✓ TWO-MINUTE DRILL

Remember that in this chapter, when we talk about classes, we're referring to non-inner classes, in other words, *top-level* classes.

Java Features and Benefits (OCA Objective 1.5)

☐ While Java provides many benefits to programmers, for the exam you should remember that Java supports object-oriented programming in general, encapsulation, automatic memory management, a large API (library), built-in security features, multiplatform compatibility, strong typing, multithreading, and distributed computing.

Identifiers (OCA Objective 2.1)

☐ Identifiers can begin with a letter, an underscore, or a currency character.

☐ After the first character, identifiers can also include digits.

☐ Identifiers can be of any length.

Executable Java Files and main() (OCA Objective 1.3)

☐ You can compile and execute Java programs using the command-line programs `javac` and `java`, respectively. Both programs support a variety of command-line options.

☐ The only versions of `main()` methods with special powers are those versions with method signatures equivalent to `public static void main(String[] args)`.

☐ `main()` can be overloaded.

Imports (OCA Objective 1.4)

☐ An `import` statement's only job is to save keystrokes.

☐ You can use an asterisk (`*`) to search through the contents of a single package.

☐ Although referred to as "static imports," the syntax is `import static`….

☐ You can import API classes and/or custom classes.

Source File Declaration Rules (OCA Objective 1.2)

☐ A source code file can have only one `public` class.

☐ If the source file contains a `public` class, the filename must match the `public` class name.

☐ A file can have only one `package` statement, but it can have multiple `imports`.

☐ The `package` statement (if any) must be the first (noncomment) line in a source file.

☐ The `import` statements (if any) must come after the `package` statement (if any) and before the first class declaration.

☐ If there is no `package` statement, `import` statements must be the first (noncomment) statements in the source file.

☐ `package` and `import` statements apply to all classes in the file.

☐ A file can have more than one nonpublic class.

☐ Files with no `public` classes have no naming restrictions.

Class Access Modifiers (OCA Objective 6.4)

- ❑ There are three access modifiers: `public`, `protected`, and `private`.
- ❑ There are four access levels: `public`, `protected`, default, and `private`.
- ❑ Classes can have only `public` or default access.
- ❑ A class with default access can be seen only by classes within the same package.
- ❑ A class with `public` access can be seen by all classes from all packages.
- ❑ Class visibility revolves around whether code in one class can
  - ❑ Create an instance of another class
  - ❑ Extend (or subclass) another class
  - ❑ Access methods and variables of another class

Class Modifiers (Nonaccess) (OCA Objectives 1.2, 7.1, and 7.5)

- ❑ Classes can also be modified with `final`, `abstract`, or `strictfp`.
- ❑ A class cannot be both `final` and `abstract`.
- ❑ A `final` class cannot be subclassed.
- ❑ An `abstract` class cannot be instantiated.
- ❑ A single `abstract` method in a class means the whole class must be `abstract`.
- ❑ An `abstract` class can have both `abstract` and nonabstract methods.
- ❑ The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

Interface Implementation (OCA Objective 7.5)

- ☐ Usually, interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
- ☐ Interfaces can be implemented by any class from any inheritance tree.
- ☐ Usually, an interface is like a 100 percent `abstract` class and is implicitly abstract whether or not you type the `abstract` modifier in the declaration.
- ☐ Usually interfaces have only `abstract` methods.
- ☐ Interface methods are by default `public` and usually `abstract`—explicit declaration of these modifiers is optional.
- ☐ Interfaces can have constants, which are always implicitly `public`, `static`, and `final`.
- ☐ Interface constant declarations of `public`, `static`, and `final` are optional in any combination.
- ☐ As of Java 8, interfaces can have concrete methods declared as either `default` or `static`.

Note: This section uses some concepts that we HAVE NOT yet covered. Don't panic: once you've read through all of the book, this section will make sense as a reference.

