't mean the programmer has to type it.
    - So it's very common (and desirable) for a class to have a no-arg constructor, regardless of how many other overloaded constructors are in the class
- **Constructor Chaining**
  - what really happens when you say new Horse () ?(Assume Horse extends Animal and Animal extends Object.)
    - Horse constructor is invoked. Every constructor invokes the constructor of its superclass with an (implicit) call to super(), unless the constructor invokes an overloaded constructor of the same class.
    - Animal constructor is invoked (Animal is the superclass of Horse).
    - Object constructor is invoked. At this point we're on the top of the stack.
    - Object instance variables are given their explicit values.
    - Object constructor completes.
    - Animal instance variables are given their explicit values (if any).
    - Animal constructor completes.
    - Horse instance variables are given their explicit values (if any).
    - Horse constructor completes.

4.	`Object()`
3.	`Animal()` **calls** `super()`
2.	`Horse()` **calls** `super()`
1.	`main()` **calls** `new Horse()`

16

Compiler-Generated Constructor Code

Class Code (WhatYouType)	Compiler Generated Constructor Code (in Bold)

class Foo { }	class Foo {
	Foo() {
	super ( ) ;
	}
	}

class Foo {
Foo() { }
}
	class Foo {
	Foo() {
	super() ;
	}
	}

public class Foo { }	class Foo {
	public Foo() {
	super();
	}
	}

class Foo {
Foo(String s) { }
}
	class Foo {
	Foo (String s) {
	super();
	}
	}

class Foo {
Foo(String s) {
super();
}
}
	Nothing, compiler doesn't need to insert anything.

class Foo {
void Foo() { }
}
	class Foo {
	void Foo() { }
	Foo() {
	super();
	}
	}
	(void Foo() *is a method, not a constructor.*)

Overloaded Constructors

- **Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:**

```
class Foo {
  Foo() { }
  Foo (String s) { }
}
```

- **The preceding Foo class has two overloaded constructors, one that takes a string, and one with no arguments.**
  - Because there's no code in the no-arg version, it's actually identical to the default constructor the compiler supplies, but remember—since there's already a constructor in this class (the one that takes a string), the compiler won't supply a default constructor.
  - If you want a no-arg constructor to overload the with-args version you already have, you're going to have to type it yourself, just as in the Foo example.

Overriding Methods in Object Class

- **Overriding toString() method**
  - You are expected to override this method to return the desired textual representation of your class
    - Reminder: While overriding, you need to be careful about the access levels, the name of the method, and its signature
- **Overriding equals() et hashCode()**
  - The equals() method is used to determine logical equivalence between object instances.
    Here is the signature of the equals() : public boolean equals(Object obj)
  - Ensure that the hashCode() method returns the same hash value for two objects if the equals() method returns true for them.
    - If you're using an object in containers like HashSet or HashMap, make sure you override the hashCode() and equals() methods correctly.

19

The Singleton Pattern

20

Design patterns

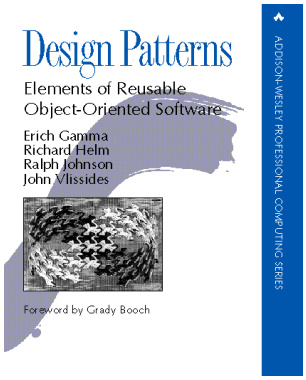
- **What is a Design Pattern ?**
  - A design pattern is a general repeatable solution to a commonly occurring problem in software design.
  - Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.
- **Why do we need design patterns ?**
  - **Make your life easier by not reinventing the wheel**
    - Patterns provide ready-made design templates that can be applied to your own designs to solve common problems.
    - Avoid falling into mistakes and known pitfalls
  - **Use the power of a shared vocabulary**
    - Simpler to say, « We put a Façade here".
  - **Provide a more abstract view of the software architecture**
    - Free designers / developers of implementation details, especially in the early stages of the development cycle

GOF Design patterns

- **23 patterns dans leur ouvrage de référence**

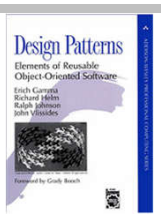
  Gamma E, Helm R, Johnson R and Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

- **Trois catégories:**
  - Patterns Créationnels
  - Patterns Structurels
  - Patterns Comportementaux
- **Souvent critiqués par les puristes pour l'aspect "basique" du catalogue**



Organisation du catalogue des Design patterns GOF

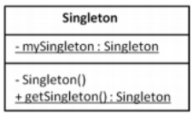
- **Créationnel** : processus de création des objets
- **Structurel** : composition des classes ou des objets
- **Comportemental** : comment les classes et les objets interagissent et distribuent les responsabilités



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Singleton Design Pattern

- **A signleton is a class that can has just only one instance at runtime**
  - Examples : java.lang.Runtime, java.awt.Desktop, …



- **In Java, broadly speaking, there are two ways to implement a singleton class:**
  1. With a private constructor and a static factory method
  2. As an enum

```
class Singleton {
  private static final Singleton instance = new Singleton();
  private Singleton() { }
  public static Singleton getInstance() { return instance; }
}
```

- **Implementing a Singleton with a private constructor**
  - A variation of this implementation is to use **a static inner class.**
  - The advantage of this is that the instance **won't be created until the inner class is referenced for the first time.**

```
class Singleton {
    private Singleton() { }
    private static class Holder {
        private static final Singleton instance
                                        = new Singleton();
    }
    public static Singleton getInstance() {
        return Holder.instance;
    }
}
```

Thread-Safe Singleton

- **In the case of multiple threads, trying to get a singleton object may result in creation of multiple objects, which of course defeats the purpose of implementing a singleton.**
  - The **synchronized** keyword guaranties that getInstance() is accessed bay one thread at a time.
  - An improvement is to lock only the portion of the code that creates the instance. For this to work properly, we have to double check if the instance is null

