JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

5

Object Orientation, Overloading and Overriding, Constructors, and Return Types

CERTIFICATION OBJECTIVES

The objectives in this section revolve (mostly) around object-oriented (OO) programming including encapsulation, inheritance, and polymorphism. For the exam, you need to know whether a code fragment is correctly or incorrectly implementing some of the key OO features supported in Java. You also need to recognize the difference between overloaded and overridden methods, and be able to spot correct and incorrect implementations of both.

Because this book focuses on your passing the programmer's exam, only the critical exam-specific aspects of OO software will be covered here. If you're not already well versed in OO concepts, you could (and should) study a dozen books on the subject of OO development to get a broader and deeper understanding of both the benefits and the techniques for analysis, design, and implementation. But for passing the exam, the relevant concepts and rules you need to know are covered here. (That's a disclaimer, because we can't say you'll be a "complete OO being" by reading this chapter.) (We can say, however, that your golf swing will improve.)

We think you'll find this chapter a nice treat after slogging your way through the technical (and picky) details of the previous chapters. Object-oriented programming is a festive topic, so may we suggest you don the appropriate clothing—say, a Hawaiian shirt and a party hat. Grab a margarita (if you're new to OO, maybe nonalcoholic is best) and let's have some fun!

(OK so maybe we exaggerated a *little* about the whole party aspect. Still, you'll find this section both smaller and less detailed than the previous four.) (And this time we really mean it.)

CERTIFICATION OBJECTIVE

Benefits of Encapsulation (Exam Objective 6.1)

State the benefits of encapsulation in object-oriented design and write code that implements tightly encapsulated classes and the relationships IS-A and HAS-A.

Imagine you wrote the code for a class, and another dozen programmers from your company all wrote programs that used your class. Now imagine that you didn't like the way the class behaved, because some of its instance variables were being set (by

the other programmers from within their code) to values you hadn't anticipated. *Their* code brought out errors in *your* code. (Relax, this is just hypothetical…) Well, it *is* a Java program, so you should be able just to ship out a newer version of the class, which they could replace in their programs without changing any of their own code.

This scenario highlights two of the promises/benefits of OO: flexibility and maintainability. But those benefits don't come automatically. You have to *do* something. You have to write your classes and code in a way that *supports* flexibility and maintainability. So *what* if Java supports OO? It can't design your code for you. For example, imagine if you (not the *real* you, but the *hypothetical-not-as-good-a-programmer you*) made your class with `public` instance variables, and those other programmers were setting the instance variables directly, as the following code demonstrates:

```
public class BadOO {
   public int size;
   public int weight;
   …
}
public class ExploitBadOO {
   public static void main (String [] args) {
      BadOO b = new BadOO();
      b.size = -5; // Legal but bad!!
   }
}
```

And now you're in trouble. How are you going to change the class in a way that lets you handle the issues that come up when somebody changes the *size* variable to a value that causes problems? Your only choice is to go back in and write method code for adjusting *size* (a `setSize(int a)` method, for example), and then protect the *size* variable with, say, a `private` access modifier. But as soon as you make that change to *your* code, *you break everyone else's!*

The ability to make changes in your implementation code *without* breaking the code of others who use your code is a key benefit of encapsulation. *You want to hide implementation details behind a* `public` *programming interface.* By interface, we mean the set of accessible methods your code makes available for other code to call—in other words, *your code's API.* By hiding implementation details, you can rework your method code (perhaps also altering the way variables are used by your class) without forcing a change in the code that calls your changed method.

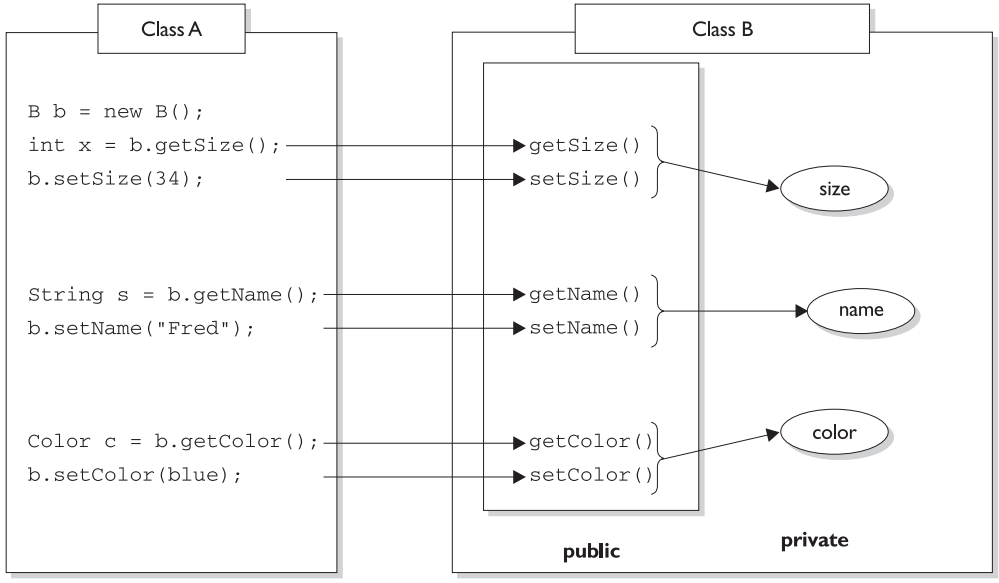
If you want maintainability, flexibility, and extensibility (and of course, you do), your design must include encapsulation. How do you do that?

- Keep your instance variables protected (with an access modifier, often `private`).
- Make *public* accessor methods, and force calling code to use those methods.
- For the methods, use the JavaBeans naming convention of set<someProperty> and get<someProperty>.

Figure 5-1 illustrates the idea that encapsulation forces callers of our code to go through methods rather than accessing variables directly.

We call the access methods *getters and setters* although some prefer the fancier terms (more impressive at dinner parties) *accessors and mutators.* Personally, we don't

FIGURE 5-1 The nature of encapsulation



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.

like the word *mutate*. Regardless of what you call them, they're methods that others must go through in order to access your instance variables. They look simple, and you've probably been using them forever:

```
public class Box {
   // protect the instance variable; only an instance
   // of Box can access it
   private int size;
   // Provide public getters and setters
   public int getSize() {
      return size;
   }
   public void setSize(int newSize) {
      size = newSize;
   }
}
```

Wait a minute…how useful is the previous code? It doesn't even do any validation or processing. What benefit can there be from having getters and setters that add no additional functionality? The point is, *you can change your mind later*, and add more code to your methods without breaking your API. Even if you don't think you really need validation or processing of the data, good OO design dictates that you plan for the future. To be safe, *force calling code to go through your methods rather than going directly to instance variables*. Always. Then you're free to rework your method implementations later, without risking the wrath of those dozen programmers who know where you live. And have been doing Tae-bo. And drink way too much Mountain Dew.

exam Watch

Look out for code that appears to be asking about the behavior of a method, when the problem is actually a lack of encapsulation. Look at the following example, and see if you can figure out what's going on:

```
class Foo {
   public int left = 9;
   public int right = 3;
   public void setLeft(int leftNum) {
      left = leftNum;
      right = leftNum/3;
   }
 // lots of complex test code here
}
```

Now consider this question: *Is the value of right always going to be one-third the value of left?* ***It looks like it will, until you realize that users of the Foo class don't need to use the*** `setLeft()` ***method! They can simply go straight to the instance variables and change them to any arbitrary*** *int* ***value.***

IS-A and HAS-A Relationships

For the exam you need to be able to look at code and determine whether the code demonstrates an IS-A or HAS-A relationship. The rules are simple, so this should be one of the few areas where answering the questions correctly is almost a no-brainer. (Well, at least it *would* have been a no-brainer if we (exam creators) hadn't tried our best to obfuscate the real problem.) (If you don't know the word "obfuscate", stop and look it up, then write and tell us what it means.)

IS-A

In OO, the concept of IS-A is based on inheritance. IS-A is a way of saying, "this thing *is a* type of that thing." For example, a Mustang is a type of horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car. Broccoli IS-A Vegetable (not a very fun one, but it still counts). You express the IS-A relationship in Java through the keyword `extends`:

```
public class Car {
  // Cool Car code goes here
}

public class Subaru extends Car {
   // Important Subaru-specific stuff goes here
   // Don't forget Subaru inherits accessible Car members
}
```

A Car is a type of Vehicle, so the inheritance tree might start from the Vehicle class as follows:

```
public class Vehicle { … }
public class Car extends Vehicle { … }
public class Subaru extends Car { … }
```

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 Car.
- Subaru inherits from Vehicle.
- Subaru is derived from Car.
- Car is derived from Vehicle.
- Subaru is derived from Vehicle.
- Subaru is a subtype of Car.
- Subaru is a subtype of Vehicle.

Returning to our IS-A relationship, the following statements are true:

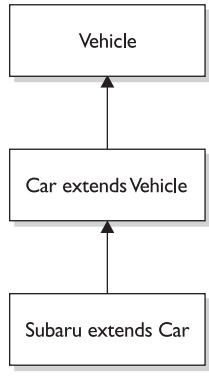
"Car extends Vehicle" means *"Car IS-A Vehicle."*
"Subaru extends Car" means *"Subaru IS-A Car."*

And we can also say:

"Subaru IS-A Vehicle" because a class is said to be "a type of" anything further up in its inheritance tree. If Foo `instanceof` Bar, then class Foo IS-A Bar, even if Foo doesn't directly extend Bar, but instead extends some other class that is a subclass of Bar. Figure 5-2 illustrates the inheritance tree for Vehicle, Car, and Subaru. The arrows move from the subclass to the superclass. In other words, a class' arrow points toward the class it extends from.