- ☐ A legal nonabstract implementing class has the following properties:
  - ☐ It provides concrete implementations for the interface's methods.
  - ☐ It must follow all legal override rules for the methods it implements.
  - ☐ It must not declare any new checked exceptions for an implementation method.
  - ☐ It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
  - ☐ It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
  - ☐ It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (but does not have to declare the exceptions of the interface).
- ☐ A class implementing an interface can itself be `abstract`.
- ☐ An `abstract` implementing class does not have to implement the interface methods (but the first concrete subclass must).
- ☐ A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
- ☐ Interfaces can extend one or more other interfaces.
- ☐ Interfaces cannot extend a class or implement a class or interface.
- ☐ When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

Member Access Modifiers (OCA Objective 6.4)

- ☐ Methods and instance (nonlocal) variables are known as "members."
- ☐ Members can use all four access levels: `public`, `protected`, default, and `private`.
- ☐ Member access comes in two forms:
  - ☐ Code in one class can access a member of another class.
  - ☐ A subclass can inherit a member of its superclass.
- ☐ If a class cannot be accessed, its members cannot be accessed.
- ☐ Determine class visibility before determining member visibility.
- ☐ `public` members can be accessed by all other classes, even in other packages.
- ☐ If a superclass member is `public`, the subclass inherits it—regardless of package.
- ☐ Members accessed without the dot operator (`.`) must belong to the same class.
- ☐ `this.` always refers to the currently executing object.
- ☐ `this.aMethod()` is the same as just invoking `aMethod()`.
- ☐ `private` members can be accessed only by code in the same class.
- ☐ `private` members are not visible to subclasses, so `private` members cannot be inherited.
- ☐ Default and `protected` members differ only when subclasses are involved:
  - ☐ Default members can be accessed only by classes in the same package.
  - ☐ `protected` members can be accessed by other classes in the same package, plus subclasses, regardless of package.
  - ☐ `protected` = package + kids (kids meaning subclasses).
  - ☐ For subclasses outside the package, the `protected` member can be accessed only through inheritance; a subclass outside the package cannot access a `protected` member by using a reference to a superclass instance. (In other words, inheritance is the only mechanism for a subclass outside the package to access a `protected` member of its superclass.)
  - ☐ A `protected` member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass's own subclasses.

Local Variables (OCA Objectives 2.1 and 6.4)

- ☐ Local (method, automatic, or stack) variable declarations cannot have access modifiers.
- ☐ `final` is the only modifier available to local variables.
- ☐ Local variables don't get default values, so they must be initialized before use.

Other Modifiers—Members (OCA Objectives 7.1 and 7.5)

- ❑ `final` methods cannot be overridden in a subclass.
- ❑ `abstract` methods are declared with a signature, a return type, and an optional `throws` clause, but they are not implemented.
- ❑ `abstract` methods end in a semicolon—no curly braces.
- ❑ Three ways to spot a nonabstract method:
  - ❑ The method is not marked `abstract`.
  - ❑ The method has curly braces.
  - ❑ The method **MIGHT** have code between the curly braces.
- ❑ The first nonabstract (concrete) class to extend an `abstract` class must implement all of the `abstract` class's `abstract` methods.
- ❑ The `synchronized` modifier applies only to methods and code blocks.
- ❑ `synchronized` methods can have any access control and can also be marked `final`.
- ❑ `abstract` methods must be implemented by a subclass, so they must be inheritable. For that reason
  - ❑ `abstract` methods cannot be `private`.
  - ❑ `abstract` methods cannot be `final`.
- ❑ The `native` modifier applies only to methods.
- ❑ The `strictfp` modifier applies only to classes and methods.

Methods with var-args (OCA Objective 1.2)

- ❑ Methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.
- ❑ A var-arg parameter is declared with the syntax `type... name`; for instance: `doStuff(int... x) { }`.
- ❑ A var-arg method can have only one var-arg parameter.
- ❑ In methods with normal parameters and a var-arg, the var-arg must come last.

Constructors (OCA Objectives 1.2, and 6.3)

- ❑ Constructors must have the same name as the class
- ❑ Constructors can have arguments, but they cannot have a return type.
- ❑ Constructors can use any access modifier (even `private`!).

Variable Declarations (OCA Objective 2.1)

- ❑ Instance variables can
  - ❑ Have any access control
  - ❑ Be marked `final` or `transient`
- ❑ Instance variables can't be `abstract`, `synchronized`, `native`, or `strictfp`.
- ❑ It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."
- ❑ `final` variables have the following properties:
  - ❑ `final` variables cannot be reassigned once assigned a value.
  - ❑ `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
  - ❑ `final` variables must be initialized before the constructor completes.
- ❑ There is no such thing as a `final` object. An object reference marked `final` does NOT mean the object itself can't change.
- ❑ The `transient` modifier applies only to instance variables.
- ❑ The `volatile` modifier applies only to instance variables.

