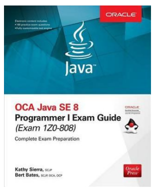


OCA Java SE 8
Programmer I Exam Guide
(Exam 1Z0-808)
Complete Exam Preparation
Kathy Sierra
Bert Bates

Chapter 2

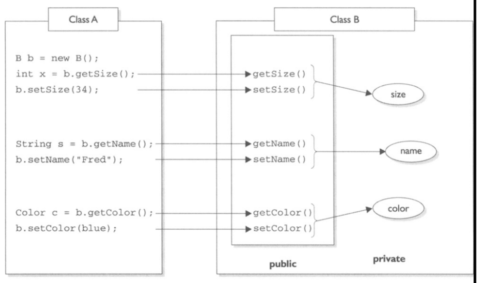
Object Orientation

Content

Encapsulation
(OCA Objectives 6.1 and 6.5)

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>.



```
B b = new B();
int x = b.getSize();
b.setSize(34);

String s = b.getName();
b.setName("Fred");

Color c = b.getColor();
b.setColor(blue);
```

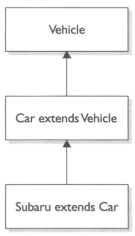
Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

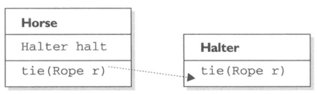
Inheritance and Polymorphism
(OCA Objectives 7.1 and 7.2)

1Z0-808 Chapter 2

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.

M. Romdhani, 2019

6

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

Polymorphism
(OCA Objective 7.2)

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

(OCA Objectives 6.1 and 7.2)

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!**

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

- **Polymorphism in Overloaded and Overridden Methods**
  - Polymorphism doesn't matter when a method is overloaded.

```
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.

Overriding / Overloading

- **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*.

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)
```

Java 8—Now with Multiple Inheritance!

- A class **CAN** implement interfaces with duplicate, concrete method signatures! But the good news is that the compiler's got your back, and if you **DO** want to implement both interfaces, you'll have to provide an overriding method in your class. Let's look at the following code:

```
interface I1 {
  default int doStuff() { return 1; }
}
interface I2 {
  default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {  // needs to override
  public static void main(String[] args) {
    new MultiInt().go();
  }
  void go() {
    System.out.println(doStuff());
}
//  public int doStuff() {
//    return 3;
// }
}
```

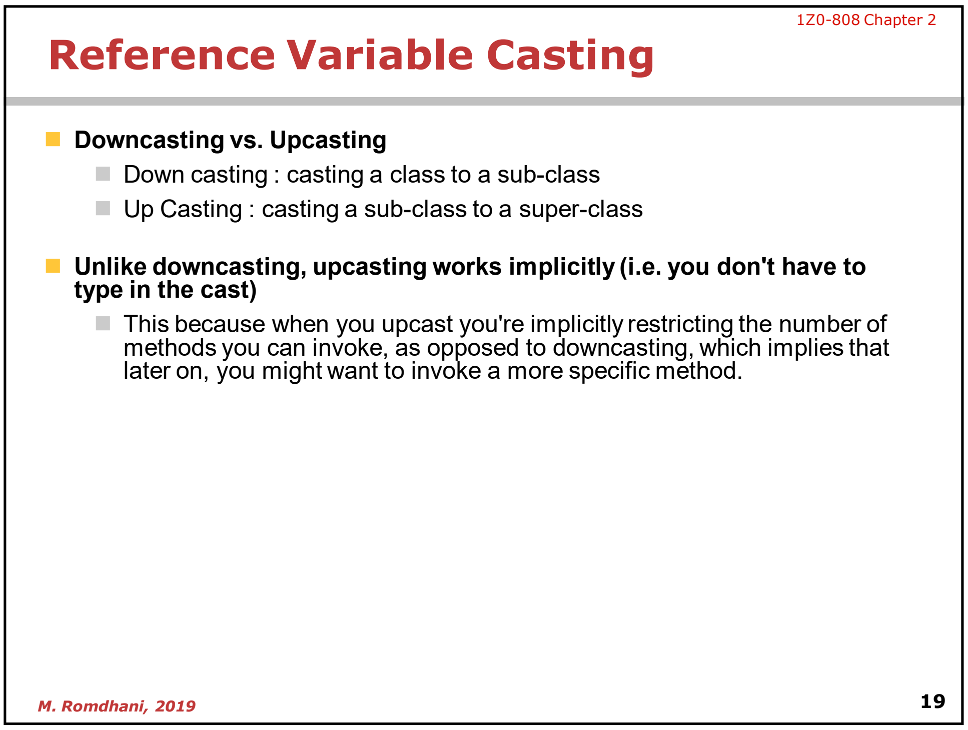
  - As the code stands, it WILL NOT COMPILE because it's not clear which version of doStuff() should be used. In order to make the code compile, you need to override doStuff() in the class. Uncommenting the class's doStuff() method would allow the code to compile
  - It is possible to specfiy a default method in go() using I1.**super**.doStuff()

