C2 Object Orientation

Java 8 Associate

1Z0-808

Link

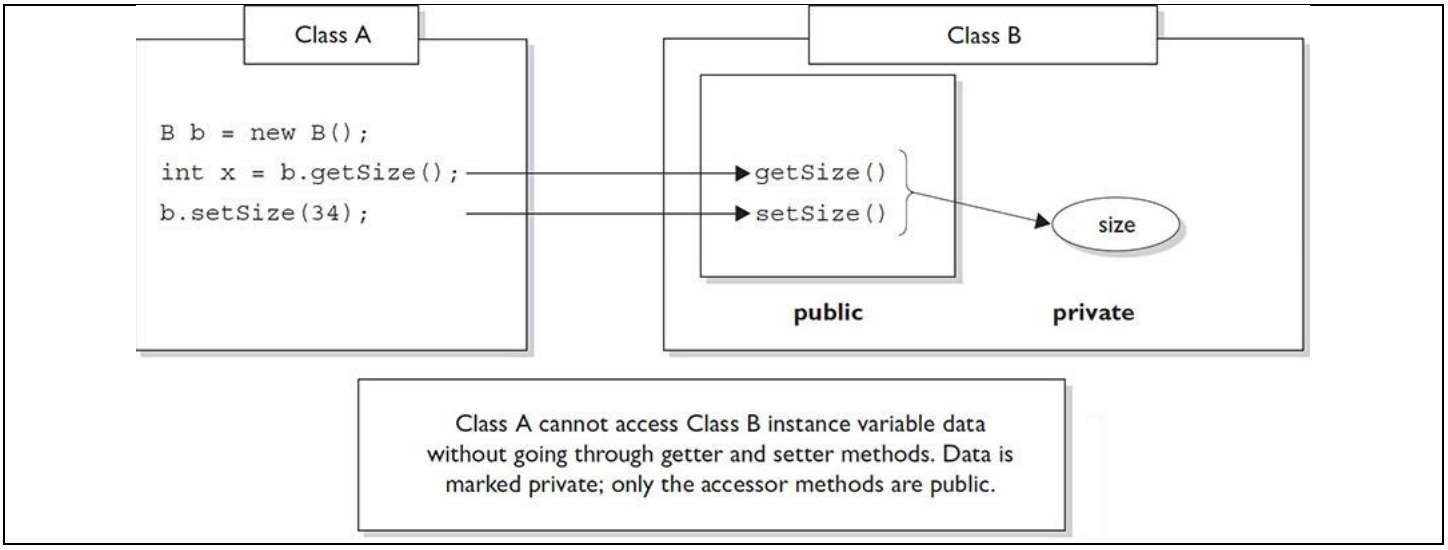
https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387
https://docs.oracle.com/javase/specs/jls/se8/html
https://docs.oracle.com/javase/tutorial
http://www.java2s.com

https://enthuware.com
https://github.com/carlosherrerah/OCA

JUAN CARLOS HERRERA HERNANDEZ
carlos.herrera@upa.edu.mx

Contenido

1. Encapsulation



- 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.

>>

2. Inheritance and Polymorphism

```
System.out.println("Paris" instanceof  String);
```

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, overloading, 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