FIGURE 5-2

Inheritance tree for Vehicle, Car, and Subaru



HAS-A

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*. For example, you can say the following,

A Horse IS-A Animal. A Horse HAS-A Halter.

and the code looks like this:

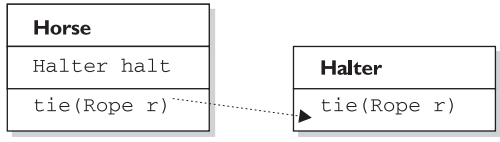
```
public class Animal { }
public class Horse extends Animal {
   private Halter myHalter;
}
```

In the preceding code, the Horse class has an instance variable of type Halter, so you can say that "Horse HAS-A Halter." In other words, *Horse has a reference to a Halter*. Horse code can use that Halter reference to invoke methods on the Halter, and get Halter behavior without having Halter-related code (methods) in the Horse class itself. Figure 5-3 illustrates the HAS-A relationship between Horse and Halter.

HAS-A relationships allow you to design classes that follow good OO practices by not having monolithic classes that do a gazillion different things. Classes (and thus the *objects* instantiated from those classes) should be specialists. The more specialized the class, the more likely it is that you can reuse the class in other applications. If you put all the Halter-related code directly into the Horse class, you'll end up duplicating code in the Cow class, Sheep class, UnpaidIntern class, and any other class that might need Halter behavior. By keeping the Halter code in a separate, specialized Halter class, you have the chance to reuse the Halter class in multiple applications.

FIGURE 5-3

HAS-A relationship between Horse and Halter



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.

Users of the Horse class (that is, code that calls methods on a Horse instance), *think* that the Horse class has Halter behavior. The Horse class might have a `tie(LeadRope rope)` method, for example. Users of the Horse class should never have to know that when they invoke the `tie()` method, the Horse object turns around and delegates the call to its Halter class by invoking `myHalter.tie(rope)`. The scenario just described might look like this:

```
public class Horse extends Animal {
   private Halter myHalter;
   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
   }
}
```

In OO, we don't want callers to worry about which class or which object is actually doing the real work. To make that happen, the Horse class hides implementation details from Horse users. Horse users ask the Horse object to do things (in this case, tie itself up), and the Horse will either do it or, as in this example, ask something else to do it. To the caller, though, it always appears that *the Horse object takes care of itself.* Users of a Horse should not even need to know that there *is* such a thing as a Halter class.

Now that we've looked at some of the OO characteristics, here are some possible scenario questions and their solutions.

SCENARIO & SOLUTION

What benefits do you gain from encapsulation?	Ease of code maintenance, extensibility, and code clarity.
What is the object-oriented relationship between a tree and an oak?	An IS-A relationship: Oak IS-A Tree.
What is the object-oriented relationship between a city and a road?	A HAS-A relationship: City HAS-A Road.

FROM THE CLASSROOM

Object-Oriented Design

IS-A and HAS-A relationships and encapsulation are just the tip of the iceberg when it comes to object-oriented design. Many books and graduate theses have been dedicated to this topic. The reason for the emphasis on proper design is simple: money. The cost to deliver a software application has been estimated to be as much as 10 times more expensive for poorly designed programs. Having seen the ramifications of poor designs, I can assure you that this estimate is not far-fetched.

Even the best object-oriented designers make mistakes. It is difficult to visualize the relationships between hundreds, or even thousands, of classes. When mistakes are discovered during the implementation (code writing) phase of a project, the amount of code that has to be rewritten can sometimes cause programming teams to start over from scratch.

The software industry has evolved to aid the designer. Visual object modeling languages, such as the Unified Modeling Language (UML), allow designers to design and easily modify classes without having to write code first, because object-oriented components are represented graphically. This allows the designer to create a map of the class relationships and helps them recognize errors before coding begins. Another recent innovation in object-oriented design is design patterns. Designers noticed that many object-oriented designs apply consistently from project to project, and that it was useful to apply the same designs because it reduced the potential to introduce new design errors. Object-oriented designers then started to share these designs with each other. Now, there are many catalogs of these design patterns both on the Internet and in book form.

Although passing the Java certification exam does not require you to understand object-oriented design this thoroughly, hopefully this background information will help you better appreciate why the test writers chose to include encapsulation and *is a* and *has a* relationships on the exam.

—Jonathan Meeks, Sun Certified Java Programmer

CERTIFICATION OBJECTIVE

Overridden and Overloaded Methods (Exam Objective 6.2)

Write code to invoke overridden or overloaded methods and parental or overloaded constructors, and describe the effect of invoking these methods.

Methods can be overloaded or overridden, but constructors can be only overloaded. *Overloaded* methods and constructors let you use the same method name (or constructor) but with different argument lists. *Overriding* lets you redefine a method in a subclass, when you need new subclass-specific behavior.

Overridden Methods

Anytime 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 following example demonstrates a Horse subclass of Animal overriding the Animal version of the `eat()` method:

```
public class Animal {
   public void eat() {
      System.out.println("Generic Animal Eating Generically");
   }
}
class Horse extends Animal {
   public void eat() {
       System.out.println("Horse eating hay, oats, and horse treats");
   }
}
```

For abstract methods you inherit from a superclass, you have no choice. You must implement the method in the subclass unless the subclass is *also* abstract. Abstract methods are said to be *implemented* by the concrete subclass, but this is virtually the same as saying that the concrete subclass *overrides* the abstract methods of the superclass. So you should *think of abstract methods as methods you're forced to override.*

The Animal class creator might have decided that for the purposes of polymorphism, all Animal subtypes should have an `eat()` method defined in a unique, specific way. Polymorphically, when someone has an Animal reference that refers not to an

Animal instance, but to an Animal *subclass* instance, the caller should be able to invoke `eat()` on the Animal reference, but the actual runtime object (say, a Horse instance) will run its own specific `eat()` method. Marking the `eat()` method abstract is the Animal programmer's way of saying to all subclass developers, "It doesn't make any sense for your new subtype to use a generic `eat()` method, so you have to come up with your own `eat()` method implementation!" An example of using polymorphism looks like this:

```
public class TestAnimals {
   public static void main (String [] args) {
      Animal a = new Animal();
      Animal b = new Horse();  //Animal ref, but a Horse object
      a.eat(); // Runs the Animal version of eat()
      b.eat(); // Runs the Horse version of eat()
   }
}
class Animal {
   public void eat() {
      System.out.println("Generic Animal Eating Generically");
   }
}
class Horse extends Animal {
   public void eat() {
       System.out.println("Horse eating hay, oats, and horse treats");
   }
   public void buck() { }
}
```

In the preceding code, the test class uses an Animal reference to invoke a method on a Horse object. Remember, the compiler will allow only methods in class Animal to be invoked when using a reference to an Animal. The following would not be legal given the preceding code:

```
Animal c = new Horse();
c.buck();  // Can't invoke buck();
           // Animal class doesn't have that method
```

The compiler looks only at the *reference* type, not the *instance* type. Polymorphism lets you use a more abstract supertype (including an interface) reference to refer to one of its subtypes (including interface implementers).

The overriding method *cannot have a more restrictive access modifier* than the method being overridden (for example, you can't override a method marked `public` and make it `protected`). Think about it: if the Animal class advertises a public `eat()`

method and someone has an Animal reference (in other words, a reference declared as type Animal), that someone will assume it's safe to call `eat()` on the Animal reference regardless of the actual instance that the Animal reference is referring to. If a subclass were allowed to sneak in and change the access modifier on the overriding method, then suddenly at runtime—when the JVM invokes the true *object's* (Horse) version of the method rather than the *reference type's* (Animal) version—the program would die a horrible death. (Not to mention the emotional distress for the one who was betrayed by the rogue subclass.) Let's modify the polymorphic example we saw earlier:

```
public class TestAnimals {
   public static void main (String [] args) {
      Animal a = new Animal();
      Animal b = new Horse();  //Animal ref, but a Horse object
      a.eat(); // Runs the Animal version of eat()
      b.eat(); // Runs the Horse version of eat()
   }
}
class Animal {
   public void eat() {
      System.out.println("Generic Animal Eating Generically");
   }
}
class Horse extends Animal {
   private void eat() {
       System.out.println("Horse eating hay, oats,
                           and horse treats");
   }
}
```

If this code were allowed to compile (which it's not, by the way—the compiler wants you to know that it didn't just fall off the turnip truck), the following would fail at runtime:

```
Animal b = new Horse();  // Animal ref, but a Horse
                         // object , so far so good
b.eat();                 // Meltdown!
```

The variable *b* is of type Animal, which has a `public eat()` method. But remember that at runtime, Java uses *virtual method invocation* to dynamically select the *actual* version of the method that will run, based on the *actual* instance. An Animal reference can always refer to a Horse instance, because Horse IS-A(n) Animal. What

makes that superclass reference to a subclass instance possible is that *the subclass is guaranteed to be able to do everything the superclass can do.* Whether the Horse instance overrides the inherited methods of Animal or simply inherits them, anyone with an Animal reference to a Horse instance is free to call all accessible Animal methods. For that reason, an overriding method must fulfill the contract of the superclass.