Array Declarations (OCA Objectives 4.1 and 4.2)

- ❑ Arrays can hold primitives or objects, but the array itself is always an object.
- ❑ When you declare an array, the brackets can be to the left or to the right of the variable name.
- ❑ It is never legal to include the size of an array in the declaration.
- ❑ An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array. For example, if `Horse` extends `Animal`, then a `Horse` object can go into an `Animal` array.

Static Variables and Methods (OCA Objective 6.2)

- ❑ They are not tied to any particular instance of a class.
- ❑ No class instances are needed in order to use `static` members of the class or interface.
- ❑ There is only one copy of a `static` variable/class, and all instances share it.
- ❑ `static` methods do not have direct access to nonstatic members.

enums (OCA Objective 1.2)

- ❑ An `enum` specifies a list of constant values assigned to a type.
- ❑ An `enum` is NOT a `String` or an `int`; an `enum` constant's type is the `enum` type. For example, `SUMMER` and `FALL` are of the `enum` type `Season`.
- ❑ An `enum` can be declared outside or inside a class, but NOT in a method.
- ❑ An `enum` declared outside a class must NOT be marked `static`, `final`, `abstract`, `protected`, or `private`.
- ❑ `enum`s can contain constructors, methods, variables, and constant-specific class bodies.
- ❑ `enum` constants can send arguments to the `enum` constructor, using the syntax `BIG(8)`, where the `int` literal 8 is passed to the `enum` constructor.
- ❑ `enum` constructors can have arguments and can be overloaded.
- ❑ `enum` constructors can NEVER be invoked directly in code. They are always called automatically when an `enum` is initialized.
- ❑ The semicolon at the end of an `enum` declaration is optional. These are legal:
  - ❑ `enum Foo { ONE, TWO, THREE} enum Foo { ONE, TWO, THREE};`
- ❑ `MyEnum.values()` returns an array of `MyEnum`'s values.

✓ TWO-MINUTE DRILL

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

Encapsulation, IS-A, HAS-A* (OCA Objective 6.5)

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

Inheritance (OCA Objective 7.1)

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

Polymorphism (OCA Objective 7.2)

- ☐ **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.**

Overriding and Overloading (OCA Objectives 6.1 and 7.2)

- ☐ **Methods can be overridden or overloaded; constructors can be overloaded but not**

Overriding and Overloading (OCA Objectives 6.1 and 7.2)

- **Methods can be overridden or overloaded; constructors can be overloaded but not overridden.**
- **With respect to the method it overrides, the overriding method**
  - **Must have the same argument list**
  - **Must have the same return type or a subclass (known as a covariant return)**
  - **Must not have a more restrictive access modifier**
  - **May have a less restrictive access modifier**
  - **Must not throw new or broader checked exceptions**
  - **May throw fewer or narrower checked exceptions, or any unchecked exception**
- `final` **methods cannot be overridden.**
- **Only inherited methods may be overridden, and remember that private methods are not inherited.**
- **A subclass uses** `super.overriddenMethodName()` **to call the superclass version of an overridden method.**
- **A subclass uses** `MyInterface.super.overriddenMethodName()` **to call the super interface version on an overridden method.**
- **Overloading means reusing a method name but with different arguments.**
- **Overloaded methods**
  - **Must have different argument lists**
  - **May have different return types, if argument lists are also different**
  - **May have different access modifiers**
  - **May throw different exceptions**
- **Methods from a supertype can be overloaded in a subtype.**
- **Polymorphism applies to overriding, not to overloading.**
- **Object type (not the reference variable's type) determines which overridden method is used at runtime.**
- **Reference type determines which overloaded method will be used at compile time.**

Reference Variable Casting (OCA Objective 7.3)

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

Implementing an Interface (OCA Objective 7.5)

□ When you implement an interface, you are fulfilling its contract.

□ You implement an interface by properly and concretely implementing all the abstract methods defined by the interface.

□ A single class can implement many interfaces.

Return Types (OCA Objectives 7.2 and 7.5)

□ Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.

□ Object reference return types can accept `null` as a return value.

□ An array is a legal return type, both to declare and return as a value.

□ For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.

□ Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return` in any method with a `void` return type to bust out of a method early. But you can't return nothing from a method with a non-`void` return type.

