



# C2 Object Orientation

## Java 8 Associate

1Z0-808

[Link](#)

<https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387>

<https://docs.oracle.com/javase/specs/jls/se8/html>

<https://docs.oracle.com/javase/tutorial>

<http://www.java2s.com>

<https://enthuware.com>

<https://github.com/carlosherrera/1Z0-808>

JUAN CARLOS HERRERA HERNANDEZ

[carlos.herrera@upa.edu.mx](mailto:carlos.herrera@upa.edu.mx)

## Contenido

1. Encapsulation .....	3
2. Inheritance, Is-A, Has-A.....	4
3. Polymorphism .....	7
4. Overriding/Overloading .....	8
5. Casting.....	11
6. Implementing an Interface.....	12
7. Return Types .....	15
7.1. Return Types on Overloaded Methods .....	15
7.2. Overriding and Return Types and Covariant Returns.....	15
8. Constructors and Instantiation.....	16
9. Initialization Blocks.....	19
10. Statics .....	20
11. Coupling and Cohesion (scjp).....	22
CERTIFICATION SUMMARY .....	23

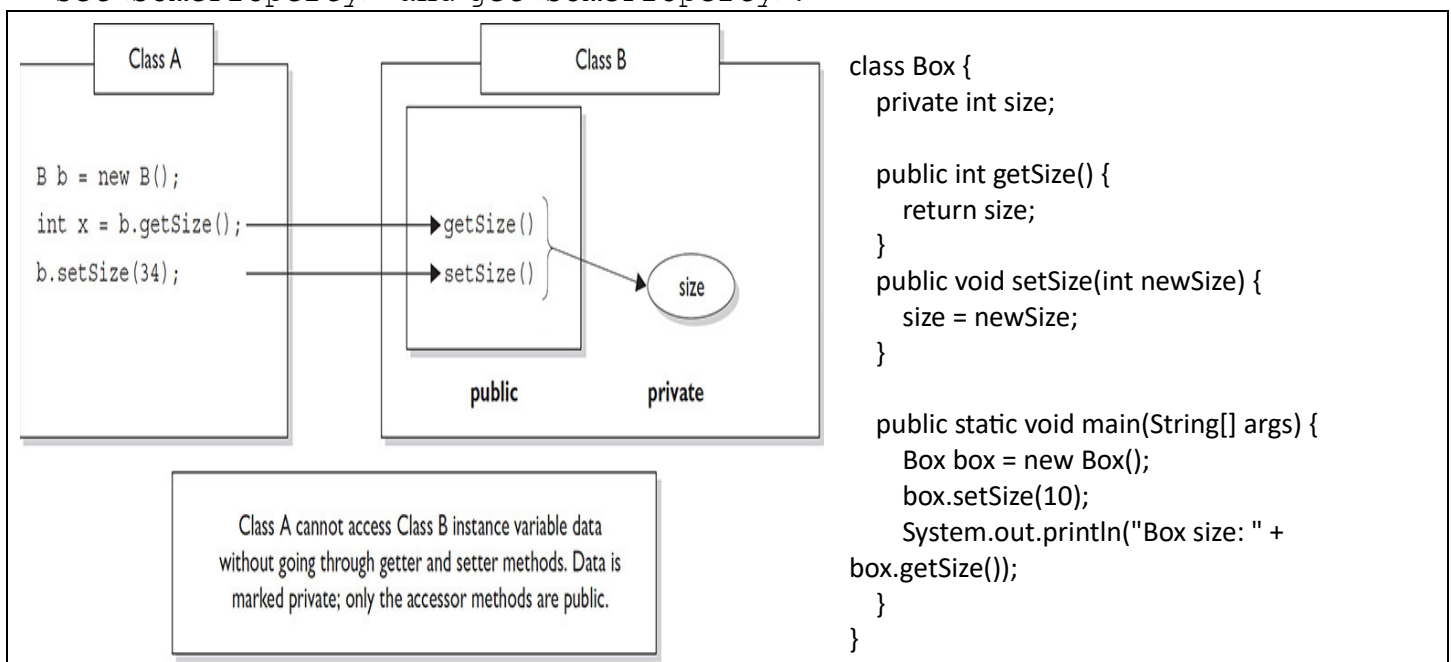
## 1. Encapsulation

6.1 Create methods with arguments and return values; including overloaded methods.

6.5 Apply encapsulation principles to a class.

If you want maintainability, flexibility, and extensibility, your design must include encapsulation. How do you do that?