The rules for overriding a method are as follows:

- The *argument list must exactly match* that of the overridden method.
- The *return type must exactly match* that of the overridden method.
- The *access level must not 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 *must not throw new or broader checked exceptions* than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.
- 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`.
- *If a method can't be inherited, you cannot override it.* For example, the following code is not legal:

```
public class TestAnimals {
   public static void main (String [] args) {
      Horse h =  new Horse();
      h.eat(); // Not  legal because Horse didn't inherit eat()
   }
}
class Animal {
   private void eat() {
      System.out.println("Generic Animal Eating Generically");
   }
}
class Horse extends Animal { }
```

Invoking a Superclass Version of an Overridden Method

Often, you'll want to take advantage of some of the code in the superclass version of a method, yet still override it to provide some additional specific behavior. It's like saying, "Run the superclass version of the method, then come back down here and finish with my subclass additional method code." (Note that there's no requirement that the superclass version run before the subclass code.) It's easy to do in code using the keyword `super` as follows:

```
public class Animal {

   public void eat() { }
   public void printYourself() {
      // Useful printing code goes here
   }
}
class Horse extends Animal {
   public void printYourself() {
      // Take advantage of Animal code, then add some more
      super.printYourself();  // Invoke the superclass
                              // (Animal) code
                              // Then come back and do
                              // additional Horse-specific
                              // print work here
   }
}
```

Examples of Legal and Illegal Method Overrides

Let's take a look at overriding the `eat()` method of Animal:

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

Table 5-1 lists examples of illegal overrides of the Animal `eat()` method, given the preceding version of the Animal class.

Overloaded Methods

Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type). Overloading a method often means you're being a little nicer to those who call your methods, because *your* code

TABLE 5-1 Examples of Illegal Overrides

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 declared 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, but not an overload either because there's no change in the argument list

takes on the burden of coping with different argument types rather than forcing the *caller* to do conversions prior to invoking your method. 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.
- A method *can* be overloaded in the same class or in a subclass.

Legal Overloads

Let's look at a method we want to overload:

```
public void changeSize(int size, String name, float pattern) { }
```

The following methods are *legal overloads* of the `changeSize()` method:

```
public void changeSize(int size, String name) { }
public int changeSize(int size, float pattern) { }
public void changeSize(float pattern, String name)
                       throws IOException { }
```

exam Watch

Be careful to recognize when a method is overloaded rather than overridden. You might see a method that appears to be violating a rule for overriding, but which is actually a legal overload, as follows:

```
public class Foo {
   public void doStuff(int y, String s) { }
   public void moreThings(int x) { }
}
class Bar extends Foo {
   public void doStuff(int y, float s) throws IOException { }
}
```

You might be tempted to see the IOException as the problem, seeing that the overridden `doStuff()` ***method doesn't declare an exception, and knowing that IOException is checked by the compiler. But the*** `doStuff()` ***method is*** *not* ***overridden at all! Subclass Bar*** *overloads* ***the*** `doStuff()` ***method, by varying the argument list, so the IOException is fine.***

Invoking Overloaded Methods

When a method is invoked, more than one method of the same name might exist for the object type you're invoking a method on. For example, the Horse class might have three methods with the same name but with different argument lists, which means the method is overloaded.

Deciding which of the matching methods to invoke is based on the arguments. If you invoke the method with a String argument, the overloaded version that takes a String is called. If you invoke a method of the same name but pass it a *float*, the overloaded version that takes a *float* will run. If you invoke the method of the same name but pass it a Foo object, and there isn't an overloaded version that takes a Foo, then the compiler will complain that it can't find a match. The following are examples of invoking overloaded methods:

```
class Adder {
   public int addThem(int x, int y) {
      return x + y;
   }

   // Overload the addThem method to add doubles instead of ints
   public double addThem(double x, double y) {
      return x + y;
   }
}
// From another class, invoke the addThem() method
public class TestAdder {
   public static void main (String [] args) {
```

```
        Adder add = new Adder();
        int b = 27;
        int c = 3;
        int result = a.addThem(b,c); // Which addThem is invoked?
        double doubleResult = a.addThem(22.5,89.36);
        // Which addThem?
    }
}
```

In the preceding TestAdder code, the first call to `a.addThem(b,c)` passes two *int*s to the method, so the first version of `addThem()`—the overloaded version that takes two *int* arguments—is called. The second call to `a.addThem(22.5, 89.36)` passes two *doubles* to the method, so the second version of `addThem()`—the overloaded version that takes two *double* arguments—is called.

Invoking overloaded methods that take object references rather than primitives is a little more interesting. Say you have an overloaded method such that one version takes an Animal and one takes a Horse (subclass of Animal). If you pass a Horse object in the method invocation, you'll invoke the overloaded version that takes a Horse. Or so it looks at first glance:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
   public void doStuff(Animal a) {
      System.out.println("In the Animal version");
   }
   public void doStuff(Horse h) {
      System.out.println("In the Horse version");
   }
   public static void main (String [] args) {
      UseAnimals ua = new UseAnimals();
      Animal animalObj = new Animal();
      Horse horseObj = new Horse();
      ua.doStuff(animalObj);
      ua.doStuff(horseObj);
   }
}
```

The output is what you expect:

```
in the Animal version
in the Horse version
```

But what if you use an Animal reference to a Horse object?

```
Animal animalRefToHorse = new Horse();
 ua.doStuff(animalRefToHorse);
```

Which of the overloaded versions is invoked? You might want to say, "The one that takes a Horse, since it's a Horse object at runtime that's being passed to the method." But that's not how it works. The preceding code would actually print

```
in the Animal version
```

Even though the actual object at runtime is a Horse and not an Animal, the choice of which *overloaded* method to call is not dynamically decided at runtime. Just remember, *the reference type (not the object type) determines which overloaded method is invoked!* To summarize, which *overridden* 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 passed at *compile* time.

Polymorphism in Overloaded and Overridden Methods How does polymorphism work with overloaded methods? From what we just looked at, it doesn't appear that polymorphism matters when a method is *overloaded*. If you pass an Animal reference, the overloaded method that takes an Animal will be invoked, *even if the actual object passed is a Horse*. Once the Horse masquerading as Animal gets *in* to the method, however, the Horse object is still a Horse despite being passed into a method expecting an Animal. So it's true that polymorphism doesn't determine which *overloaded* version is called; polymorphism does come into play when the decision is about which *overridden* version of a method is called. But sometimes, a method is both overloaded *and* overridden. Imagine the Animal and Horse classes look like this:

```
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);
    }
}
```

Notice that the Horse class has both overloaded *and* overridden the `eat()` method. Table 5-2 shows which version of the three `eat()` methods will run depending on how they are invoked.

TABLE 5-2 Overloaded and Overridden Method Invocations

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 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 type, 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.

exam Watch

Don't be fooled by a method that's overloaded but not overridden by a subclass. It's perfectly legal to do the following:

```
public class Foo {
   void doStuff() { }
}
class Bar extends Foo {
   void doStuff(String s) { }
}
```

The Bar class has two `doStuff()` ***methods: the no-arg version it inherits from Foo (and does not override), and the overloaded*** `doStuff(String s)` ***defined in the Bar class. Code with a reference to a Foo can invoke only the no-arg version, but code with a reference to a Bar can invoke*** *either* ***of the overloaded versions.***

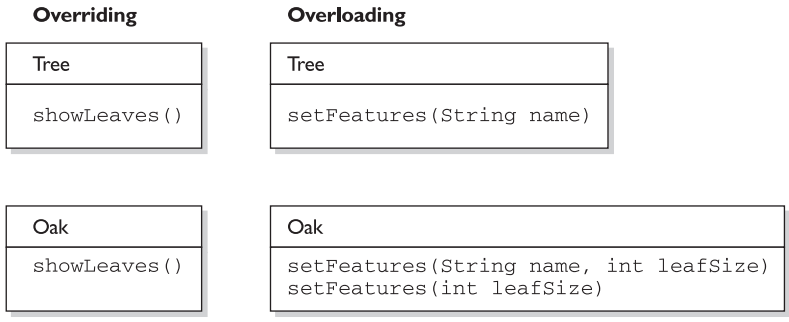
Table 5-3 summarizes the difference between overloaded and overridden methods.

TABLE 5-3 Difference Between Overloaded and Overridden Methods

	Overloaded Method	Overridden Method
argument list	Must change	Must not change
return type	Can change	Must not change
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 actual *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*.

FIGURE 5-4

Overloaded and overridden methods in class relationships



The current objective (5.2) covers both method and constructor overloading, but we'll cover constructor overloading in the next section, where we'll also cover the other constructor-related topics that are on the exam. Figure 5-4 illustrates the way overloaded and overridden methods appear in class relationships.

CERTIFICATION OBJECTIVE

Constructors and Instantiation (Exam Objectives 1.3, 6.3, 6.2)

For a given class, determine if a default constructor will be created, and if so, state the prototype of that constructor.

Write code to construct instances of any concrete class, including normal top-level classes and nested classes.

Write code to invoke parental or overloaded constructor, and describe the effect of those invocations.

Objects are constructed. *You can't 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`. We've got plenty to talk about here—we'll look at how constructors are coded, *who* codes them, and how they work at runtime. So grab your hardhat and a hammer, and let's do some object building. (Don't forget your lunch box and thermos.)

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. A constructor looks like this:

```
class Foo {
   Foo() { } // The constructor for the Foo class
}
```

Notice what's missing? *There's no return type!* Remember from Chapter 2 that a constructor has no return type and its name must exactly match the class name. Typically, constructors are used to initialize instance variable state, as follows:

```
class Foo {
   int size;
   String name;
   Foo(String name, int size) {
      this.name = name;
      this.size = size;
   }
}
```

In the preceding code example, the Foo class does not have a no-arg constructor. That means the following will *fail* to compile,

```
Foo f = new Foo();  // Won't compile, no matching constructor
```

but the following *will* compile,

```
Foo f = new Foo("Fred", 43);  // No problem. Arguments match Foo constructor.
```

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 (yes, constructors can be overloaded). You can't always make that work for your classes; occasionally you have a class where it makes no sense to create an instance without supplying information to the constructor. A java.awt.Color object, for example, can't be created by calling a no-arg constructor, because that would be like saying to the JVM, "Make me a new Color object, and I really don't care what color it is…*you* pick." (Imagine if the JVM were allowed to make aesthetic decisions. What if it's favorite color is mauve?)

Constructor Chaining

We know that constructors are invoked at runtime when you say `new` on some class type as follows:

```
Horse h = new Horse();
```

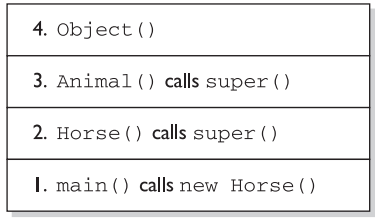
But what really happens when you say `new Horse()`?

1. Horse constructor is invoked.
2. Animal constructor is invoked (Animal is the superclass of Horse).
3. Object constructor is invoked (Object is the ultimate superclass of *all* classes, so class Animal extends Object even though you don't actually type `"extends Object" in to the Animal class declaration. It's implicit.) At this point we're` on the top of the stack.
4. Object instance variables are given their explicit values (if any).
5. Object constructor completes.
6. Animal instance variables are given their explicit values (if any).
7. Animal constructor completes.
8. Horse instance variables are given their explicit values (if any).
9. Horse constructor completes.

Figure 5-5 shows how constructors work on the call stack.

FIGURE 5-5

Constructors on the call stack



Rules for Constructors

The following list summarizes the rules you'll need to know for the exam (and to understand the rest of this section):

- Constructors can use any access modifier, including `private`. (A *private* constructor means only code within the class itself can instantiate an object of that type, so if the *private*-constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
- The constructor name must match the name of the class.
- Constructors must not have a return type.
- It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor.
- If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
- The default constructor is *always* a no-arg constructor.
- If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, *if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself*!
- Every constructor must have as its first statement either a call to an overloaded constructor (`this()`) or a call to the superclass constructor (`super()`).
- If you *do* type in a constructor (as opposed to relying on the compiler-generated default constructor), and you do *not* type in the call to `super()`, *the compiler will insert a no-arg call to* `super()` *for you.*
- A call to `super()` can be either a no-arg call or can include arguments passed to the super constructor.
- A no-arg constructor is not necessarily the *default* constructor, although the default constructor is always a no-arg constructor. *The default constructor is the one the compiler provides!* While the default constructor is *always* a no-arg constructor, you're free to put in your *own* no-arg constructor.

- You *cannot* make a call to an instance method, or access an instance variable, until *after* the super constructor runs.
- You *can* access static variables and methods, although you can use them only as part of the call to `super()` or `this()`. (`Example: super(Animal.DoThings())`)
- Abstract classes have constructors, and those constructors are *always* called when a concrete subclass is instantiated.
- Interfaces do *not* have constructors. Interfaces are not part of an object's inheritance tree.
- The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor as follows:

```
class Horse {
   Horse() { } // constructor
   void doStuff() {
     Horse();  // calling the constructor - illegal!
   }
}
```

Determine if a Default Constructor Will Be Created

The following example shows a Horse class with two constructors:

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

Will the compiler put in a default constructor for the class above? No!
How about for the following variation of the class?

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

Now will the compiler insert a default constructor? No!
What about this class?

```
class Horse { }
```

Now we're talking. The compiler will generate a default constructor for the preceding class, because the class doesn't have any constructors defined.

OK, what about *this* class?

```
class Horse {
   void Horse() { }
}
```

It might *look* like the compiler won't create one, since there already is a constructor in the Horse class. Or is there? Take another look at the preceding Horse class.

What's wrong with the `Horse()` constructor? It isn't a constructor at all! It's simply a method that happens to have the same name as the class. Remember, the return type is a dead giveaway that we're looking at a method, and not a constructor.

How do you know for sure whether a default constructor will be created? Because *you* didn't write *any* constructors in your class.

How do you know what the default constructor will look like? Because…

- The default constructor has the *same access modifier as the class.*
- The default constructor has *no arguments.*
- The default constructor includes a *no-arg call to the super constructor* (`super()`).

The Table 5-4 shows what the compiler will (or won't) generate for your class.

TABLE 5-4 Compiler-Generated Constructor Code

Class Code (What *You* Type)	Compiler-Generated Constructor Code (In Bold Type)
`class Foo {}`	*`class Foo {`* **`Foo() {`** **`super();`** **`}`** `}`

TABLE 5-4 Compiler-Generated Constructor Code *(continued)*

Class Code (What *You* Type)	Compiler-Generated Constructor Code (In Bold Type)
`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)

What happens if the super constructor has arguments? Constructors can have arguments just as methods can, and if you try to invoke a method that takes, say, an *int*, but you don't pass anything to the method, the compiler will complain as follows:

```
class Bar {
    void takeInt(int x) { }
}

class UseBar {
```

```
    public static void main (String [] args) {
      Bar b = new Bar();
      b.takeInt();  // Try to invoke a no-arg takeInt() method
    }
}
```

The compiler will complain that you can't invoke `takeInt()` without passing an *int*. Of course, the compiler enjoys the occasional riddle, so the message it spits out on some versions of the JVM (your mileage may vary) is less than obvious:

```
UseBar.java:7: takeInt(int) in Bar cannot be applied to ()
      b.takeInt();
       ^
```

But you get the idea. The bottom line is that there must be a match for the method. And by match, we mean that the argument types must be able to accept the values or variables you're passing, and in the order you're passing them. Which brings us back to constructors (and here you were thinking we'd never get there), which work exactly the same way.

So if your super constructor (that is, the constructor of your immediate superclass/parent) has arguments, you must type in the call to `super()`, supplying the appropriate arguments. Crucial point: if your superclass does *not* have a no-arg constructor, you *must* type a constructor in your class (the subclass) because *you need a place to put in the call to super with the appropriate arguments.*

The following is an example of the problem:

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

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

And once again the compiler treats us with the stunningly lucid:

```
Horse.java:7: cannot resolve symbol
symbol  : constructor Animal  ()
location: class Animal
      super();  // Problem!
      ^
```

If you're lucky (and eat all your vegetables, including broccoli), your compiler might be a little more explicit. But again, the problem is that there just isn't a match for what we're trying to invoke with `super()`—an Animal constructor with no arguments.

Another way to put this—and you can bet your favorite Grateful Dead t-shirt it'll be on the exam—is *if your superclass does not have a no-arg constructor, then in your subclass you will not be able to use the default constructor supplied by the compiler.* It's that simple. Because the compiler can only put in a call to a no-arg `super()`, you won't even be able to compile something like the following:

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

Trying to compile this code gives us exactly the same error we got when we put a constructor in the subclass with a call to the no-arg version of `super()`:

```
Clothing.java:4: cannot resolve symbol
symbol  : constructor Clothing  ()
location: class Clothing
class TShirt extends Clothing { }
^
```

In fact, the preceding Clothing and TShirt code is implicitly the same as the following code, where we've supplied a constructor for TShirt that's identical to the default constructor supplied by the compiler:

```
class Clothing {
   Clothing(String s) { }
}
class TShirt extends Clothing {
   // Constructor identical to compiler-supplied default constructor
   TShirt() {
      super();
   }
}
```

One last point on the whole default constructor thing (and it's probably very obvious, but we have to say it or we'll feel guilty for years), *constructors are never inherited.* They aren't methods. They can't be *overridden* (because they aren't methods and only methods can be overridden). So the type of constructor(s) your superclass has in no way determines the type of default constructor you'll get. Some folks mistakenly believe that the default constructor somehow matches the super

constructor, either by the arguments the default constructor will have (remember, *the default constructor is always a no-arg*), or by the arguments used in the compiler-supplied call to `super()`.

But although constructors can't be overridden, you've already seen that they can be overloaded, and typically are.

Overloaded Constructors

Overloading a constructor means typing in multiple versions of the constructor, each having a different argument lists, 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.

Overloading a constructor is used typically to provide alternate ways for clients to instantiate objects of your class. For example, if a client knows the animal name, they can pass that to an Animal constructor that takes a string. But if they don't know the name, the client can call the no-arg constructor and that constructor can supply a default name. Here's what it looks like:

```
 1. public class Animal {
 2.   String name;
 3.   Animal(String name) {
 4.     this.name = name;
 5.  }
 6.
 7.  Animal() {
 8.   this(makeRandomName());
 9.  }
10.
11.  static String makeRandomName() {
```

```
12.     int x = (int) (Math.random() * 5);
13.     String name = new String[] {"Fluffy", "Fido",
                                    "Rover", "Spike",
                                    "Gigi"}[x];
14.     return name;
15.  }
16.
17.  public static void main (String [] args) {
18.    Animal a = new Animal();
19.    System.out.println(a.name);
20.    Animal b = new Animal("Zeus");
21.    System.out.println(b.name);
22.  }
23. }
```

Running this code four times produces the output:

```
% java Animal
Gigi
Zeus

% java Animal
Fluffy
Zeus

% java Animal
Rover
Zeus

% java Animal
Fluffy
Zeus
```

There's a lot going on in the preceding code. Figure 5-6 shows the call stack for constructor invocations when a constructor is overloaded. Take a look at the call stack, and then let's walk through the code straight from the top.

- **Line 2** Declare a String instance variable *name*.
- **Lines 3–5** Constructor that takes a String, and assigns it to instance variable *name*.
- **Line 7** Here's where it gets fun. Assume every animal needs a name, but the client (calling code) might not always know what the name should be, so you'll assign a random name. The no-arg constructor generates a name by invoking the `makeRandomName()` method.

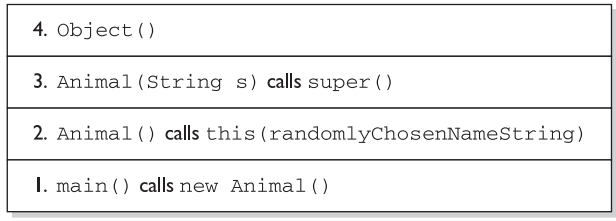


FIGURE 5-6

Overloaded constructors on the call stack

- **Line 8** The no-arg constructor invokes its own overloaded constructor that takes a string, in effect calling it the same way it would be called if client code were doing a `new` to instantiate an object, passing it a string for the name. The overloaded invocation uses the keyword `this`, but uses it as though it were a method name, `this()`. So line 8 is simply calling the constructor on line 3, passing it a randomly selected string rather than a client-code chosen name.
- **Line 11** Notice that the `makeRandomName()` method is marked `static`! That's because you *cannot* invoke an instance (in other words, *nonstatic*) method (or access an instance variable) until *after* the super constructor has run. And since the super constructor will be invoked from the constructor on line 3, rather than from the one on line 7, line 8 can use only a *static* method to generate the name. If we wanted all animals not specifically named by the caller to have the same default name, say, "Fred," then line 8 could have read

  ```
  this("Fred");
  ```

  rather than calling a method that returns a string with the randomly chosen name.
- **Line 12** Line 12 doesn't have anything to do with constructors, but since we're all here to learn…it generates a random number between 0 and 5.
- **Line 13** Weird syntax, we know. We're creating a new String object (just a single String instance), but we want the string to be selected randomly from a list. Except we don't have the list, so we need to make it. So in that one line of code we

  1. Declare a String variable, *name*.
  2. Create a String array (anonymously—we don't assign the array itself to anything).
  3. Retrieve the string at index [*x*] (*x* being the random number generated on line 12) of the newly created String array.

4. Assign the string retrieved from the array to the declared instance variable *name*. We could have made it *much* easier to read if we'd just written
```
String[] nameList = {"Fluffy", "Fido", "Rover",
"Spike", "Gigi"};
String name = nameList[x];
```

But where's the fun in that? Throwing in unusual syntax (especially for code wholly unrelated to the real question) is in the spirit of the exam. Don't be startled! (OK, be startled, but then just say to yourself, "Whoa" and get on with it.)

- **Line 18** We're invoking the no-arg version of the constructor (causing a random name from the list to be selected and passed to the *other* constructor).
- **Line 20** We're invoking the overloaded constructor that takes a string representing the *name*.

The key point to get from this code example is in line 8. Rather than calling `super()`, we're calling `this()`, and `this()` *always means a call to another constructor in the same class*. OK, fine, but what happens *after* the call to `this()`? Sooner or later the `super()` constructor gets called, right? Yes indeed. A call to `this()` just means you're delaying the inevitable. Some constructor, somewhere, must make the call to `super()`.

Key Rule: The first line in a constructor must be a call to `super()` or a call to `this()`.

No exceptions. If you have *neither* of those calls in your constructor, the compiler will insert the no-arg call to `super()`. In other words, if constructor `A()` has a call to `this()`, the compiler knows that constructor `A()` will *not* be the one to invoke `super()`.

exam Watch

The preceding rule means a constructor can never have both a call to `super()` and a call to `this()`. Because each of those calls must be the very first statement in a constructor, you can't legally use both in the same constructor. That also means the compiler will not put a call to `super()` in any constructor that has a call to `this()`.

Thought Question: What do you think will happen if you try to compile the following code?

```
class A {
   A() {
```

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

Your compiler may not actually catch the problem (it varies depending on your compiler, but most won't catch the problem). It assumes you know what you're doing. Can you spot the flaw? Given that a super constructor must always be called, where would the call to `super()` go? Remember, the compiler won't put in a default constructor if you've already got one or more constructors in your class. And when the compiler doesn't put in a default constructor, it *still* inserts a call to `super()` in any constructor that doesn't explicitly have a call to the super constructor—unless, that is, the constructor already has a call to `this()`. So in the preceding code, where can `super()` go? The only two constructors in the class both have calls to `this()`, and in fact you'll get exactly what you'd get if you typed the following method code:

```
public void go() {
   doStuff();
}

public void doStuff() {
   go();
}
```

Now can you see the problem? Of course you can. *The stack explodes!* It gets higher and higher and higher until it just bursts open and method code goes spilling out, oozing out of the JVM right onto the floor. Two overloaded constructors both calling `this()` are two constructors calling each other. Over and over and over, resulting in

```
% java A
Exception in thread "main" java.lang.StackOverflowError
```

The benefit of having overloaded constructors is that you offer flexible ways to instantiate objects from your class. The benefit of having one constructor invoke another overloaded constructor is to avoid code duplication. In the Animal example, there wasn't any code other than setting the name, but imagine if after line 4 there was still more work to be done in the constructor. By putting all the other constructor work in just one constructor, and then having the other constructors invoke it, you don't have to write and maintain multiple versions of that other

important constructor code. Basically, each of the other not-the-real-one overloaded constructors will call another overloaded constructor, passing it whatever data it needs (data the client code didn't supply).

Constructors and instantiation become even *more* exciting (just when you thought it was safe) when you get to inner classes, but we know you can only stand to have so much fun in one chapter, so we're holding the rest of the discussion on instantiating inner classes until Chapter 8.

CERTIFICATION OBJECTIVE

Legal Return Types (Exam Objective 1.4)

Identify legal return types for any method given the declarations of all related methods in this or parent classes.

This objective covers two aspects of return types: What you can *declare* as a return type, and what you can actually *return* as a value. What you can and cannot declare is pretty straightforward, but it all depends on whether you're overriding an inherited method or simply declaring a new method (which includes overloaded methods). We'll take just a quick look at the difference between return type rules for overloaded and overriding methods, because we've already covered that in this chapter. We'll cover a small bit of new ground, though, when we look at polymorphic return types and the rules for what is and is not legal to actually *return*.

Return Type Declarations

This section looks at what you're allowed to declare as a return type, which depends primarily on whether you are overriding, overloading, or declaring a new method.

Return Types on Overloaded Methods

Remember that method overloading is not much more than name reuse. The overloaded method is a completely different method from any other method of the same name. So if you inherit a method but *overload* it in a subclass, you're not subject to the restrictions of overriding, which means you can declare any return type you like. What you *can't* do is change *just* the return type. To overload a method, remember, *you must change the argument list.* The following code shows an overloaded method:

```
public class Foo{
   void go() { }
}
public class Bar extends Foo {
   String go(int x) {
     return null;
   }
}
```

Notice that the Bar version of the method uses a different return type. That's perfectly fine. As long as you've changed the argument list, you're overloading the method, so the return type doesn't have to match that of the superclass version. What you're not allowed to do is this:

```
public class Foo{
   void go() { }
}
public class Bar extends Foo {
   String go() { // Not legal! Can't change only the return type
      return null;
   }
}
```

Overriding and Return Types

When a subclass wants to change the method implementation of an inherited method, the subclass must define a method that matches the inherited version *exactly*. As we saw earlier in this chapter, an exact match means the arguments and return types must be identical. Other rules apply to overriding, including those for access modifiers and declared exceptions, but those rules aren't relevant to the return type discussion.

For the exam, be sure you know that *overloaded* methods *can* change the return type, but *overriding* methods *cannot*. Just that knowledge alone will help you through a wide range of exam questions.

Returning a Value

You have to remember only six rules for returning a value:

1. You can return `null` in a method that has 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();  // Assume Horse extends Animal
}

public Object getObject() {
    int[] nums = {1,2,3};
    return nums;  // Return an int array, which is still an object
}

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
        }
    }
```

exam Watch

Watch for methods that declare an abstract class or interface return type, and know that any object that passes the IS-A test (in other words, would test `true` using the `instanceof` operator) can be returned from that method—for example:

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

exam Watch

Be sure you understand the rules for casting primitives. Take a look at the following:

```
public short s = (short) (90 + 900000);
```

The preceding code compiles fine. But look at this variation:

```
public short s = (short) 90 + 900000;  // Illegal!
```

By leaving off the parentheses around the arithmetic expression, the cast (short) applies only to the first number! So the compiler gives us

```
Test.java:4: possible loss of precision
found   : int
required: short
   short s = (short) 90 + 900000;
                     ^
```

Casting rules matter when returning values, so the following code would not compile,

```
public short foo() {
   return (short) 90 + 900000;
}
```

but with parentheses around `(90 + 900000)`, it compiles fine.

