ORACLE®

Oracle Press™

OCA Java® SE 8 Programmer I Exam Guide

(Exam 1Z0-808)

Kathy Sierra
Bert Bates

Mc Graw Hill Education

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

1

Declarations and Access Control

CERTIFICATION OBJECTIVES

- Java Features and Benefits
- Identifiers and Keywords
- javac, java, main(), and Imports
- Declare Classes and Interfaces
- Declare Class Members
- Declare Constructors and Arrays
- Create static Class Members
- Use enums

✓ Two-Minute Drill

Q&A Self Test

```
    OVERWHELMING(16) {              // start a code block that defines
                                    // the "body" for this constant

      public String getLidCode() {  // override the method
                                    // defined in CoffeeSize
        return "A";
      }
    };    // the semicolon is REQUIRED when more code follows

    CoffeeSize(int ounces) {
      this.ounces = ounces;
    }

  private int ounces;

  public int getOunces() {
    return ounces;
  }
  public String getLidCode() {      // this method is overridden
                                    // by the OVERWHELMING constant

    return "B";                     // the default value we want to
                                    // return for CoffeeSize constants
  }
}
```

CERTIFICATION SUMMARY

After absorbing the material in this chapter, you should be familiar with some of the nuances of the Java language. You may also be experiencing confusion around why you ever wanted to take this exam in the first place. That's normal at this point. If you hear yourself asking, "What was I thinking?" just lie down until it passes. We would like to tell you that it gets easier…that this was the toughest chapter and it's all downhill from here.

Let's briefly review what you'll need to know for the exam:

There will be many questions dealing with keywords indirectly, so be sure you can identify which are keywords and which aren't.

You need to understand the rules associated with creating legal identifiers and the rules associated with source code declarations, including the use of `package` and `import` statements.

You learned the basic syntax for the `java` and `javac` command-line

programs.

You learned about when `main()` has superpowers and when it doesn't.

We covered the basics of `import` and `import static` statements. It's tempting to think that there's more to them than saving a bit of typing, but there isn't.

You now have a good understanding of access control as it relates to classes, methods, and variables. You've looked at how access modifiers (`public`, `protected`, and `private`) define the access control of a class or member.

You learned that `abstract` classes can contain both `abstract` and nonabstract methods, but that if even a single method is marked `abstract`, the class must be marked `abstract`. Don't forget that a concrete (nonabstract) subclass of an `abstract` class must provide implementations for all the `abstract` methods of the superclass, but that an `abstract` class does not have to implement the `abstract` methods from its superclass. An `abstract` subclass can "pass the buck" to the first concrete subclass.

We covered interface implementation. Remember that interfaces can extend another interface (even multiple interfaces), and that any class that implements an interface must implement all methods from all the interfaces in the inheritance tree of the interface the class is implementing.

You've also looked at the other modifiers, including `static`, `final`, `abstract`, `synchronized`, and so on. You've learned how some modifiers can never be combined in a declaration, such as mixing `abstract` with either `final` or `private`.

Keep in mind that there are no `final` objects in Java. A reference variable marked `final` can never be changed, but the object it refers to can be modified. You've seen that `final` applied to methods means a subclass can't override them, and when applied to a class, the `final` class can't be subclassed.

Methods can be declared with a var-arg parameter (which can take from zero to many arguments of the declared type), but that you can have only one var-arg per method, and it must be the method's last parameter.

Make sure you're familiar with the relative sizes of the numeric primitives. Remember that while the values of nonfinal variables can change, a reference variable's type can never change.

You also learned that arrays are objects that contain many variables of the same type. Arrays can also contain other arrays.

Remember what you've learned about `static` variables and methods, especially that `static` members are per-class as opposed to per-instance.

Don’t forget that a `static` method can’t directly access an instance variable from the class it’s in because it doesn’t have an explicit reference to any particular instance of the class.