- Keep instance variables hidden (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. These so called accessor methods allow users of your class to **set** a variable's value or **get** a variable's value.
- For these accessor methods, use the most common naming convention of `set<SomeProperty>` and `get<SomeProperty>`.



- Encapsulation helps hide implementation behind an interface (or API).
- Encapsulated code has two features:
  - Instance variables are kept protected (usually with the `private` modifier).
  - Getter and setter methods provide access to instance variables.
- IS-A refers to inheritance or implementation.
- IS-A is expressed with the keyword `extends` or `implements`.
- IS-A, “inherits from,” and “is a subtype of” are all equivalent expressions.
- HAS-A means an instance of one class “has a” reference to an instance of another class or another instance of the same class.
- \*HAS-A is NOT on the exam, but it’s good to know.

Example: One employee has a bank card.

>>

## 2. Inheritance, Is-A, Has-A

```
System.out.println("Paris" instanceof String);
if (t1 instanceof Object) { System.out.println("t1's an Object");}
```

Left operand of "instanceof" MUST be an object and not a primitive.  
 Left operand of "instanceof" MUST be a reference variable and not a primitive.  
 Right operand of "instanceof" MUST be a reference TYPE name, i.e., a class, an interface, or an enum name.

- Inheritance allows a type to be a subtype of a supertype and thereby inherit `public` and `protected` variables and methods of the supertype.
- Inheritance is a key concept that underlies IS-A, polymorphism, overriding (sobrescritura), overloading (sobrecarga), and casting.
- All classes (except class `Object`) are subclasses of type `Object`, and therefore they inherit `Object`'s methods.

**TABLE 2-1** Inheritable Elements of Classes and Interfaces

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

You'll need to know that you can create inheritance relationships in Java by *extending* a class or by implementing an interface.

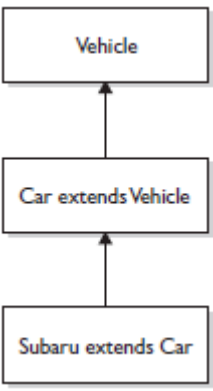
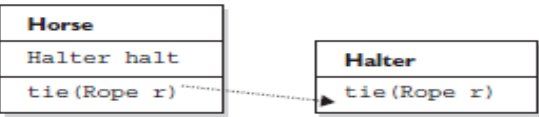
It's also important to understand that the two most common reasons to use inheritance are:

- To promote code reuse
- To use polymorphism

### IS-A and HAS-A Relationships

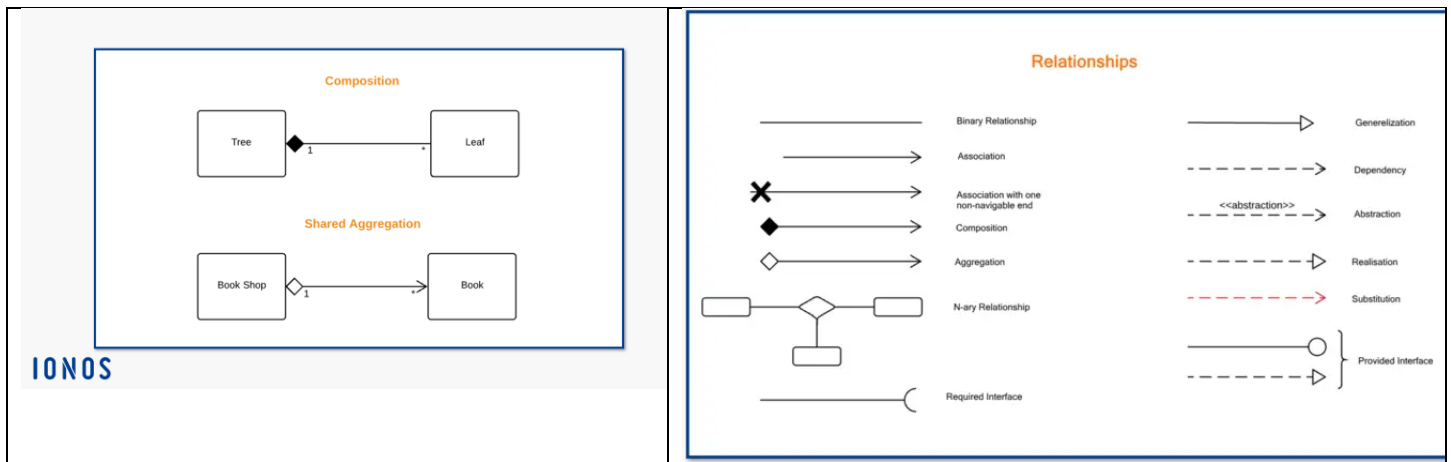
- IS-A is based on inheritance (or interface implementation).
- IS-A is expressed with the keyword `extends` or `implements`.