CERTIFICATION SUMMARY

Let's take a stroll through Chapter 5 and see where we've been. You looked at how encapsulation can save you from being ripped to shreds by programmers whose code you could break if you change the way client code accesses your data. Protecting the instance variables (often by marking them private) and providing more accessible getter and setter methods represent the good OO practice of encapsulation, and support flexibility and maintainability by hiding your implementation details from other code.

You learned that inheritance relationships are described using IS-A, as in "Car IS-A Vehicle," and that the keyword `extends` is used to define IS-A relationships in Java:

```
class Car extends Vehicle
```

You also learned that reference relationships are described using HAS-A, as in "Car HAS-A Engine." HAS-A relationships in Java often are defined by giving one class a reference to another, usually through instance variable declarations:

```
class Car extends Vehicle {
    private Engine eng;  // Now Car has-a Engine,
                         // and can thus invoke
methods on it.
}
```

We looked at the difference between overridden and overloaded methods, learning that an overridden method occurs when a subclass inherits a method from a superclass, but the subclass redefines it to add more specialized behavior. We learned that at runtime, the JVM will invoke the subclass version on an instance of a subclass, and the superclass version on an instance of the superclass. Remember that abstract methods *must* be overridden (*technically* abstract methods must be *implemented*, as opposed to overridden, since there really isn't anything *to* override in an abstract method, but who's counting?).

We saw that overriding methods must keep the same argument list and return type as the 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();`.

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.

We covered constructors in detail, learning that even if you don't provide a constructor for your class, the compiler will always insert one. The compiler-generated constructor is called the *default* constructor, and it is *always* a no-arg constructor with a no-arg call to `super()`. The default constructor will never be generated if there is even a single constructor in your class (and regardless of the arguments of that constructor), so if you need more than one constructor in your class and you want a no-arg constructor, you'll have to write it yourself. We also saw that constructors are not inherited, and that you can be confused by a method that has the same name as the class (which is legal). The return type is the giveaway that a method is not a constructor, since constructors do not have return types.

We saw how all of the constructors in an object's inheritance tree will always be invoked when the object is instantiated using `new`. We also saw that constructors can be overloaded, which means defining constructors with different argument lists. A constructor can invoke another constructor of the same class using the keyword `this()`, as though the constructor were a method named `this()`. We saw that every constructor *must* have either `this()` or `super()` as the first statement.

We also looked at method return types, and saw that you can declare any return type you like (assuming you have access to a class for an object reference return type), unless you're overriding a method. An overriding method must have the same return type as the overridden method of the superclass. We saw that while overriding methods must *not* change the return type, overloaded methods *can* (as long as they *also* change the argument list).

Finally, you learned that 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 a Horse reference can be returned when the return type is declared an Animal (assuming Horse extends Animal).

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. If you took our advice about the margarita, you might want to review the following Two-Minute Drill again after you're sober.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 5.

Encapsulation, IS-A, HAS-A (Sun Objective 6.1)

- ❑ The goal of encapsulation is to 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.
- ❑ IS-A is expressed with the keyword `extends`.
- ❑ "IS-A," "inherits from," "is derived 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.

Overriding and Overloading (Sun Objective 6.2)

- ❑ Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- ❑ Abstract methods *must* be overridden by the first concrete (nonabstract) subclass.
- ❑ With respect to the method it overrides, the overriding method
  - ❑ Must have the same argument list
  - ❑ Must have the same return type
  - ❑ 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.
- ❑ A subclass uses `super.overriddenMethodName` to call the superclass version of an overridden method.
- ❑ Overloading means reusing the same method name, but with different arguments.
- ❑ Overloaded methods
  - ❑ Must have different argument lists
  - ❑ May have different return types, as long as the argument lists are also different
  - ❑ May have different access modifiers
  - ❑ May throw different exceptions
- ❑ Methods from a superclass can be overloaded in a subclass.
- ❑ Polymorphism applies to overriding, not to overloading
- ❑ Object type determines which overridden method is used at runtime.
- ❑ Reference type determines which overloaded method will be used at compile time.

Instantiation and Constructors (Sun Objectives 6.3 and 1.3)

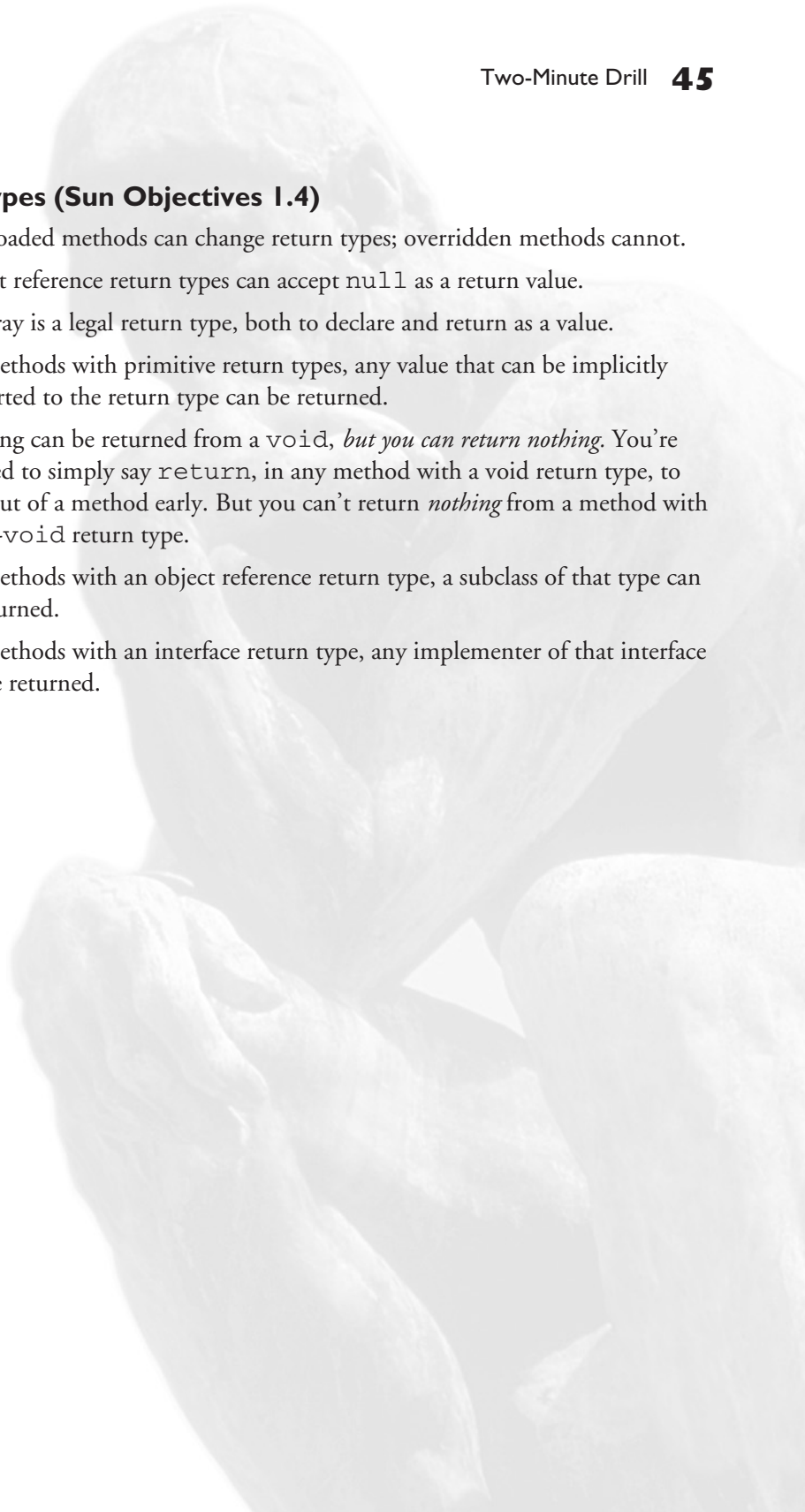
- ❑ Objects are constructed:
  - ❑ You cannot create a new object without invoking a constructor.
  - ❑ Each superclass in an object's inheritance tree will have a constructor called.
- ❑ Every class, even abstract classes, has at least one constructor.
- ❑ Constructors must have the same name as the class.
- ❑ Constructors do not have a return type. If there *is* a return type, then it is simply a method with the same name as the class, and not a constructor.
- ❑ 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.