Finally, we covered `enums`. An `enum` is a safe and flexible way to implement constants. Because they are a special kind of class, `enums` can be declared very simply, or they can be quite complex—including such attributes as methods, variables, constructors, and a special type of inner class called a constant specific class body.

Before you hurl yourself at the practice test, spend some time with the following optimistically named “Two-Minute Drill.” Come back to this particular drill often as you work through this book and especially when you’re doing that last-minute cramming. Because—and here’s the advice you wished your mother had given you before you left for college—it’s not what you know, it’s when you know it.

For the exam, knowing what you can’t do with the Java language is just as important as knowing what you can do. Give the sample questions a try! They’re very similar to the difficulty and structure of the real exam questions and should be an eye opener for how difficult the exam can be. Don’t worry if you get a lot of them wrong. If you find a topic that you are weak in, spend more time reviewing and studying. Many programmers need two or three serious passes through a chapter (or an individual objective) before they can answer the questions confidently.

✓ 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`.
- `enums` 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.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself like, "I am smart enough to understand `enums`" and "OK, so that other guy knows `enums` better than I do, but I bet he can't <insert something you *are* good at> like me."

1. Which are true? (Choose all that apply.)

A. "X extends Y" is correct if and only if X is a class and Y is an interface

B. "X extends Y" is correct if and only if X is an interface and Y is a class

C. "X extends Y" is correct if X and Y are either both classes or both interfaces

D. "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces

2. Given:

```
class Rocket {
  private void blastOff() { System.out.print("bang "); }
}
public class Shuttle extends Rocket {
  public static void main(String[] args) {
    new Shuttle().go();
  }
  void go() {
    blastOff();
    // Rocket.blastOff();  // line A
  }
  private void blastOff() { System.out.print("sh-bang "); }
}
```

Which are true? (Choose all that apply.)

A. As the code stands, the output is `bang`

B. As the code stands, the output is `sh-bang`

C. As the code stands, compilation fails

D. If line A is uncommented, the output is `bang bang`

E. If line A is uncommented, the output is `sh-bang bang`

F. If line A is uncommented, compilation fails.

3. Given that the `for` loop's syntax is correct, and given:

```
import static java.lang.System.*;
class _ {
  static public void main(String[] __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )   // for loop
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

4. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.   static Animals a;
8.   public static void main(String[] args) {
9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.  }
11. }
```

What is the result?

A. `woof burble`

B. Multiple compilation errors

C. Compilation fails due to an error on line 2

D. Compilation fails due to an error on line 3

E. Compilation fails due to an error on line 4

F. Compilation fails due to an error on line 9

5. Given two files:

```
1. package pkgA;
2. public class Foo {
3. int a = 5;
4. protected int b = 6;
5. public int c = 7;
6. }
```

```
3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.   public static void main(String[] args) {
7.     Foo f = new Foo();
8.     System.out.print(" " + f.a);
9.     System.out.print(" " + f.b);
10.    System.out.println(" " + f.c);
11.  }
12. }
```

What is the result? (Choose all that apply.)

A. `5 6 7`

B. `5` followed by an exception

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

F. Compilation fails with an error on line 10

6. Given:

```
1. public class Electronic implements Device
     { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
     { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
     { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 1

C. Compilation fails with an error on line 3

D. Compilation fails with an error on line 5

E. Compilation fails with an error on line 7

F. Compilation fails with an error on line 9

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
8.     long [] x [5];
9.     Boolean []ba[];
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 6

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

8. Given:

```
3. public class TestDays {
4.   public enum Days { MON, TUE, WED };
5.   public static void main(String[] args) {
6.     for(Days d : Days.values() )
7.       ;
8.     Days [] d2 = Days.values();
9.     System.out.println(d2[2]);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `TUE`

B. `WED`

C. The output is unpredictable

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 6

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

9. Given:

```
 4. public class Frodo extends Hobbit {
 5.   public static void main(String[] args) {
 6.     int myGold = 7;
 7.     System.out.println(countGold(myGold, 6));
 8.   }
 9. }
10. class Hobbit {
11.   int countGold(int x, int y) { return x + y; }
12. }
```

What is the result?