the IS-A relationship in Java through the keywords `extends` (for *class* inheritance) and `implements` (for *interface* implementation).

 <pre> classDiagram     Vehicle &lt; -- Car     Car &lt; -- Subaru </pre>	<pre> public class Vehicle { ... } public class Car extends Vehicle { ... } public class Subaru extends Car { ... } </pre> <p>In OO terms, you can say the following:  Vehicle is the superclass of Car.  Car is the subclass of Vehicle.  Car is the superclass of Subaru.  Subaru is the subclass of Vehicle.  Car inherits from Vehicle.  Subaru inherits from both Vehicle and Car.  Subaru is derived from Car.  Car is derived from Vehicle.  Subaru is derived from Vehicle.  Subaru is a subtype of both Vehicle and Car.</p> <p>Returning to our IS-A relationship, the following statements are true:</p> <p>"Car extends Vehicle" means "Car IS-A Vehicle."  "Subaru extends Car" means "Subaru IS-A Car."</p>
 <pre> classDiagram     class Horse {         Halter halt         tie(Rope r)     }     class Halter {         tie(Rope r)     }     Horse --&gt; Halter : HAS-A </pre>	<p>Horse class has a Halter, (cabestro, collarin)</p> <pre> public class Horse extends Animal {     private Halter myHalter = new Halter();      public void tie(LeadRope rope) {         myHalter.tie(rope); // Delegate tie behavior to                            // the Halter object     } }  public class Halter {     public void tie(LeadRope aRope) {         // Do the actual tie work here     } } </pre>

HAS-A relationships are based on use, 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.

- A subclass uses `super.overriddenMethodName()` to call the superclass version of an overridden method.
- A subclass uses `MyInterface.super.overriddenMethodName()` to call the super interface version on an overridden method.



&gt;&gt;

### 3. Polymorphism

*7.2 Develop code that demonstrates the use of polymorphism;  
Including overriding and object type versus reference type (sic).*

Remember that any Java object that can pass more than one **IS-A** test can be considered polymorphic. Without considering the `Object` class.

Remember, too, that the only way to access an object is through a **reference variable**.

- Polymorphism means “many forms.”
- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
- A single object can be referred to by reference variables of many different types—as long as they are the same type or a supertype of the object.
- The reference variable’s type (not the object’s type) determines which methods can be called!
- Polymorphic method invocations apply only to overridden instance methods.
- A reference variable can be declared as a class type or an interface type.
  - If the variable is declared as an interface type, it can reference any object of any class that *implements* the interface.

A *class* cannot *extend* more than one class: that means one parent per class. A class *can* have multiple ancestors. So ***Interfaces can have concrete methods (called default methods). This allows for a form of multiple inheritance.***

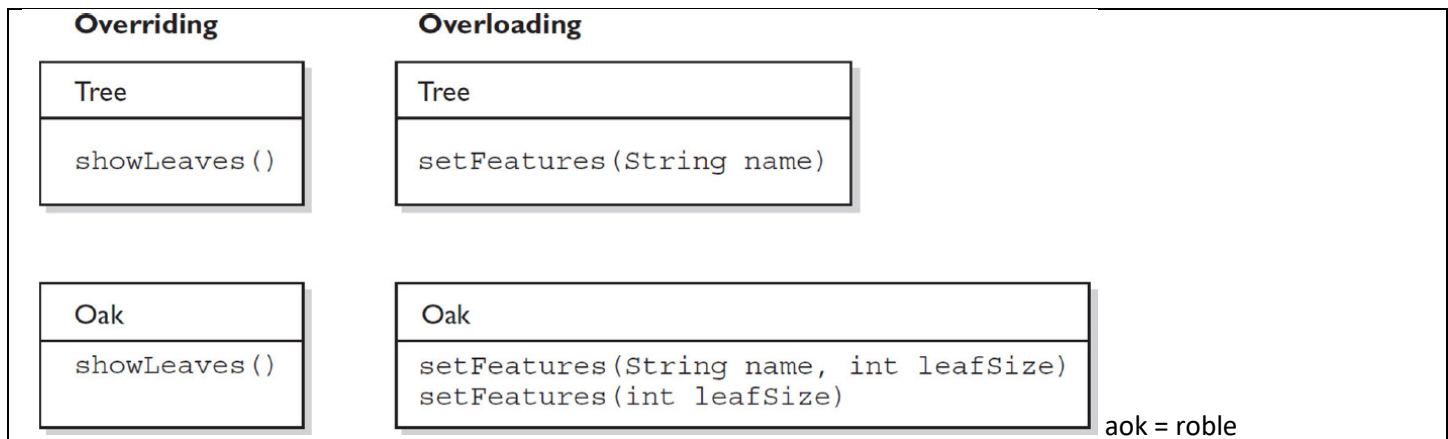
Class `PlayerPiece` extends `GameShape` implements `Animatable`