Static methods and inheritance

- **Static methods in interface in Java 8 cannot be inherited by the class implementing it.**
  - And that makes sense, as a class can implement multiple interface. And if 2 interfaces have same static method, they both would be inherited, and compiler wouldn't know which one to invoke.
- **However, with extending class, that's no issue. static class methods are inherited by subclass**

Casting

(OCA Objectives 2.2 and 7.3)

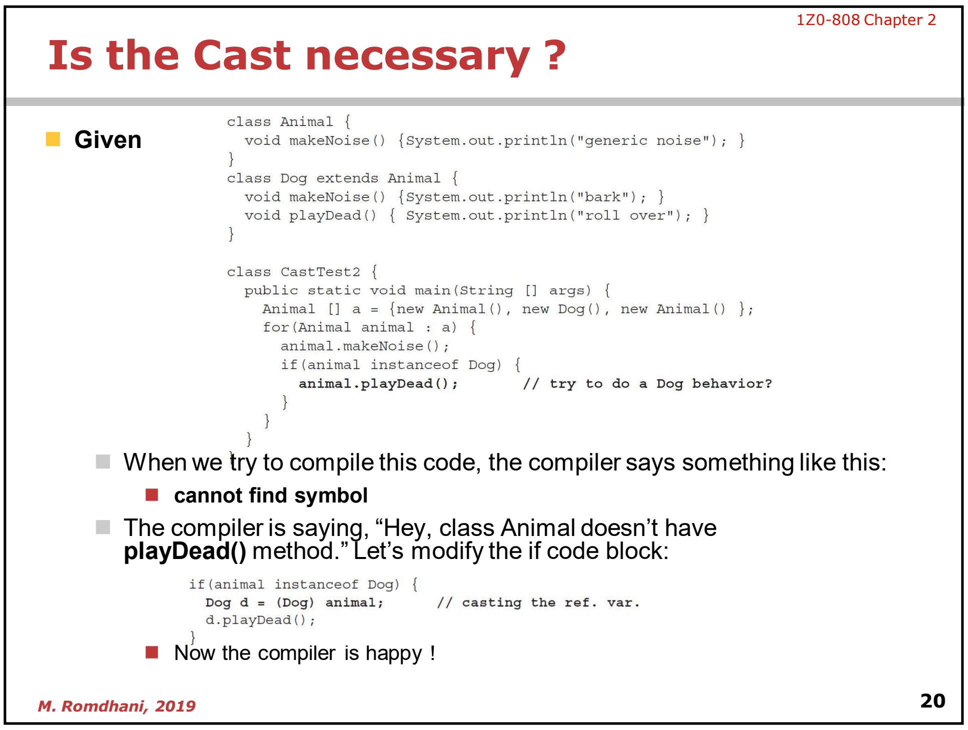


1Z0-808 Chapter 2

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.

M. Romdhani, 2019 **19**



1Z0-808 Chapter 2

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 !