A. `13`

B. Compilation fails due to multiple errors

C. Compilation fails due to an error on line 6

D. Compilation fails due to an error on line 7

E. Compilation fails due to an error on line 11

10. Given:

```
interface Gadget {
  void doStuff();
}
abstract class Electronic {
  void getPower() { System.out.print("plug in "); }
}
public class Tablet extends Electronic implements Gadget {
  void doStuff() { System.out.print("show book "); }
  public static void main(String[] args) {
    new Tablet().getPower();
    new Tablet().doStuff();
  }
}
```

Which are true? (Choose all that apply.)

A. The class `Tablet` will NOT compile

B. The interface `Gadget` will NOT compile

C. The output will be `plug in show book`

D. The `abstract` class `Electronic` will NOT compile

E. The class `Tablet` CANNOT both extend and implement

11. Given that the Integer class is in the java.lang package and given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `static import java.lang.Integer.*;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

12. Given:

```
interface MyInterface {
  // insert code here
}
```

Which lines of code—inserted independently at `insert code here`—will compile? (Choose all that apply.)

A. `public static m1() {;}`

B. `default void m2() {;}`

C. `abstract int m3();`

D. `final short m4() {return 5;}`

E. `default long m5();`

F. `static void m6() {;}`

13. Which are true? (Choose all that apply.)

A. Java is a dynamically typed programming language

B. Java provides fine-grained control of memory through the use of pointers

C. Java provides programmers the ability to create objects that are well encapsulated

D. Java provides programmers the ability to send Java objects from one machine to another

E. Java is an implementation of the ECMA standard

F. Java's encapsulation capabilities provide its primary security mechanism

SELF TEST ANSWERS

1. ☑ **C** is correct.
☒ **A** is incorrect because classes implement interfaces, they don't extend them. **B** is incorrect because interfaces only "inherit from" other interfaces. **D** is incorrect based on the preceding rules. (OCA Objective 7.5)

2. ☑ **B** and **F** are correct. Since `Rocket.blastOff()` is `private`, it can't be overridden, and it is invisible to class `Shuttle`.
☒ **A, C, D,** and **E** are incorrect based on the above. (OCA Objective 6.4)

3. ☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, `main()`, and pre-incrementing logic. (Note: You might get a compiler warning when compiling this code.)
☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 1.2)

4. ☑ **A** is correct; `enums` can have constructors and variables.
☒ **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax. (OCA Objective 1.2)

5. ☑ **D** and **E** are correct. Variable `a` has default access, so it cannot be accessed from outside the package. Variable `b` has protected access in `pkgA`.

☒ **A, B, C,** and **F** are incorrect based on the above information. (OCA Objectives 1.4 and 6.5)

6. ☑ **A** is correct; all of these are legal declarations.
☒ **B, C, D, E,** and **F** are incorrect based on the above information. (OCA Objective 7.5)

7. ☑ **C** and **D** are correct. Variable names cannot begin with a #, and an array declaration can't include a size without an instantiation. The rest of the code is valid.
☒ **A, B,** and **E** are incorrect based on the above. (OCA Objective 2.1)

8. ☑ **B** is correct. Every `enum` comes with a `static values()` method that returns an array of the `enum`'s values in the order in which they are declared in the `enum`.
☒ **A, C, D, E, F,** and **G** are incorrect based on the above information. (OCP Objective 1.2)

9. ☑ **D** is correct. The `countGold()` method cannot be invoked from a static context.
☒ **A, B, C,** and **E** are incorrect based on the above information. (OCA Objective 6.2)

10. ☑ **A** is correct. By default, an interface's methods are `public` so the `Tablet.doStuff` method must be public, too. The rest of the code is valid.
☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objective 7.5)

11. ☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static imports`, so the code will also compile with none of the imports.
☒ **A, B, D,** and **F** are incorrect based on the above. (OCA Objective 1.4)