So now we have a `PlayerPiece` that passes the IS-A test for both the `GameShape` class and the `Animatable` interface. That means a `PlayerPiece` can be treated polymorphically as one of four things at any given time, depending on the declared type of the reference variable:

- An `Object` (since any object inherits from `Object`)
- A `GameShape` (since `PlayerPiece` extends `GameShape`)
- A `PlayerPiece` (since that’s what it really is)
- An `Animatable` (since `PlayerPiece` implements `Animatable`)

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
>>
```

## 4. Overriding/Overloading



```
import java.io.FileNotFoundException;

class X {}
class Y extends X {}

class A {
    protected X /*int*/ getValue() throws Exception { return null; } // overridden
}
class B extends A {
    public Y /*int*/ getValue() throws FileNotFoundException { return null; } // overriding
}
```

Animal a = new Animal();

Animal h = new Hourse(); // Animal ref, but a Hourse object

// h: Eres caballo pero solo con las variables y métodos de animal.

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- With respect to the method it overrides, **the overriding method**:
  - Must have the same argument list
  - Must have the same return type or a subclass (known as a covariant return)
  - Must not have a more restrictive access modifier
  - May have same or less restrictive access modifier
  - Must not throw new or broader checked exceptions
  - May throw fewer or narrower checked exceptions, or any unchecked exception
- `final` methods cannot be overridden.



- Only inherited methods may be overridden, and remember that private methods are not inherited.
- A subclass uses `super.overriddenMethodName()` to call the superclass version of an overridden method.
- A subclass uses `MyInterface.super.overriddenMethodName()` to call the super interface version on an overridden method.
- Overloading means reusing a method name but with different arguments.
- Overloaded methods
  - Must have different argument lists
  - May have different return types, if argument lists are also different
  - May have different access modifiers
  - May throw different exceptions
- Methods from a supertype can be overloaded in a subtype.
- Polymorphism applies to overriding, not to overloading.
- Object type (not the reference variable's type) determines which overridden method is used at runtime..
- Reference type (not the object type) determines which overloaded method will be used at compile time.

## Overridden Methods

Any time a type inherits a method from a supertype, you have the opportunity to override the method (unless, as you learned earlier, the method is marked `final`).

For abstract methods you inherit from a supertype, you have no choice:

You *must* implement the method in the subtype ***unless the subtype is also abstract.***

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.
- The access level can't be more restrictive than that of the overridden method.
- The access level CAN be less restrictive than that of the overridden method.
- 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.
- You cannot override a method marked neither `final` nor `static`.

To summarize, which *overridden* version of the method to call (in other words, from which class in the inheritance tree) is decided at runtime based on object type, but which *overloaded* version of the method to call is based on the reference type of the argument passed at compile time.

**TABLE 2-4** Differences Between Overloaded and Overridden Methods

	Overloaded Method	Overridden Method
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns. (Covered later this chapter.)
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	<i>Reference</i> type determines which overloaded version (based on declared argument types) is selected. Happens at <i>compile</i> time. The actual <i>method</i> that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the <i>signature</i> of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the <i>class</i> in which the method lives.	<i>Object</i> type (in other words, <i>the type of the actual instance on the heap</i> ) determines which method is selected. Happens at <i>runtime</i> .

Ocjp\_6 page 153, ocjp\_8 page 183

>>

## 5. Casting

*Differentiate between object reference variables and primitive variables.*

*Determine when casting is necessary.*

- There are two types of reference variable casting: upcasting and downcasting.
- **Upcasting** You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.
- **Downcasting** If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.

**upcasting** (casting *up* the inheritance tree to a more general type) works implicitly

```
Dog d = new Dog();           El perro es un Animal
Animal ad1 = d;              // implicit
Animal ad2 = (Animal) d;     // explicit
```

**downcast**, because we're casting down the inheritance tree to a more specific class.