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

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


Singleton using « Initialization-on-demand holder » idiom

- **« Initialization-on-demand holder» is based on inner classes**
  - The approach here is to create a 'holder' as an inner class, which will statically initialize the singleton.
  - Class initialization is 'on-demand' is performed the first time the class is used.

```
public class Logger {
        private Logger() {
                // private constructor
        }
        public static class LoggerHolder {
                public static Logger logger = new Logger();
        }
        public static Logger getInstance() {
                return LoggerHolder.logger;
        }
        public void log(String s) {
                // log implementation
                System.err.println(s);
        }
}
```

  - Calling getInstance() references the inner class, triggering the JVM to load & initialize it. This is thread-safe, since classloading uses locks.
    - For subsequent calls, the JVM resolves our already-loaded inner class & returns the existing singleton. Thus — a cache.
    - And thanks to the magic of JVM optimizations, a very very efficient one.

Immutale classes

27

Defining immutable classes

- **Immutable objects cannot change after they are created.**
  - This means that they cannot have setter methods or public variables, so the only way to set its properties is through the constructor.
  - There are some immutable classes in the Java JDK, for example:
    - java.lang.String
    - Wrappers classes (like Integer)
- **Keep the following aspects in mind for creating your own immutable objects:**
  - Has a class declared final to prevent inheritance.
  - Declares all its attributes private and does not define setter methods.
  - Make the fields final and initialize them in the constructor.
    - For primitive types, the field values are final, there is no possibility of changing the state after it got initialized. For reference types, you cannot change the reference.
  - Protects access to any mutable state. For example, if it has a List member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change).

28

Defining immutable classes

- **Example of an immutable class**

```
// ImmutableCircle is an immutable class - the state of its objects
// cannot be modified once the object is created

public final class ImmutableCircle {
        private final Point center;
        private final int radius;
        public ImmutableCircle(int x, int y, int r) {
                center = new Point(x, y);
                radius = r;
        }
        public String toString() {
                return "center: " + center + " and radius = " + radius;
        }
        public int getRadius() {
                return radius;
        }
        public Point getCenter() {
                // return a copy of the object to avoid
                // the value of center changed from code outside the class
                return new Point(center.getX(), center.getY());
        }
        public static void main(String []s) {
                System.out.println(new ImmutableCircle(10, 10, 20));
        }
        // other members are elided ...
}
```


29

Use Static members

30

Static Variables and Methods

- **The following code declares and uses a static counter variable:**

```
class Frog {
    static int frogCount = 0;  // Declare and initialize static variable
    public Frog() {
        frogCount += 1;  // Modify the value in the constructor
    }
    public static void main (String [] args) {
    new Frog();  new Frog(); new Frog();
    System.out.println ("Frog count is now " + frogCount);
    }
}
```

  - When this code executes, three Frog instances are created in main(), and the result is
    Frog count is now 3

- **Exam Watch**
  - The following code is an example of illegal access of a nonstatic variable from a static method:

```
class Foo {
   int x = 3;
   public static void main (String [] args) {
     System.out.println("x is " + x);
} }
```

Understand that this code will never compile, because you can't access a nonstatic (instance) variable from a static method. Just think of the compiler saying,

31

Accessing Static Methods and Variables

```
class Foo
int size = 42;
static void doMore( ){
  int x = size;
}
```

static method cannot access an instance (non-static) variable

```
class Bar
void go ( );
static void doMore( ){
    go( );
}
```

static method cannot access a non-static method

```
class Baz
static int count;
static void woo( ){ }
static void doMore( ){
    woo( );
    int x = count;
}
```

static method *can* access a static method or variable

32

Initialization Blocks

- Initialization blocks run when the class is first loaded (a static initialization block) or when an instance is created (an instance initialization block). Let's look at an example:

```
class SmallInit {
  static int x;
  int y;
  static { x = 7 ; }      // static init block
  { y = 8; }              // instance init block
}
```

- Remember these rules:
  - Init blocks execute in the order they appear.
  - Static init blocks run once, when the class is first loaded.
  - Instance init blocks run every time a class instance is created.
  - Instance init blocks run after the constructor's call to super().

Initialization Blocks

- Can you determine the output of the following program?

```
class Init {
    Init(int x) { System.out.println("1-arg const"); }
    Init() {
      super();
      System.out.println("no-arg const"); }
    static { System.out.println("1st static init"); }
    { System.out.println("1st instance init"); }
    { System.out.println("2nd instance init"); }
    static { System.out.println("2nd static init"); }

    public static void main(String [] args) {
          new Init();
          new Init(7);
    }
}
```

- The following output should make sense:

```
1st static init
2nd static init
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const
```

Statics : Points to remember

- **You cannot override a static method provided in a base class.**
  - Why? Based on the instance type, the method call is resolved with runtime polymorphism. Since static methods are associated with a class (and not with an instance), you cannot override static methods, and runtime polymorphism is not possible with static methods.
- **A static method cannot use the this keyword in its body.**
  - Why? Remember that static methods are associated with a class and not an instance. Only instance methods have an implicit reference associated with them; hence class methods do not have a this reference associated with them.
- **A static method cannot use the super keyword in its body.**
  - Why? You use the super keyword for invoking the base class method from the overriding method in the derived class. Since you cannot override static methods, you cannot use the super keyword in its body.
- **Since static methods cannot access instance variables (nonstatic variables), they are most suited for utility functions.**
  - That's why there are many utility methods in Java. For example, all methods in the java.lang.Math library are static.