- 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 to either `this()` (an overloaded constructor) or `super()`.
- The compiler will add a call to `super()` if you do not, unless you have already put in a call to `this()`.
- Instance methods and variables are only accessible *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()`).
- Issues with calls to `this()`:
  - May appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.
  - Constructors can call constructors can call constructors, and so on, but sooner or later *one* of them better call `super()` or the stack will explode.
  - `this()` and `super()` *cannot* be in the same constructor. You can have one or the other, but never both.

Return Types (Sun Objectives 1.4)

- ❑ Overloaded methods can change return types; overridden methods cannot.
- ❑ Object reference return types can accept `null` as a return value.
- ❑ 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.
- ❑ For methods with an object reference return type, a subclass of that type can be returned.
- ❑ For methods with an interface return type, any implementer of that interface can be returned.



SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Don't even *think* about skipping this test. You really need to see what the questions on the exam can be like, and check your grasp and memorization of this chapter's topics.

Encapsulation, IS-A, HAS-A (Sun Objective 6.1)

1. Given the following,

```
1.   public class Barbell {
2.     public int getWeight() {
3.       return weight;
4.     }
5.     public void setWeight(int w) {
6.       weight = w;
7.     }
8.     public int weight;
9.   }
```

which is true about the class described above?

A. Class Barbell is tightly encapsulated.

B. Line 2 is in conflict with encapsulation.

C. Line 5 is in conflict with encapsulation.

D. Line 8 is in conflict with encapsulation.

E. Lines 5 and 8 are in conflict with encapsulation.

F. Lines 2, 5, and 8 are in conflict with encapsulation.

2. Given the following,

```
1.   public class B extends A {
2.     private int bar;
3.     public void setBar(int b) {
4.       bar = b;
5.     }
6.   }
7.   class A {
8.     public int foo;
9.   }
```

which is true about the classes described above?

A. Class A is tightly encapsulated.

B. Class B is tightly encapsulated.
C. Classes A and B are both tightly encapsulated.
D. Neither class A nor class B is tightly encapsulated.

3. Which is true?

A. Tightly encapsulated classes are typically easier to reuse.
B. Tightly encapsulated classes typically use inheritance more than unencapsulated classes.
C. Methods in tightly encapsulated classes cannot be overridden.
D. Methods in tightly encapsulated classes cannot be overloaded.
E. Tightly encapsulated classes typically do not use HAS-A relationships.

4. Which two are *not* benefits of encapsulation? (Choose two.)

A. Clarity of code
B. Code efficiency
C. The ability to add functionality later on
D. Modifications require fewer coding changes
E. Access modifiers become optional

5. Given the following,

```
1.   class B extends A {
2.     int getID() {
3.       return id;
4.     }
5.   }
6.   class C {
7.     public int name;
8.   }
9.   class A {
10.    C c = new C();
11.    public int id;
12.  }
```

which two are true about instances of the classes listed above? (Choose two.)

A. `A is-a B`
B. `C is-a A`
C. `A has-a C`
D. `B has-a A`
E. `B has-a C`

Overriding and Overloading (Sun Objective 6.2)

6. Given the following,

```
class A {
 public void baz() {
   System.out.println("A");
 }
}
public class B extends A {
  public static void main(String [] args) {
    A a = new B();
    a.baz();
  }
  public void baz() {
    System.out.println("B");
  }
}
```

what is the result?

A. `A`

B. `B`

C. Compilation fails.

D. An exception is thrown at runtime.

7. Given the following,

```
class Foo {
  String doStuff(int x) { return "hello"; }
}
```

which method would not be legal in a subclass of Foo?

A. `String doStuff(int x) { return "hello"; }`

B. `int doStuff(int x) { return 42; }`

C. `public String doStuff(int x) { return "Hello"; }`

D. `protected String doStuff(int x) { return "Hello"; }`

E. `String doStuff(String s) { return "Hello"; }`

F. `int doStuff(String s) { return 42; }`

8. Given the following,

```
1.   class ParentClass {
2.     public int doStuff(int x) {
3.       return x * 2;
```

```
4.      }
5.    }
6.
7.    public class ChildClass extends ParentClass {
8.      public static void main(String [] args ) {
9.        ChildClass cc = new ChildClass();
10.       long x = cc.doStuff(7);
11.       System.out.println("x = " + x);
12.     }
13.
14.     public long doStuff(int x) {
15.       return x * 3;
16.     }
17.   }
```

What is the result?

A. `x = 14`

B. `x = 21`

C. Compilation fails at line 2.

D. Compilation fails at line 11.

E. Compilation fails at line 14.

F. An exception is thrown at runtime.

9. Given the following,

```
1.  class Over {
2.    int doStuff(int a, float b) {
3.      return 7;
4.    }
5.  }
6.
7.  class Over2 extends Over {
8.      // insert code here
9.  }
```

which two methods, if inserted independently at line 8, will not compile? (Choose two.)

A. `public int doStuff(int x, float y) { return 4; }`

B. `protected int doStuff(int x, float y) {return 4; }`

C. `private int doStuff(int x, float y) {return 4; }`

D. `private int doStuff(int x, double y) { return 4; }`

E. `long doStuff(int x, float y) { return 4; }`

F. `int doStuff(float x, int y) { return 4; }`

Instantiation and Constructors (Sun Objectives 6.3 and 1.3)

10. Given the following,

```
1.   public class TestPoly {
2.     public static void main(String [] args ){
3.       Parent p = new Child();
4.     }
5.   }
6.
7.   class Parent {
8.     public Parent() {
9.       super();
10.      System.out.println("instantiate a parent");
11.    }
12.  }
13.
14.  class Child extends Parent {
15.    public Child() {
16.      System.out.println("instantiate a child");
17.    }
18.  }
```

what is the result?

A. `instantiate a child`

B. `instantiate a parent`

C. `instantiate a child`
`instantiate a parent`

D. `instantiate a parent`
`instantiate a child`

E. Compilation fails.

F. An exception is thrown at runtime.

11. Given the following,

```
1.   public class TestPoly {
2.     public static void main(String [] args ){
3.       Parent p = new Child();
4.     }
5.   }
6.
7.   class Parent {
8.     public Parent() {
9.       super();
```

```
10.       System.out.println("instantiate a parent");
11.     }
12.   }
13.
14.   class Child extends Parent {
15.     public Child() {
16.       System.out.println("instantiate a child");
17.       super();
18.     }
19.   }
```

what is the result?

A. `instantiate a child`

B. `instantiate a parent`

C. `instantiate a child`
`instantiate a parent`

D. `instantiate a parent`
`instantiate a child`

E. Compilation fails.

F. An exception is thrown at runtime.

12. Given the following,

```
1.    class MySuper {
2.      public MySuper(int i) {
3.        System.out.println("super " + i);
4.      }
5.    }
6.
7.    public class MySub extends MySuper {
8.      public MySub() {
9.        super(2);
10.       System.out.println("sub");
11.     }
12.
13.     public static void main(String [] args) {
14.       MySuper sup = new MySub();
15.     }
16.   }
```

what is the result?

A. `sub`
`super 2`

B. `super 2`
`sub`

C. Compilation fails at line 2.

D. Compilation fails at line 8.

E. Compilation fails at line 9.

F. Compilation fails at line 14.

13. Given the following,

```
1.   public class ThreeConst {
2.     public static void main(String [] args) {
3.       new ThreeConst(4L);
4.     }
5.     public ThreeConst(int x) {
6.       this();
7.       System.out.print(" " + (x * 2));
8.     }
9.     public ThreeConst(long x) {
10.      this((int) x);
11.      System.out.print(" " + x);
12.    }
13.
14.    public ThreeConst() {
15.      System.out.print("no-arg ");
16.    }
17.  }
```

what is the result?

A. `4`

B. `4 8`

C. `8 4`

D. `8 4 no-arg`

E. `no-arg 8 4`

F. Compilation fails.

14. Given the following,

```
1.   public class ThreeConst {
2.     public static void main(String [] args) {
3.       new ThreeConst();
4.     }
```

```
5.     public void ThreeConst(int x) {
6.       System.out.print(" " + (x * 2));
7.     }
8.     public void ThreeConst(long x) {
9.      System.out.print(" " + x);
10.    }
11.
12.    public void ThreeConst() {
13.      System.out.print("no-arg ");
14.    }
15.  }
```

what is the result?

A. `no-arg`
B. `8 4 no-arg`
C. `no-arg 8 4`
D. Compilation fails.
E. No output is produced.
F. An exception is thrown at runtime.

15. Given the following,

```
1.   class Dog {
2.     Dog(String name) { }
3.   }
```

if class Beagle extends Dog, and class Beagle has only one constructor, which of the following could be the legal constructor for class Beagle?

A. `Beagle() { }`
B. `Beagle() { super(); }`
C. `Beagle() { super("fido"); }`
D. `No constructor, allow the default constructor`

16. Which two of these statements are true about constructors? (Choose two.)

A. Constructors must not have arguments if the superclass constructor does not have arguments.
B. Constructors are not inherited.
C. Constructors cannot be overloaded.
D. The first statement of every constructor is a legal call to the `super()` or `this()` method.

Return Types (Sun Objective 1.4)

17. Given the following,

```
13.   int x;
14.   x = n.test();
18.   int test() {
19.
20.     return y;
21.   }
```

which line of code, inserted at line 19, will not compile?