```
Animal animal = new Animal();    Acaso Animal es un Perro
Dog dh = animal;                // Error de compilacion: cannot convert from Animal to Dog
Dog d = (Dog)animal;            // Error de ejecucion: Animal cannot be cast to Dog
```

<pre>Animal ad = new Dog(); Dog d = (Dog) ad; d.eat();</pre>	<pre>Animal ad = new Dog(); ((Dog)ad).eat();</pre> <p>In this case the compiler needs all those parentheses; otherwise, It thinks it's been handed an incomplete statement.</p>
--	---

>>

## 6. [Implementing an Interface](#)

### 7.5 *Use abstract classes and interfaces.*

- When you implement an interface, you are fulfilling its contract.
- You implement an interface by properly and concretely implementing all the abstract methods defined by the interface.
- A single class can implement many interfaces.

As of Java 8, interfaces can have concrete methods, which are labeled `default`.

```
interface I1 {
    default int doStuff() {
        return 1;
    }
}
```

Also, remember that a class can implement more than one interface

```
public class Ball implements Bounceable, Serializable, Runnable { ... }
```

You can extend only one class, but you can implement many interfaces

```
class BeachBall extends Ball implements Bounceable, Inflatable { . . . }
```

An interface can itself extend another interface or interfaces.

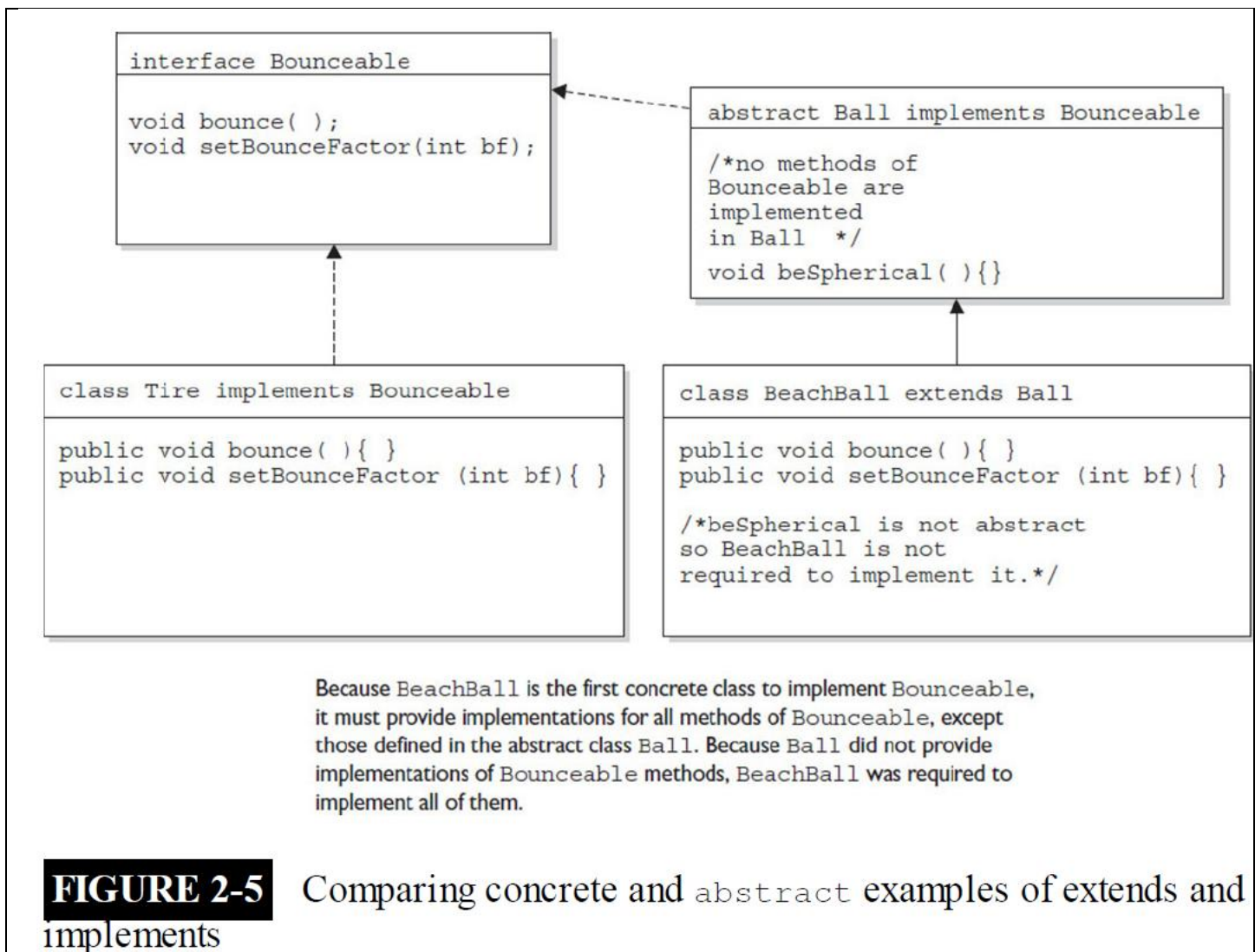
```
public interface Bounceable extends Moveable, Spherical { } // ok!
```