□ Methods with an object reference return type can return a subtype.

□ Methods with an interface return type can return any implementer.

Constructors and Instantiation (OCA Objectives 6.3 and 7.4)

- **A constructor is always invoked when a new object is created.**
- **Each superclass in an object's inheritance tree will have a constructor called.**
- **Every class, even an abstract class, has at least one constructor.**
- **Constructors must have the same name as the class.**
- **Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class; it's not a constructor.**
- **Typical constructor execution occurs as follows:**
  - **The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the `Object` constructor.**
  - **The `Object` constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.**
- **Constructors can use any access modifier (even `private`!).**
- **The compiler will create a default constructor if you don't create any constructors in your class.**
- **The default constructor is a no-arg constructor with a no-arg call to `super()`.**
- **The first statement of every constructor must be a call either to `this()` (an overloaded constructor) or to `super()`.**
- **The compiler will add a call to `super()` unless you have already put in a call to `this()` or `super()`.**
- **Instance members are accessible only after the `super` constructor runs.**
- **`Abstract` classes have constructors that are called when a concrete subclass is instantiated.**
- **Interfaces do not have constructors.**
- **If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.**
- **Constructors are never inherited; thus they cannot be overridden.**
- **A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).**
- **Regarding issues with calls to `this()`:**
  - **They may appear only as the first statement in a constructor.**
  - **The argument list determines which overloaded constructor is called.**
  - **Constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.**
  - **Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.**

Initialization Blocks (OCA Objective 1.2 and 6.3-ish)

☐ **Use `static init` blocks—`static { /* code here */ }`—for code you want to have run once, when the class is first loaded. Multiple blocks run from the top down.**

☐ **Use normal `init` blocks—`{ /* code here }`—for code you want to have run for every new instance, right after all the super constructors have run. Again, multiple blocks run from the top of the class down.**

Statics (OCA Objective 6.2)

☐ **Use `static` methods to implement behaviors that are not affected by the state of any instances.**

☐ **Use `static` variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a `static` variable.**

☐ **All `static` members belong to the class, not to any instance.**

☐ **A `static` method can't access an instance variable directly.**

☐ **Use the dot operator to access `static` members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable; for instance:**

```
d.doStuff();
```

becomes

```
Dog.doStuff();
```

☐ **To invoke an interface's static method use `MyInterface.doStuff()` syntax.**

☐ **`static` methods can't be overridden, but they can be redefined.**

✓ TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Stack and Heap

- ☐ Local variables (method variables) live on the stack.
- ☐ Objects and their instance variables live on the heap.

Literals and Primitive Casting (OCA Objective 2.1)

- ☐ Integer literals can be binary, decimal, octal (such as `013`), or hexadecimal (such as `0x3d`).
- ☐ Literals for `longs` end in `L` or `l`. (For the sake of readability, we recommend "`L`".)
- ☐ Float literals end in `F` or `f`, and `double` literals end in a digit or `D` or `d`.
- ☐ The `boolean` literals are `true` and `false`.
- ☐ Literals for `chars` are a single character inside single quotes: `'d'`.

Scope (OCA Objective 1.1)

- ☐ Scope refers to the lifetime of a variable.
- ☐ There are four basic scopes:
  - ☐ Static variables live basically as long as their class lives.
  - ☐ Instance variables live as long as their object lives.
  - ☐ Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
  - ☐ Block variables (for example, in a `for` or an `if`) live until the block completes.

Basic Assignments (OCA Objectives 2.1, 2.2, and 2.3)

- ☐ Literal integers are implicitly `ints`.
- ☐ Integer expressions always result in an `int`-sized result, never smaller.
- ☐ Floating-point numbers are implicitly doubles (64 bits).
- ☐ Narrowing a primitive truncates the *high order* bits.
- ☐ Compound assignments (such as `+=`) perform an automatic cast.
- ☐ A reference variable holds the bits that are used to refer to an object.
- ☐ Reference variables can refer to subclasses of the declared type but not to superclasses.
- ☐ When you create a new object, such as `Button b = new Button();`, the JVM does three things:
  - ☐ Makes a reference variable named `b`, of type `Button`.
  - ☐ Creates a new `Button` object.
  - ☐ Assigns the `Button` object to the reference variable `b`.