M. Romdhani, 2019 **20**

Implementing an Interface
(OCA Objective 7.5)

Implementing an Interface

- **When you implement an interface, you're agreeing to adhere to the contract defined in the interface.**
  - That means you're agreeing to provide legal implementations for every method defined in the interface, and that anyone who knows what the interface methods look like can rest assured that they can invoke those methods on an instance of your implementing class.
- **Two more rules you need to know and then we can put this topic to sleep (or put you to sleep; we always get those two confused):**
  1. A class can implement more than one interface. It's perfectly legal to say, for example, the following:
     - public class Ball implements Bounceable, Serializable, Runnable{...}
  2. You can extend only one class, but implement many interfaces. An interface can itself extend another interface, but never implement anything. The following code is perfectly legal:
     - public interface Bounceable extends Moveable { } //ok!

Implementing an Interface

- **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! Interface can't
                                   // implement an interface
interface Zee implements Foo { }   //No! Interface can't
                                   // implement a class
interface Zoo extends Foo { }      //No! Interface can't
                                   // extend a class
interface Boo extends Fi { }       // OK. Interface can extend
                                   // an interface
class Toon extends Foo, Button { } //No! Class can't extend
                                   // multiple classes
class Zoom implements Fi, Fee { }  // OK. class can implement
                                   // multiple interfaces
interface Vroom extends Fi, Fee { }  // OK. interface can extend
                                     // multiple interfaces
class Yow extends Foo implements Fi { } // OK. Class can do both
                                        // (extends must be 1st)
```

Legal Return Types
(OCA Objectives 2.2 and 6.1)

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.

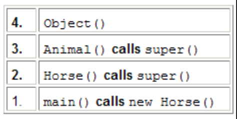
       ```
       public Animal getAnimal() {  return new Horse(); }
       ```

Constructors and Instantiation

(OCA Objectives 6.3 and 7.4)

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()`

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.

Initialization Blocks

(OCA Objectives 1.2 and 6.3)

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:**

Statics
(OCA Objective 6.2)

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

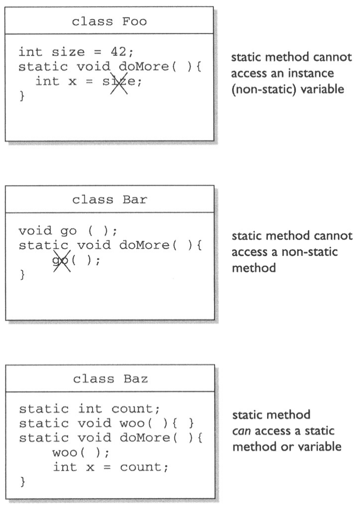
```
Frog count is now 3
```

- **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,

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

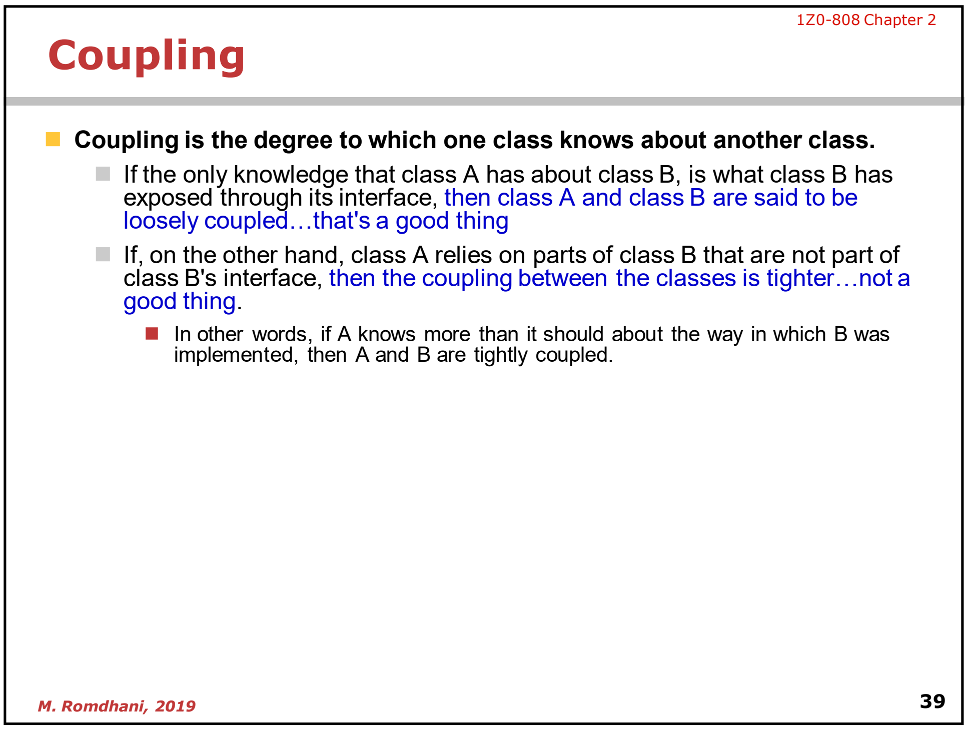
```
class Baz

