type), unless you're overriding a method. Barring a covariant return, an overriding method must have the same return type as the overridden method of the superclass. We saw that, although 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 (assuming `Horse` extends `Animal`), a `Horse` reference can be returned when the return type is declared an `Animal`.

We covered constructors in detail, learning that if you don't provide a constructor for your class, the compiler will insert one. The compiler-generated constructor is called the default constructor, and it is always a no-arg constructor with a no-arg call to `super()`. The default constructor will never be generated if even a single constructor exists in your class (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 (although the compiler can insert it for you).

After constructors, we discussed the two kinds of initialization blocks and how and when their code runs.

We looked at `static` methods and variables. `static` members are tied to the class, not an instance, so there is only one copy of any `static` member. A common mistake is to attempt to reference an instance variable from a `static` method. Use the class name with the dot operator to access `static` members.

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.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter.

Encapsulation, IS-A, HAS-A (OCA Objective 6.7)

- ❑ Encapsulation helps hide implementation behind an interface (or API).
- ❑ Encapsulated code has two features:
  - ❑ Instance variables are kept protected (usually with the `private` modifier).
  - ❑ Getter and setter methods provide access to instance variables.
- ❑ IS-A refers to inheritance or implementation.
- ❑ IS-A is expressed with the keyword `extends` or `implements`.
- ❑ IS-A, "inherits from," and "is a subtype of" are all equivalent expressions.
- ❑ HAS-A means an instance of one class "has a" reference to an instance of another class or another instance of the same class.

Inheritance (OCA Objectives 7.1 and 7.3)

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

Polymorphism (OCA Objectives 7.2 and 7.3)

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