```
interface Bounceable {
    void bounce();
}
interface Inflatable {
    void inflate();
}

class Ball {}

class BeachBall extends Ball implements Bounceable, Inflatable {
    @Override
    public void bounce() {
        System.out.println("The beach ball bounces!");
    }

    @Override
    public void inflate() {
        System.out.println("Unimplemented method 'inflate'");
    }
}
```





***Look for illegal uses of extends and implements. The following shows examples of legal and illegal class and interface declarations:***

```

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)

```

```

interface I1 {
    default int doStuff() { return 1; }
}

interface I2 {
    default int doStuff() { return 2; }
}

public class MultiInterface implements I1, I2 { // Error duplicados doStuff

    public static void main(String[] args) {
        new MultiInterface().go();
    }

    void go() {
        System.out.println(doStuff()); // 3
    }
}
/*
public int doStuff() { // Must be public, Cannot reduce the visibility
    return 3;
}
*/
}

```

&gt;&gt;



## 7. Return Types

2.2 Differentiate between object reference variables and primitive variables.

6.1 Create methods with arguments and return values; including overloaded methods.

This section covers two aspects of return types: whether you're overriding an inherited method or simply declaring a new method (which includes overloaded methods).

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
- Object reference return types can accept `null` as a return value. `return null;`
- An array is a legal return type, both to declare and return as a value.
- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
- Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return` in any method with a `void` return type to bust out of a method early. But you can't return nothing from a method with a non-`void` return type.
- Methods with an object reference return type can return a subtype.
- Methods with an interface return type can return any implementer.

### 7.1. Return Types on Overloaded Methods

You can declare any return type you like. What you can't do is change only the return type. Therefore, the return type doesn't have to match that of the supertype version. To overload a method, remember, you must change the argument list.

### 7.2. Overriding and Return Types and Covariant Returns

You are 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.

<pre>class Alpha {     Alpha doStuff(char c) {         return new Alpha();     } } class Beta extends Alpha {     Beta doStuff(char c) {         return new Beta();     } }  public int foo() {     char c = 'c';     return c; // char is compatible with int }</pre>	<pre>public abstract class Animal { } public class Bear extends Animal { } public class Test {     public Animal go() {         return new Bear(); // OK, Bear "is-a" Animal     } }  Gum is Chewable public interface Chewable { } public class Gum implements Chewable { }  public class TestChewable {     // Method with an interface return type     public Chewable getChewable() {         return new Gum(); // Return interface implementer     } }</pre>
--	---

>>

## 8. Constructors and Instantiation

You CANNOT make a new object without invoking a constructor. In fact, you can't make a new object without invoking not just the constructor of the object's actual class type, but also the constructor of **each** of its superclasses! Constructors are the code that runs whenever you use the keyword `new`. They have no return type, and their names must exactly match the class name.

If you do not provide a constructor for your class, the compiler will insert one. The compiler generated constructor is called the default constructor, and it is always a noarg constructor with a no-arg call to `super()`. The default constructor will **never** be generated if even a single constructor exists in your class.

Constructors are not inherited. A constructor can invoke another constructor of the same class using the keyword `this()`. Every constructor must have either `this()` or `super()` as the **first** statement (although the compiler can insert it for you).

### Constructors and Instantiation:

- The default constructor is ALWAYS a no-arg constructor.
- A constructor is always invoked when a new object is created.
- Each superclass in an object's inheritance tree will have a constructor called.
- Every class, even an abstract class, has at least one constructor.
- Constructors must have the same name as the class.
- Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class; it's not a constructor.
- Typical constructor execution occurs as follows:
  - The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the `Object` constructor.
  - The `Object` constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created. "Los constructores se recorren de arriba abajo".
- Constructors can use any access modifier (even `private`!).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a no-arg constructor with a no-arg call to `super()`.
- The first statement of every constructor must be a call either to `this()` (an overloaded constructor) or to `super()`.



- The compiler will add a call to `super()` unless you have already put in a call to `this()` or `super()`.
- Instance members are accessible only after the `super` constructor runs.
- Abstract classes have constructors that are called when a concrete subclass is instantiated.
- Interfaces do not have constructors.
- If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.
- Constructors are never inherited; thus they cannot be overridden.
- A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).