12. ☑ **B, C,** and **F** are correct. As of Java 8, interfaces can have `default` and `static` methods.
☒ **A, D,** and **E** are incorrect. **A** has no return type; **D** cannot have a method body; and **E** needs a method body. (OCA Objective 7.5)

13. ☑ **C** and **D** are correct.
☒ **A** is incorrect because Java is a statically typed language. **B** is incorrect because it does not provide pointers. **E** is incorrect because

JavaScript is an implementation of the ECMA standard, not Java. **F** is incorrect because the use of bytecode and the JVM provide Java's primary security mechanisms.

2

Object Orientation

CERTIFICATION OBJECTIVES

- Describe Encapsulation
- Implement Inheritance
- Use IS-A and HAS-A Relationships (OCP)
- Use Polymorphism
- Use Overriding and Overloading
- Understand Casting
- Use Interfaces
- Understand and Use Return Types
- Develop Constructors
- Use static Members

✓ Two-Minute Drill

Q&A Self Test

```
class Animal {
  static void doStuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void doStuff() {           // it's a redefinition,
                                    // not an override
    System.out.print("d ");
  }
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for(int x = 0; x < a.length; x++) {
      a[x].doStuff();               // invoke the static method
    }
    Dog.doStuff();                  // invoke using the class name
  }
}
```

Running this code produces this output:

```
a a a d
```

Remember, the syntax a `[x].doStuff()` is just a shortcut (the syntax trick)—the compiler is going to substitute something like `Animal.doStuff()` instead. Notice also that you can invoke a `static` method by using the class name.

Notice that we didn't use the *enhanced* for *loop* here (covered in Chapter 5), even though we could have. Expect to see a mix of both Java 1.4 and Java 5–8 coding styles and practices on the exam.

CERTIFICATION SUMMARY

We started the chapter by discussing the importance of encapsulation in good OO design, and then we talked about how good encapsulation is implemented: with private instance variables and public getters and setters.

Next, we covered the importance of inheritance, so that you can grasp overriding, overloading, polymorphism, reference casting, return types, and constructors.

We covered IS-A and HAS-A. IS-A is implemented using inheritance, and HAS-A is implemented by using instance variables that refer to other objects.

Polymorphism was next. Although a reference variable's type can't be changed, it can be used to refer to an object whose type is a subtype of its own. We learned how to determine what methods are invocable for a given reference variable.

We looked at the difference between overridden and overloaded methods, learning that an overridden method occurs when a subtype inherits a method from a supertype and then reimplements the method to add more specialized behavior. We learned that, at runtime, the JVM will invoke the subtype version on an instance of a subtype and the supertype version on an instance of the supertype. `Abstract` methods must be "overridden" (technically, `abstract` methods must be implemented, as opposed to overridden, since there really isn't anything to override).

We saw that overriding methods must declare the same argument list and return type or they can return a subtype of the declared return type of the supertype's 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 learned the mechanics of casting (mostly downcasting) reference variables and when it's necessary to do so.

Implementing interfaces came next. An interface describes a *contract* that the implementing class must follow. The rules for implementing an interface are similar to those for extending an `abstract` class. As of Java 8, interfaces can have concrete methods, which are labeled `default`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

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. 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 because constructors do not have return types.

We saw how all 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 or interface, 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 respective class or interface 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 6.5)

- 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 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.

SELF TEST

1. Given:

```
public abstract interface Frobnicate { public void
twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
  public abstract void twiddle(String s) { }
}
```

B.
```
public abstract class Frob implements Frobnicate { }
```

C.
```
public class Frob extends Frobnicate {
  public void twiddle(Integer i) { }
}
```

D.
```
public class Frob implements Frobnicate {
  public void twiddle(Integer i) { }
}
```

E.
```
public class Frob implements Frobnicate {
  public void twiddle(String i) { }
  public void twiddle(Integer s) { }
}
```

2. Given:

```
class Top {
  public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
  public Bottom2(String s) { System.out.print("D"); }
  public static void main(String [] args) {
    new Bottom2("C");
    System.out.println(" ");
  }
}
```

What is the result?