A. `short y = 7;`

B. `int y = (int) 7.2d;`

C. `Byte y = 7;`

D. `char y = 's';`

E. `int y = 0xface;`

18. Given the following,

```
14.    long test( int x, float y) {
15.
16.    }
```

which two of the following lines, inserted independently, at line 15 would not compile? (Choose two.)

A. `return x;`

B. `return (long) x / y;`

C. `return (long) y;`

D. `return (int) 3.14d;`

E. `return ( y / x );`

F. `return  x / 7;`

19. Given the following,

```
1.   import java.util.*;
2.   class Ro {
3.      public static void main(String [] args) {
4.         Ro r = new Ro();
5.         Object o = r.test();
6.      }
```

```
7.
8.      Object test() {
9.
10.
11.     }
12.  }
```

which two of the following code fragments inserted at lines 9/10 will not compile? (Choose two.)

A. `return null;`

B.
```
Object t = new Object();
return t;
```

C.
```
int [] a = new int [2];
return a;
```

D.
```
char [] [] c = new char [2][2];
return c[0] [1];
```

E.
```
char [] [] c = new char [2][2];
return c[1];
```

F. `return 7;`

20. Given the following,

```
1.   import java.util.*;
2.   class Ro {
3.      public static void main(String [] args) {
4.         Ro r = new Ro();
5.         Object o = r.test();
6.      }
7.
8.      Object test() {
9.
10.
11.     }
12.  }
```

which two of the following code fragments inserted at lines 9/10 will not compile? (Choose two.)

A.
```
char [] [] c = new char [2][2];
return c;
```

B. `return (Object) 7;`

C. `return (Object) (new int [] {1,2,3} );`

D.
```
ArrayList a = new ArrayList();
return a;
```

E. `return (Object) "test";`

F. `return (Float) 4.3;`

21. Given the following,

```
1.   class Test {
2.      public static Foo f = new Foo();
3.      public static Foo f2;
4.      public static Bar b = new Bar();
5.
6.      public static void main(String [] args) {
7.         for (int x=0; x<6; x++) {
8.            f2 = getFoo(x);
9.            f2.react();
10.        }
11.     }
12.     static Foo getFoo(int y) {
13.          if ( 0 == y % 2 ) {
14.            return f;
15.          } else {
16.            return b;
17.          }
18.     }
19.  }
20.
21.  class Bar extends Foo {
22.     void react() { System.out.print("Bar "); }
23.  }
24.
25.  class Foo {
26.     void react() { System.out.print("Foo "); }
27.  }
```

what is the result?

A. `Bar Bar Bar Bar Bar Bar`

B. `Foo Bar Foo Bar Foo Bar`

C. `Foo Foo Foo Foo Foo Foo`

D. Compilation fails.

E. An exception is thrown at runtime.

SELF TEST ANSWERS

Encapsulation, IS-A, HAS-A (Sun Objective 6.1)

1. ☑ **D.** If a class has an instance variable that is marked `public`, the class cannot be said to be encapsulated.
   ☒ **A, B, C, E,** and **F** are incorrect based on the program logic described above. `Public` getter and setter methods are compatible with the concept of encapsulation.

2. ☑ **D.** Class A is clearly not encapsulated because it has a `public` instance variable. At first glance class B appears to be encapsulated, however because it extends from class A it inherits the `public` instance variable *foo*, which is not encapsulated.
   ☒ **A, B,** and **C** are incorrect based on the program logic described above.

3. ☑ **A.** One of the main benefits of encapsulation is that encapsulated code is much easier to reuse than unencapsulated code.
   ☒ **B, C, D,** and **E** are incorrect. **B** is incorrect because inheritance is a concept that is independent of encapsulation. **C** and **D** are incorrect because encapsulation does not restrict the use of overloading or overriding. **E** is incorrect because HAS-A relationships are independent of encapsulation.

4. ☑ **B** and **E.** Encapsulation tends to make code more maintainable, extensible, and debuggable, but not necessarily any more efficient at runtime. Encapsulation is a design approach and in no way affects any Java language rules such as the use of access modifiers.
   ☒ **A, C,** and **D** are well-known benefits of encapsulation.

5. ☑ **C** and **E. C** is correct because class A has an instance variable, *c*, that is a reference to an object of class C. **E** is correct because class B extends from class A, which HAS-A class C reference, so class B, through inheritance, HAS-A class C.
   ☒ **A, B,** and **D** are incorrect based on the program logic described. **A** is incorrect because class B extends from class A, not the other way around. **B** is incorrect because class C is not in class A's inheritance tree. **D** is incorrect because class B IS-A class A; HAS-A is not used to describe inheritance relationships.

Overriding and Overloading (Sun Objective 6.2)

6. ☑ **B.** Reference variable '*a*' is of type `A`, but it refers to an object of type `B`. Line 9 is a polymorphic call, and the VM will use the version of the `baz()` method that is in the class that the reference variable refers to at that point.
   ☒ **A, C,** and **D** are incorrect because of the logic described above.

7. ☑ **B. B** is neither a legal override (the return type has been changed) nor a legal overload (the arguments have not changed).
☒ **A, C,** and **D** are legal overrides of the `doStuff()` method, and **E** and **F** are legal overloads of the `doStuff()` method.

8. ☑ **E.** Line 14 is an illegal override of the `doStuff()` method in ParentClass. When you override a method, you must leave both the arguments and the return types the same.
☒ **A, B, C, D,** and **F** are incorrect based on the program logic described above. If line 14 had returned an *int*, then **B** would be correct.

9. ☑ **C** and **E. C** is an illegal override because the `private` modifier is more restrictive than `doStuff()`'s default modifier in class Over. **E** is an illegal override because you can't change an overridden method's return type, or **E** is an illegal overload because you must change an overloaded method's arguments.
☒ **A** and **B** are simple overrides (`protected` is less restrictive than default). **D** and **F** are simple overloads (swapping arguments of different types creates an overload).

Instantiation and Constructors (Sun Objectives 6.3 and 1.3)

10. ☑ **D.** The class Child constructor calls the class Parent constructor implicitly before any code in the Child constructor runs. When the class Parent constructor's code runs, it prints the first line of output, finishes, and returns control to the Child constructor, which prints out its line of output and finishes. The call to `super()` is redundant.
☒ **A, B, C, E,** and **F** are incorrect based on the program logic described above.

11. ☑ **E.** Line 17 will cause the compiler to fail. The call to `super()` must be the first statement in a constructor.
☒ **A, B, C, D,** and **F** are incorrect based on the program logic described above. If line 17 were removed, **D** would be correct.

12. ☑ **B.** Class MySuper does not need a no-args constructor because MySub explicitly calls the MySuper constructor with an argument.
☒ **A** is incorrect because other than the implicit calls to `super()`, constructors run in order from base class to extended class, so MySuper's output will print first. **C, D, E,** and **F** are incorrect based on the program logic described above.

13. ☑ **E.** The `main()` method calls the *long* constructor which calls the *int* constructor, which calls the no-arg constructor, which runs, then returns to the *int* constructor, which runs, then returns to the *long* constructor, which runs last.
☒ **A, B, C, D,** and **F** are incorrect based on the program logic described above.

14. ☑ **E.** The class elements declared in lines 5, 8, and 12 are badly named methods, not constructors. The default constructor runs with no output, and these methods are never called.
☒ **A, B, C, D,** and **F** are incorrect because of the logic described above.

15. ☑ **C.** Only **C** is correct because the Dog class does not have a no-arg constructor; therefore, you must explicitly make the call to `super()`, passing in a string.
☒ **A, B,** and **D** are incorrect based on the program logic described above.

16. ☑ **B** and **D** are simply stating two rules about constructors.
☒ **A** is wrong because subclass constructors do not have to match the arguments of the superclass constructor. Only the call to `super()` must match. **C** is incorrect because constructors can be and are frequently overloaded.

Return Types (Sun Objective 1.4)

17. ☑ **C.** Byte is a wrapper object, not a primitive.
☒ **A** and **D** are primitives that are shorter than *int* so they are cast implicitly. **B** is a *double* explicitly cast to an *int*. **E** is a valid integer initialized with a hexadecimal value.

18. ☑ **B** and **E. B** won't compile because the *long* cast only applies to *x*, not to the expression *x* / *y*. (We know it's tricky, but so is the test.) **E** won't compile because the result of (*y* / *x*) is a *float*.
☒ **A, C, D,** and **F** all return either *longs* or *ints* (which are automatically cast to *longs*).

19. ☑ **D** and **F. D** is a reference to a *char* primitive that happens to be in an array. **F** returns a primitive, not an object.
☒ **A, B, C,** and **E** all return objects. For **A**, `null` is always a valid object return. For **C**, an array is an object that holds other things (either objects or primitives). For **E**, we are returning an array held in an array, and it's still an object!

20. ☑ **B** and **F** are both attempting to cast a primitive to an object—can't do it.
☒ **A, C, D,** and **E** all return objects. **A** is an array object that holds other arrays. **C** is an array object. **D** is an ArrayList object. **E** is a string cast to an object.

21. ☑ **B.** Line 8 is an example of a polymorphic return type. The VM will determine on a case-by-case basis what class of object `f2` refers to, Bar or Foo. This is only possible because the classes Foo and Bar are in the same inheritance tree.
☒ **A, C, D,** and **E,** are incorrect based on the logic described above.