- Regarding issues with calls to `this()`:
  - They may appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.
  - Constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.
  - Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.
- It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor.
- A call to `super()` can either be a no-arg call or can include arguments passed to the super constructor.

<pre>class Animal {}  class Horse extends Animal {}  class Main {     public static void main(String[] args) {         Horse h = new Horse();     } }</pre>	<h3>Constructors on the call stack</h3> <table><tr><td>4. Object()</td></tr><tr><td>3. Animal() calls super()</td></tr><tr><td>2. Horse() calls super()</td></tr><tr><td>1. main() calls new Horse()</td></tr></table> <p>Se ejecutan de arriba hacia abajo: 4, 3, 2, 1</p>	4. Object()	3. Animal() calls super()	2. Horse() calls super()	1. main() calls new Horse()
4. Object()					
3. Animal() calls super()					
2. Horse() calls super()					
1. main() calls new Horse()					

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)
<code>class Foo { }</code>	<code>class Foo {     <b>Foo()</b> {         super();     } }</code>
<code>class Foo {     Foo() { } }</code>	<code>class Foo {     Foo() {         <b>super();</b>     } }</code>
<code>public class Foo { }</code>	<code>public class Foo {     <b>public Foo()</b> {         super();     } }</code>
<code>class Foo {     Foo(String s) { } }</code>	<code>class Foo {     Foo(String s) {         <b>super();</b>     } }</code>
<code>class Foo {     Foo(String s) {         super();     } }</code>	<i>Nothing; compiler doesn't need to insert anything.</i>
<code>class Foo {     void Foo() { } }</code>	<code>class Foo {     void Foo() { }     <b>Foo()</b> {         super();     } }</code> (void Foo() is a method, not a constructor.)

<code>class Animal {     Animal(String name) { } }  class Horse extends Animal {     Horse() {         super(); // Problem!     } }</code>	<code>class Clothing {     Clothing(String s) { } } class TShirt extends Clothing { } // Problem</code>
<code>class A {     A() {         this("foo");     }     A(String s) {         this();     } } // Problem!</code>	<code>class Clothing {     Clothing(String s) { } } class TShirt extends Clothing {     // Constructor identical to compiler-supplied     // default constructor     TShirt() {         super(); // Won't work!     } // tries to invoke a no-arg Clothing constructor     // but there isn't one } // Problem</code>
<code>class Horse {     Horse() { } // constructor     void doStuff() {         Horse(); // calling the constructor - illegal!     } }</code>	

&gt;&gt;

## 9. Initialization Blocks

### 1.2 Define the structure of a Java class

### 6.3 Create and overload constructors; including impact on default constructors

<p>There are two blocks:</p> <ul style="list-style-type: none"> <li>• Static <u>initialization</u> block</li> <li>• Instance initialization block</li> </ul>	<pre>class SmallInt {     static int x;     int y;      static { x = 5; } // static init block     { y = 10; }      // instance init block }</pre>
--	--

Remember these rules:

- 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()`.
  - `init` blocks execute in the order in which they appear.
- Use `static init` blocks—`static { /* code here */ }` —for code you want to have run once, when the class is first loaded.  
Multiple blocks run from the top down.
  - Use normal `init` blocks—`{ /* code here }` —for code you want to have run for every new instance, right after all the super constructors have run. Again, multiple blocks run from the top of the class down.

First, static statements/blocks are called IN THE ORDER they are defined.

Next, instance initializer statements/blocks are called IN THE ORDER they are defined.

Finally, the constructor is called.

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

>>

## 10. Statics

### 6.2 *Apply the static keyword to methods and fields.*

Static members are tied (ligados) to the class or interface, not an instance.

Therefore, there is only one copy of any static member. Use the respective class or interface name with the **dot operator** to access `static` members.

A common mistake is to attempt to reference an instance variable from a `static` method.

"State", is represented by instance fields.

- you don't actually need to initialize a static variable to zero;
- Think `static` = class, `nonstatic` = instance.
- Use `static` methods to implement behaviors that are not affected by the state of any instances.
- Use `static` variables to hold data that is class specific as opposed to instance specific there will be only one copy of a `static` variable.
- All `static` members belong to the class, not to any instance.
- A `static` method can't access an instance variable directly.
- *you can't access a nonstatic (instance) variable from a static method.*
- Use the dot operator to access `static` members, but remember that using a reference variable with the dot operator is really a syntax **trick**, and the compiler will substitute the class name for the reference variable; for instance:  
`d.doStuff();` becomes `Dog.doStuff();`
- To invoke an interface's static method use `MyInterface.doStuff()` syntax.
- Remember that *static methods can't be overridden!* but they can be redefined..

Un metodo no puede estar definido dos veces, aunque uno sea estatico y el otro no. Lo mismo con las variables.

Los métodos estáticos y no estáticos con la misma firma no están permitidos en un ámbito.

A class cannot have two methods with the same signature in its scope where one is static and one is instance