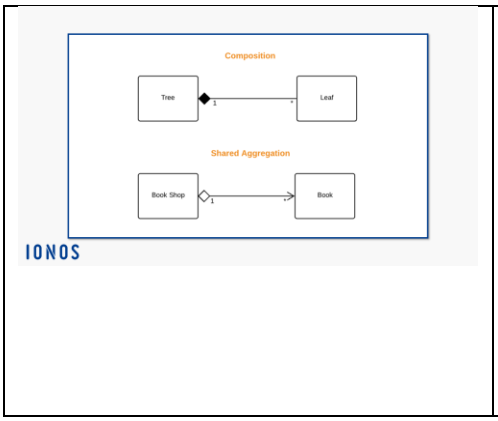
IS-A and HAS-A Relationships

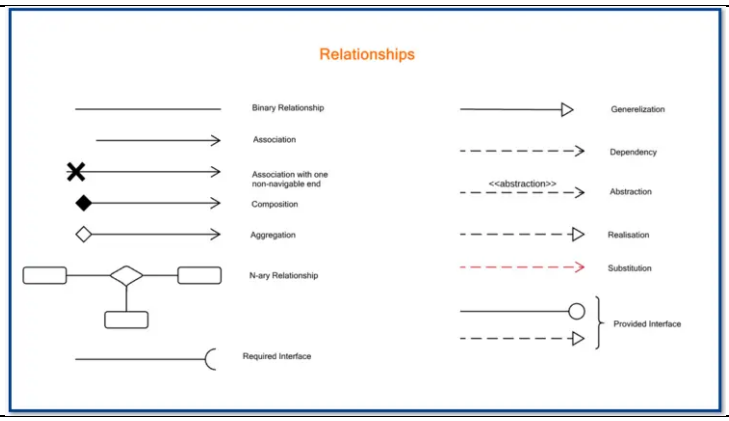
- IS-A refers to inheritance or implementation.
- IS-A is expressed with the keyword `extends` or `implements`.

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

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.





>>

3. Polymorphism

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."
- Polymorphic method invocations apply only to **overridden** *instance* methods.

- A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
- A reference variable **can refer to any** *subtype* **of the declared type!**
- 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.
Interfaces can have concrete methods (called default methods). This allows for a form of multiple interhance.

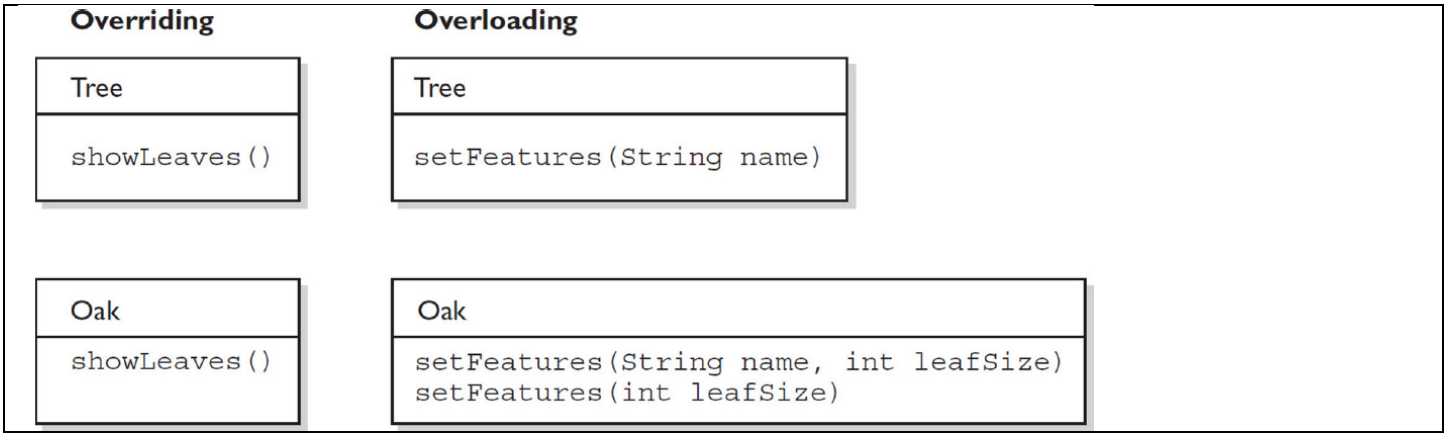
`Class PlayerPiece extends GameShape implements Animatable`

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 0 = 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: Contenido de las variables de Animal, pero métodos de Hourse que existan en 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 a 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.

- Overloading means reusing a method name but with **different arguments**.
- The overriding method, May have a less restrictive access modifier than `overridden`.
- The overriding method, May throw fewer or narrower checked exceptions, or any unchecked exception.
- The overriding method, Must have the same return type or a subclass (known as a covariant return)
- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.