Using a Variable or Array Element That Is Uninitialized and Unassigned (OCA Objectives 4.1 and 4.2)

- ☐ When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of `null`.
- ☐ When an array of primitives is instantiated, elements get default values.
- ☐ Instance variables are always initialized with a default value.
- ☐ Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Passing Variables into Methods (OCA Objective 6.6)

- ☐ Methods can take primitives and/or object references as arguments.
- ☐ Method arguments are always copies.
- ☐ Method arguments are never actual objects (they can be references to objects).
- ☐ A primitive argument is an unattached copy of the original primitive.
- ☐ A reference argument is another copy of a reference to the original object.
- ☐ Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs and hard-to-answer exam questions.

Garbage Collection (OCA Objective 2.4)

- ☐ In Java, garbage collection (GC) provides automated memory management.
- ☐ The purpose of GC is to delete objects that can't be reached.
- ☐ Only the JVM decides when to run the GC; you can only suggest it.
- ☐ You can't know the GC algorithm for sure.
- ☐ Objects must be considered eligible before they can be garbage collected.
- ☐ An object is eligible when no live thread can reach it.
- ☐ To reach an object, you must have a live, reachable reference to that object.
- ☐ Java applications can run out of memory.
- ☐ Islands of objects can be garbage collected, even though they refer to each other.
- ☐ Request garbage collection with `System.gc();`.
- ☐ The `Object` class has a `finalize()` method.
- ☐ The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.
- ☐ The garbage collector makes no guarantees; `finalize()` may never run.
- ☐ You can ineligible-ize an object for GC from within `finalize()`.

✓ TWO-MINUTE DRILL

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

Relational Operators (OCA Objectives 3.1 and 3.2)

- ❑ Relational operators always result in a `boolean` value (`true` or `false`).
- ❑ There are six relational operators: >, >=, <, <=, ==, and `!=`. The last two (== and `!=`) are sometimes referred to as *equality operators*.
- ❑ When comparing characters, Java uses the Unicode value of the character as the numerical value.
- ❑ Equality operators
  - ❑ There are two equality operators: == and `!=`.
  - ❑ Four types of things can be tested: numbers, characters, booleans, and reference variables.
- ❑ When comparing reference variables, == returns `true` only if both references refer to the same object.

instanceof Operator (OCA Objective 3.1)

- ❑ `instanceof` is for reference variables only; it checks whether the object is of a particular type.
- ❑ The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy.
- ❑ For interfaces, an object passes the `instanceof` test if any of its superclasses implement the interface on the right side of the `instanceof` operator.

Arithmetic Operators (OCA Objective 3.1)

- ❑ The four primary math operators are add (+), subtract (–), multiply (`*`), and divide (`/`).
- ❑ The remainder (a.k.a. modulus) operator (`%`) returns the remainder of a division.
- ❑ Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
- ❑ The `*`, `/`, and `%` operators have higher precedence than + and –.

String Concatenation Operator (OCA Objective 3.1)

- ❑ If either operand is a `String`, the + operator concatenates the operands.
- ❑ If both operands are numeric, the + operator adds the operands.

Increment/Decrement Operators (OCA Objective 3.1)

☐ Prefix operators (e.g. `--x`) run before the value is used in the expression.

☐ Postfix operators (e.g., `x++`) run after the value is used in the expression.

☐ In any expression, both operands are fully evaluated *before* the operator is applied.

☐ Variables marked `final` cannot be incremented or decremented.

Ternary (Conditional) Operator (OCA Objective 3.3)

☐ Returns one of two values based on the state of its `boolean` expression.

☐ Returns the value after the `?` if the expression is `true`.

☐ Returns the value after the `:` if the expression is `false`.

Logical Operators (OCA Objective 3.1)

☐ The exam covers six "logical" operators: `&`, `|`, `^`, `!`, `&&`, and `||`.

☐ Work with two expressions (except for `!`) that must resolve to boolean values.

☐ The `&&` and `&` operators return `true` only if both operands are `true`.

☐ The `||` and | operators return `true` if either or both operands are `true`.

☐ The `&&` and `||` operators are known as short-circuit operators.

☐ The `&&` operator does not evaluate the right operand if the left operand is `false`.

☐ The `||` does not evaluate the right operand if the left operand is `true`.

☐ The `&` and `|` operators always evaluate both operands.

☐ The `^` operator (called the "logical XOR") returns `true` if exactly one operand is `true`.