```
Padre ph = new Hijo();
```

```
ph.metodoEstatico(); // Imprime "Método estático de la clase Padre"      trick
```

```
Padre.metodoEstatico();
```

<pre> class Foo  int size = 42; static void doMore( ){     int x = <del>size</del>; } </pre>	static method cannot access an instance (nonstatic) variable
<pre> class Bar  void go(){ } static void doMore( ){     <del>go</del>( ); } </pre>	static method cannot access a nonstatic method
<pre> class Baz  static int count; static void woo( ){ } static void doMore( ){     woo( );     int x = count; } </pre>	static method <i>can</i> access a static method or variable

**FIGURE 2-8** The effects of `static` on methods and variables

Estatico solo llama a estatico

Crecimiento Vertical: Jerarquico

Crecimiento Horizontal : Especializado.

>>

## 11. Coupling and Cohesion (scjp)

- ✓ Coupling refers to the degree to which one class knows about or uses members of another class.
- ✓ Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage.
- ✓ Tight coupling is the undesirable state of having classes that break the rules of loose coupling.
- ✓ Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.
- ✓ High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.
- ✓ Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.

Deseable: Alta cohesion y Bajo acoplamiento

>>

## CERTIFICATION SUMMARY

For the exam, be sure you know that overloaded methods can change the return type, but overriding methods can do so only within the bounds of covariant returns.

**Overriding methods** must declare the same argument list and return type or they can return a subtype of the declared return type of the supertype's overridden method), and that the access modifier can't be more restrictive. The overriding method also can't throw any new or broader checked exceptions that weren't declared in the overridden method. You also learned that the overridden method can be invoked using the syntax `super.doSomething();`.

An overriding method must have the same return type as the overridden method of the superclass.

**Overloaded methods** let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Whereas overriding methods must not change the argument list, overloaded methods must. But unlike overriding methods, overloaded methods are free to vary the return type, access modifier, and declared exceptions any way they like.

It is legal to return any value or variable that can be implicitly converted to the declared return type. So, for example, a `short` can be returned when the return type is declared as an `int`. And (assuming `Horse` extends `Animal`), a `Horse` reference can be returned when the return type is declared an `Animal`.

Every constructor must have either `this()` or `super()` as the first statement (although the compiler can insert it for you).

A common mistake is to attempt to reference an instance variable from a `static` method. Use the respective class or interface name with the dot operator to access `static` members.

As of Java 8, interfaces can have concrete methods, which are labeled `default`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

And, once again, you learned that the exam includes tricky questions designed largely to test your ability to recognize just how tricky the questions can be.