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 over*ridden* 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 over*loaded* 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	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the *class* in which the method lives.	*Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

>>

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 (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
```

Animal ad = new Dog(); Dog d = (Dog) ad; d.eat();	Animal ad = new Dog(); ((Dog)ad).eat();

>>

6. Implementing an Interface

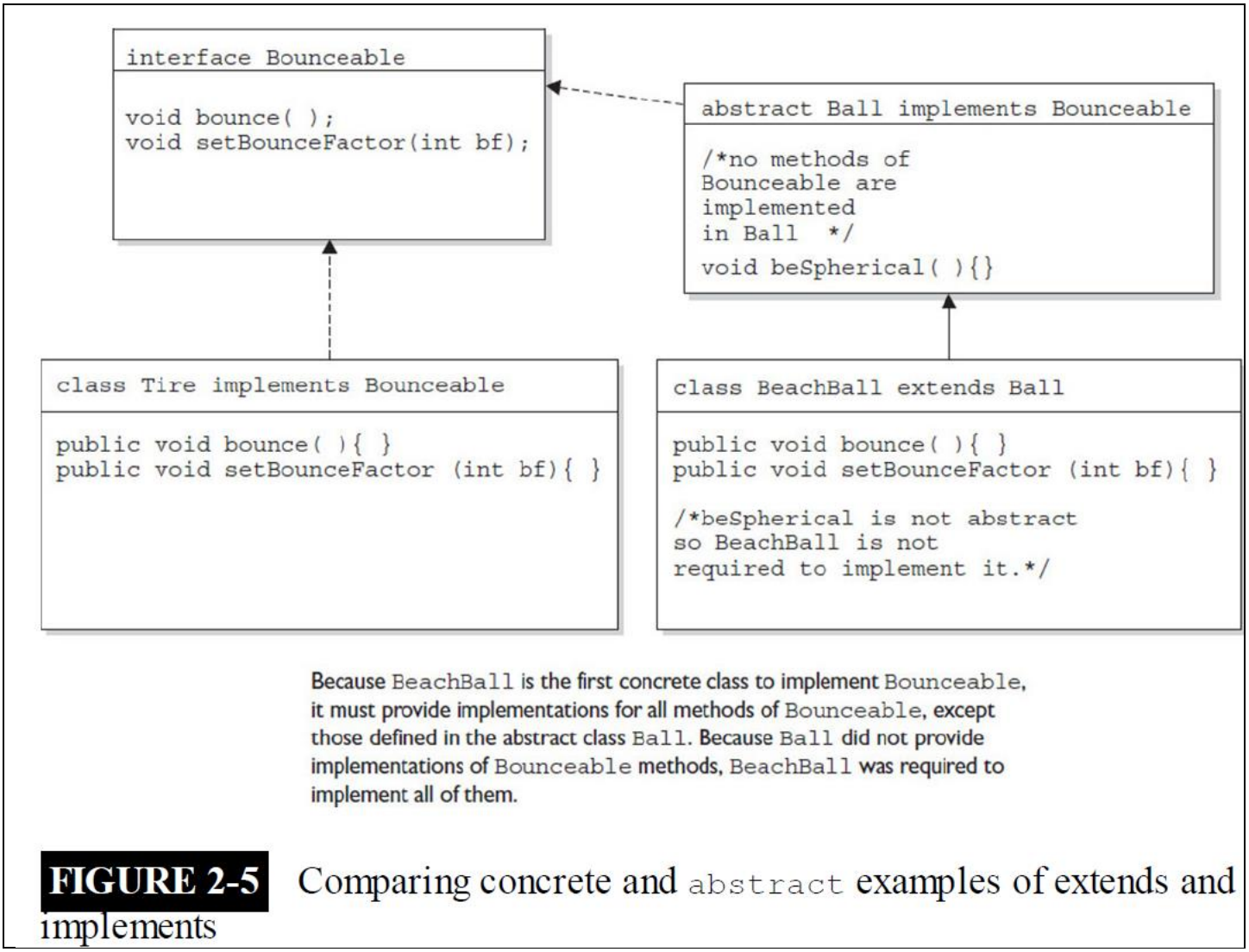
As of Java 8, interfaces can have concrete methods, which are labeled `default`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

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

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

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

An interface can itself extend another interface.



Because `BeachBall` is the first concrete class to implement `Bounceable`, it must provide implementations for all methods of `Bounceable`, except those defined in the abstract class `Ball`. Because `Ball` did not provide implementations of `Bounceable` methods, `BeachBall` was required to implement all of them.

FIGURE 2-5 Comparing concrete and `abstract` examples of extends and implements

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 Il {
    default int doStuff() { return 1; }
}

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

public class MultiInterface implements Il, I2 {     // duplicados doStuff

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

    void go() {
        System.out.println(doStuff());
    }
/*
    public int doStuff() {                      // debe ser public
         return 3;
    }
*/
}
```