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

static method *can* access a static method or variable

Coupling and cohesion

OO design perspective

- **Coupling and cohesion, have to do with the quality of an OO design.**
- **In general, good OO design calls for loose coupling and shuns tight coupling, and good OO design calls for high cohesion, and shuns low cohesion.**
  - As with most OO design discussions, the goals for an application are
    - Ease of creation
    - Ease of maintenance
    - Ease of enhancement

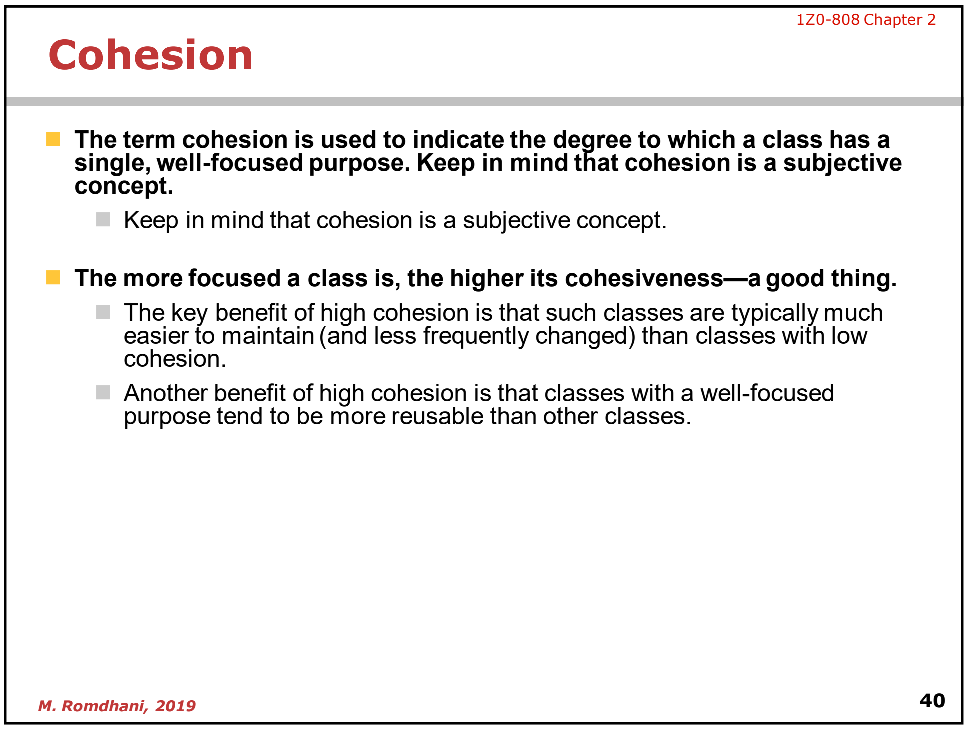


1Z0-808 Chapter 2

Coupling

- **Coupling is the degree to which one class knows about another class.**
  - If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled…that's a good thing
  - If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…not a good thing.
    - In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

M. Romdhani, 2019 **39**



1Z0-808 Chapter 2

Cohesion

- **The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Keep in mind that cohesion is a subjective concept.**
  - Keep in mind that cohesion is a subjective concept.
- **The more focused a class is, the higher its cohesiveness—a good thing.**
  - The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion.
  - Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

M. Romdhani, 2019 **40**