☐ The `!` operator (called the "inversion" operator) returns the opposite value of the boolean operand it precedes.

Parentheses and Operator Precedence (OCA Objective 3.1)

☐ In real life, use parentheses to clarify your code, and force Java to evaluate expressions as intended.

☐ For the exam, memorize Table 4-2 to determine how parentheses-free code will be evaluated.

✓ TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. You might want to loop through them several times.

Writing Code Using if and switch Statements (OCA Objectives 3.3 and 3.4)

- ☐ The only legal expression in an `if` statement is a `boolean` expression—in other words, an expression that resolves to a `boolean` or a `Boolean` reference.
- ☐ Watch out for `boolean` assignments (=) that can be mistaken for `boolean` equality (==) tests:

```
   boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```

- ☐ Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- ☐ `switch` statements can evaluate only to `enums` or the `byte`, `short`, `int`, `char`, and, as of Java 7, `String` data types. You can't say this:

```
   long s = 30;
switch(s) { }
```

- ☐ The `case` constant must be a literal or a compile-time constant, including an `enum` or a `String`. You cannot have a case that includes a nonfinal variable or a range of values.
- ☐ If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.
- ☐ The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- ☐ The `default` block can be located anywhere in the `switch` block, so if no preceding `case` matches, the `default` block will be entered; if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

Writing Code Using Loops (OCA Objectives 5.1, 5.2, 5.3, and 5.4)

☐ A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.

☐ If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop or within the `for` loop declaration.

☐ A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop—in other words, code below the `for` loop won't be able to use the variable.

☐ You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be comma separated.

☐ An enhanced `for` statement (new as of Java 5) has two parts: the *declaration* and the *expression*. It is used only to loop through arrays or collections.

☐ With an enhanced `for`, the *expression* is the array or collection through which you want to loop.

☐ With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.

☐ Unlike with C, you cannot use a number or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.

☐ The `do` loop will **always** enter the body of the loop at least once.

Using break and continue (OCA Objective 5.5)

☐ An unlabeled `break` statement will cause the current iteration of the innermost loop to stop and the line of code following the loop to run.

☐ An unlabeled `continue` statement will cause the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.

☐ If the `break` statement or the `continue` statement is labeled, it will cause a similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (OCA Objectives 8.1, 8.2, 8.3, 8.4, and 8.5)

- ☐ Some of the benefits of Java's exception-handling features include organized error-handling code, easy error detection, keeping exception-handling code separate from other code, and the ability to reuse exception-handling code for a range of issues.
- ☐ Exceptions come in two flavors: checked and unchecked.
- ☐ Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- ☐ Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws` or handle the exception with an appropriate `try/catch`.
- ☐ Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.
- ☐ A `finally` block will always be invoked, regardless of whether an exception is thrown or caught in its `try/catch`.
- ☐ The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- ☐ Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- ☐ Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()` and `main()` is "ducking" the exception by declaring it).
- ☐ You can almost always create your own exceptions by extending `Exception` or one of its checked exception subtypes. Such an exception will then be considered a checked exception by the compiler. (In other words, it's rare to extend `RuntimeException`.)
- ☐ All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes!
- ☐ Some exceptions are created by programmers and some by the JVM.

Using String and StringBuilder (OCA Objectives 9.2 and 9.1)

- ☐ `String` objects are immutable, and `String` reference variables are not.
- ☐ If you create a new `String` without assigning it, it will be lost to your program.
- ☐ If you redirect a `String` reference to a new `String`, the old `String` can be lost.
- ☐ `String` methods use zero-based indexes, except for the second argument of `substring()`.
- ☐ The `String` class is `final`—it cannot be extended.
- ☐ When the JVM finds a `String` literal, it is added to the `String` literal pool.
- ☐ Strings have a *method* called `length()`—arrays have an *attribute* named `length`.
- ☐ `StringBuilder` objects are mutable—they can change without creating a new object.
- ☐ `StringBuilder` methods act on the invoking object, and objects can change without an explicit assignment in the statement.
- ☐ Remember that chained methods are evaluated from left to right.
- ☐ `String` methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- ☐ `StringBuilder` methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

Manipulating Calendar Data (OCA Objective 9.3)

- ☐ On the exam all the objects created using the calendar classes are immutable, but their reference variables are not.
- ☐ If you create a new calendar object without assigning it, it will be lost to your program.
- ☐ If you redirect a calendar reference to a new calendar object, the old calendar object can be lost.
- ☐ All of the objects created using the exam's calendar classes must be created using factory methods (e.g., `from()`, `now()`, `of()`, `parse()`); the keyword `new` is not allowed.
- ☐ The `until() and between()` methods perform complex calculations that determine the amount of time between the values of two calendar objects.
- ☐ The `DateTimeFormatter` class uses the `parse()` method to parse input Strings into valid calendar objects.
- ☐ The `DateTimeFormatter` class uses the `format()` method to format calendar objects into beautifully formed Strings.

Using Arrays (OCA Objectives 4.1 and 4.2)

- ☐ Arrays can hold primitives or objects, but the array itself is always an object.
- ☐ When you declare an array, the brackets can be to the left or right of the name.
- ☐ It is never legal to include the size of an array in the declaration.
- ☐ You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- ☐ Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- ☐ You'll get a `NullPointerException` if you try to use an array element in an object array if that element does not refer to a real object.
- ☐ Arrays are indexed beginning with zero.
- ☐ An `ArrayIndexOutOfBoundsException` occurs if you use a bad index value.
- ☐ Arrays have a `length` attribute whose value is the number of array elements.
- ☐ The last index you can access is always one less than the length of the array.
- ☐ Multidimensional arrays are just arrays of arrays.
- ☐ The dimensions in a multidimensional array can have different lengths.
- ☐ An array of primitives can accept any value that can be promoted implicitly to the array's declared type—for example, a `byte` variable can go in an `int` array.
- ☐ An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array. For example, if `Horse` extends `Animal`, then a `Horse` object can go into an `Animal` array.
- ☐ If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- ☐ You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a `Honda` array can be assigned to an array declared as type `Car` (assuming `Honda` extends `Car`).

Using ArrayList (OCA Objective 9.4)

☐ `ArrayLists` allow you to resize your list and make insertions and deletions to your list far more easily than arrays.

☐ `ArrayLists` are ordered by default. When you use the `add()` method with no index argument, the new entry will be appended to the end of the `ArrayList`.

☐ For the OCA 8 exam, the only `ArrayList` declarations you need to know are of this form:

```
ArrayList<type> myList = new ArrayList<type>();
List<type> myList2 = new ArrayList<type>();  // polymorphic
List<type> myList3 = new ArrayList<>();  // diamond operator, polymorphic
optional
```

☐ `ArrayLists` can hold only objects, not primitives, but remember that autoboxing can make it look like you're adding primitives to an `ArrayList` when, in fact, you're adding a wrapper object version of a primitive.

☐ An `ArrayList`'s index starts at 0.

☐ `ArrayLists` can have duplicate entries. Note: Determining whether two objects are duplicates is trickier than it seems and doesn't come up until the OCP 8 exam.

☐ `ArrayList` methods to remember: `add(element)`, `add(index, element)`, `clear()`, `contains(object)`, `get(index)`, `indexOf(object)`, `remove(index)`, `remove(object)`, and `size()`.

Encapsulating Reference Variables (OCA Objective 6.5)

☐ If you want to encapsulate mutable objects like `StringBuilders` or arrays or `ArrayLists`, you cannot return a reference to these objects; you must first make a copy of the object and return a reference to the copy.

☐ Any class that has a method that returns a reference to a mutable object is breaking encapsulation.

Using Predicate Lambda Expressions (OCA Objective 9.5)

☐ Lambdas allow you to pass bits of code from one method to another. And the receiving method can run whatever complying code it is sent.

☐ While there are many types of lambdas that Java 8 supports, for this exam, the only lambda type you need to know is the `Predicate`.

☐ The `Predicate` interface has a single method to implement that's called `test()`, and it takes one argument and returns a `boolean`.

☐ As the `Predicate.test()` method returns a boolean, it can be placed (mostly?) wherever a boolean expression can go, e.g., in `if`, `while`, `do`, and ternary statements.

☐ `Predicate` lambda expressions have three parts: a single argument, an arrow (`->`), and an expression or code block.

☐ A `Predicate` lambda expression's argument can be just a variable or a type and variable together in parentheses, e.g., `(MyClass m)`.

☐ A `Predicate` lambda expression's body can be an expression that resolves to a boolean, OR it can be a block of statements (surrounded by curly braces) that ends with a boolean-returning `return` statement.