>>

7. Legal Return Types

2.2 Differentiate between object reference variables and primitive variables.
6.1Create 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).

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.

```
  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
}
```

```
public abstract class Animal { }
public class Bear extends Animal { }
public class Test {
  public Animal go() {
    return new Bear();  // OK, Bear "is-a" Animal
  }
}

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
  }
}
```

>>

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 constructor name must match the name of the class.
- Constructors must not have a return type.
- The default constructor is ALWAYS a no-arg constructor.
- The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the `Object` constructor.
- 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()`.
- 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.

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)
`class Foo { }`	`class Foo {` `  Foo() {` `  super();` `  }` `}`
`class Foo {` `  Foo() { }` `}`	`class Foo {` `  Foo() {` `  super();` `  }` `}`
`public class Foo { }`	`public 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.)

```
class Animal {
  Animal(String name) { }
}

class Horse extends Animal {
  Horse() {
    super();  // Problem!
  }
}
```

```
class Clothing {
  Clothing(String s) { }
}
class TShirt extends Clothing { }
```

```
class A {
  A() {
    this("foo");
  }
  A(String s) {
    this();
  }
}
```

```
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
}
```

>>

9. Initialization Blocks

There are two blocks:

Static initialization blocks
Instance `init` block

Remember these rules:

- `init` blocks execute in the order in which 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()`.

- 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.

>>

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.

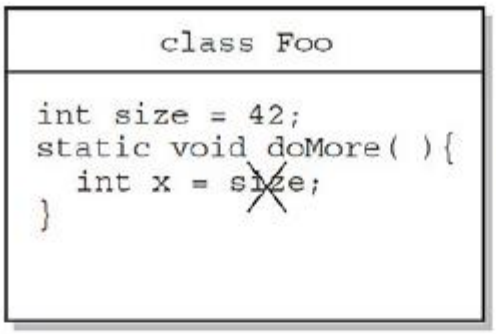
"State", is represented by instance fields.

- 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*!

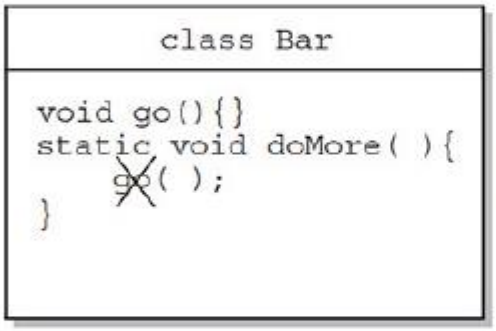
Un metodo no puede estar definido dos veces, aunque uno sea estatico y el otro no. Lo mismo con las variables.

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

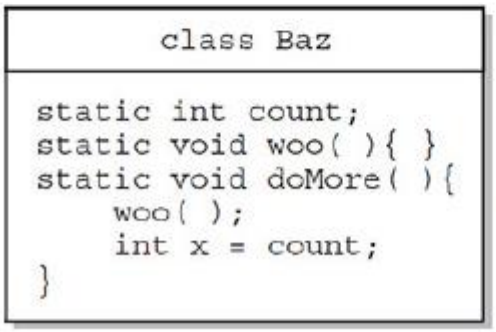
A class cannot have two methods with the same signature in its scope where one is static and one is instance



static method cannot access an instance (nonstatic) variable



static method cannot access a nonstatic method



static method *can* access a static method or variable

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

>>