A. `BD`

B. `DB`

C. `BDC`

D. `DBC`

E. Compilation fails

3. Given:

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}
public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet");  }
  public static void main(String [] args) {
    new Clidlet().flipper();
  }
}
```

What is the result?

A. `Clidlet`

B. `Clidder`

C. `Clidder`
`Clidlet`

D. `Clidlet`
`Clidder`

E. Compilation fails

Special Note: The next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its

Java exams, but just in case Oracle's policy changes, we left a few in the book.

4. Using the **fragments** below, complete the following **code** so it compiles. Note that you may not have to fill in all of the slots.

Code:

```
class AgedP {
  _________ _________ _________ _________ _________
  public AgedP(int x) {
    _________ _________ _________ _________ _________
  }
}
public class Kinder extends AgedP {
  _________ _________ _________ _________ _______ _________
  public Kinder(int x) {
    _________ _________ _________ _________ _________ ();
  }
}
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
(	)	{	}
;			

5. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`

B. `pre b2 b1 r2 r3 hawk`

C. `pre b2 b1 r2 r3 hawk r1 r4`

D. `r1 r4 pre b1 b2 r3 r2 hawk`

E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted

I. Compilation fails

Note: You'll probably never see this many choices on the real exam!

6. Given the following:

```
1. class X { void do1() { } }
2. class Y extends X { void do2() { } }
3.
4. class Chrome {
5.   public static void main(String [] args) {
6.     X x1 = new X();
 7.     X x2 = new Y();
 8.     Y y1 = new Y();
 9.     // insert code here
10. } }
```

Which of the following, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`

B. `(Y)x2.do2();`

C. `((Y)x2).do2();`

D. None of the above statements will compile

7. Given:

```
public class Locomotive {
  Locomotive() { main("hi"); }

  public static void main(String[] args) {
    System.out.print("2 ");
  }
  public static void main(String args) {
    System.out.print("3 " + args);
  }
}
```

What is the result? (Choose all that apply.)

A. `2` will be included in the output

B. `3` will be included in the output

C. `hi` will be included in the output

D. Compilation fails

E. An exception is thrown at runtime

8. Given:

```
3. class Dog {
4.   public void bark() { System.out.print("woof "); }
5. }
6. class Hound extends Dog {
7.   public void sniff() { System.out.print("sniff "); }
8.   public void bark() { System.out.print("howl "); }
9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

What is the result? (Choose all that apply.)

A. `howl howl sniff`

B. `howl woof sniff`

C. `howl howl` followed by an exception

D. `howl woof` followed by an exception

E. Compilation fails with an error at line 14

F. Compilation fails with an error at line 15

9. Given:

```
 3. public class Redwood extends Tree {
 4.   public static void main(String[] args) {
 5.     new Redwood().go();
 6.   }
 7.   void go() {
 8.     go2(new Tree(), new Redwood());
 9.     go2((Redwood) new Tree(), new Redwood());
10.   }
11.   void go2(Tree t1, Redwood r1) {
12.      Redwood r2 = (Redwood)t1;
13.      Tree t2 = (Tree)r1;
14.   }
15. }
16. class Tree { }
```

What is the result? (Choose all that apply.)

A. An exception is thrown at runtime

B. The code compiles and runs with no output

C. Compilation fails with an error at line 8

D. Compilation fails with an error at line 9

E. Compilation fails with an error at line 12

F. Compilation fails with an error at line 13

10. Given:

```
 3. public class Tenor extends Singer {
 4.   public static String sing() { return "fa"; }
 5.   public static void main(String[] args) {
 6.     Tenor t = new Tenor();
 7.     Singer s = new Tenor();
 8.     System.out.println(t.sing() + " " + s.sing());
 9.   }
10. }
11. class Singer { public static String sing() { return "la"; } }
```

What is the result?

A. `fa fa`

B. `fa la`

C. `la la`

D. Compilation fails

E. An exception is thrown at runtime

11. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. `subsub`

B. `sub subsub`

C. `alpha subsub`

D. `alpha sub subsub`

E. Compilation fails

F. An exception is thrown at runtime

12. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. `h hn x`

B. `hn x h`

C. `b h hn x`

D. `b hn x h`

E. `bn x h hn x`

F. `b bn x h hn x`

G. `bn x b h hn x`

H. Compilation fails

13. Given:

```
3. class Mammal {
4.   String name = "furry ";
5.   String makeNoise() { return "generic noise"; }
6. }
7. class Zebra extends Mammal {
8.   String name = "stripes ";
9.   String makeNoise() { return "bray"; }
10. }
11. public class ZooKeeper {
12.   public static void main(String[] args) { new ZooKeeper().go(); }
13.   void go() {
14.     Mammal m = new Zebra();
15.     System.out.println(m.name + m.makeNoise());
16.   }
17. }
```

What is the result?

A. `furry bray`

B. `stripes bray`

C. `furry generic noise`

D. `stripes generic noise`

E. Compilation fails

F. An exception is thrown at runtime

14. Given:

```
1. interface FrogBoilable {
2.   static int getCtoF(int cTemp) {
3.     return (cTemp * 9 / 5) + 32;
4.   }
5.   default String hop() { return "hopping "; }
```

```
 6. }
 7. public class DontBoilFrogs implements FrogBoilable {
 8.   public static void main(String[] args) {
 9.     new DontBoilFrogs().go();
10.   }
11.   void go() {
12.     System.out.print(hop());
13.     System.out.println(getCtoF(100));
14.     System.out.println(FrogBoilable.getCtoF(100));
15.     DontBoilFrogs dbf = new DontBoilFrogs();
16.     System.out.println(dbf.getCtoF(100));
17.   }
18. }
```

What is the result? (Choose all that apply.)

A. `hopping 212`

B. Compilation fails due to an error on line 2

C. Compilation fails due to an error on line 5

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 13

F. Compilation fails due to an error on line 14

G. Compilation fails due to an error on line 16

15. Given:

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

What is the result?

A. `1`

B. 2

C. 3

D. The output is unpredictable

E. Compilation fails

F. An exception is thrown at runtime

16. Given:

```
interface MyInterface {
  default int doStuff() {
    return 42;
  }
}
public class IfaceTest implements MyInterface {
  public static void main(String[] args) {
    new IfaceTest().go();
  }
  void go() {
    // INSERT CODE HERE
  }
  public int doStuff() {
    return 43;
  }
}
```

Which line(s) of code, inserted independently at // INSERT CODE HERE, will allow the code to compile? (Choose all that apply.)

A. `System.out.println("class: " + doStuff());`

B. `System.out.println("iface: " + super.doStuff());`

C. `System.out.println("iface: " + MyInterface.super.doStuff());`

D. `System.out.println("iface: " + MyInterface.doStuff());`

E. `System.out.println("iface: " + super.MyInterface.doStuff());`

F. None of the lines, A–E will allow the code to compile

SELF TEST ANSWERS

1. ☑ **B** and **E** are correct. B is correct because an `abstract` class need not implement any or all of an interface's methods. E is correct because the class implements the interface method and additionally

overloads the `twiddle()` method.
☒ **A, C,** and **D** are incorrect. A is incorrect because `abstract` methods have no body. C is incorrect because classes implement interfaces; they don't extend them. D is incorrect because overloading a method is not implementing it. (OCA Objectives 7.1 and 7.5)

2. ☑ **E** is correct. The implied `super()` call in `Bottom2`'s constructor cannot be satisfied because there is no no-arg constructor in `Top`. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.
☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objective 6.3)

3. ☑ **A** is correct. Although a `final` method cannot be overridden, in this case, the method is private and, therefore, hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.
☒ **B, C, D,** and **E** are incorrect based on the preceding. (OCA Objective 7.2)

Special Note: This next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle's policy changes, we left a few in the book.

4. Here is the answer:

```
class AgedP {
  AgedP() {}
  public AgedP(int x) {
  }
}
public class Kinder extends AgedP {
  public Kinder(int x) {
    super();
  }
}
```

As there is no droppable tile for the variable `x` and the parentheses (in the `Kinder` constructor) are already in place and empty, there is

no way to construct a call to the superclass constructor that takes an argument. Therefore, the only remaining possibility is to create a call to the no-arg superclass constructor. This is done as `super();`. The line cannot be left blank, as the parentheses are already in place. Further, since the superclass constructor called is the no-arg version, this constructor must be created. It will not be created by the compiler because another constructor is already present. (OCA Objectives 6.3 and 7.4) Note: As you can see, many questions test for OCA Objective 7.1, we're going to stop mentioning objective 7.1.

5. ☑ **D** is correct. Static `init` blocks are executed at class loading time; instance `init` blocks run right after the call to `super()` in a constructor. When multiple `init` blocks of a single type occur in a class, they run in order, from the top down.
   ☒ **A, B, C, E, F, G, H,** and **I** are incorrect based on the above. Note: You'll probably never see this many choices on the real exam! (OCA Objective 6.3)

6. ☑ **C** is correct. Before you can invoke `Y`'s `do2` method, you have to cast `x2` to be of type `Y`.
   ☒ **A, B,** and **D** are incorrect based on the preceding. B looks like a proper cast, but without the second set of parentheses, the compiler thinks it's an incomplete statement. (OCA Objective 7.3)

7. ☑ **A** is correct. It's legal to overload `main()`. Since no instances of `Locomotive` are created, the constructor does not run and the overloaded version of `main()` does not run.
   ☒ **B, C, D,** and **E** are incorrect based on the preceding. (OCA Objectives 1.3 and 6.3)

8. ☑ **F** is correct. Class `Dog` doesn't have a `sniff` method.
   ☒ **A, B, C, D,** and **E** are incorrect based on the above information. (OCA Objectives 7.2 and 7.3)

9. ☑ **A** is correct. A `ClassCastException` will be thrown when the code attempts to downcast a `Tree` to a `Redwood`.
   ☒ **B, C, D, E,** and **F** are incorrect based on the above information. (OCA Objective 7.3)

10. ☑ **B** is correct. The code is correct, but polymorphism doesn't apply to `static` methods.
    ☒ **A, C, D,** and **E** are incorrect based on the above information.

(OCA Objectives 6.2 and 7.2)

11. ☑ **C** is correct. Watch out, because `SubSubAlpha` extends `Alpha`! Because the code doesn't attempt to make a `SubAlpha`, the private constructor in `SubAlpha` is okay.
☒ **A, B, D, E,** and **F** are incorrect based on the above information. (OCA Objectives 6.3 and 7.2)

12. ☑ **C** is correct. Remember that constructors call their superclass constructors, which execute first, and that constructors can be overloaded.
☒ **A, B, D, E, F, G,** and **H** are incorrect based on the above information. (OCA Objectives 6.3 and 7.4)

13. ☑ **A** is correct. Polymorphism is only for instance methods, not instance variables.
☒ **B, C, D, E,** and **F** are incorrect based on the above information. (OCA Objective 6.3)

14. ☑ **E** and **G** are correct. Neither of these lines of code uses the correct syntax to invoke an interface's static method.
☒ **A, B, C, D,** and **F** are incorrect based on the above information. (OCP Objectives 6.2 and 7.5)

15. ☑ **E** is correct. This is kind of a trick question; the implementing method must be marked `public`. If it was, all the other code is legal, and the output would be 3. If you understood all the multiple inheritance rules and just missed the access modifier, give yourself half credit.
☒ **A, B, C, D,** and **F** are incorrect based on the above information. (OCP Objective 7.5)

16. ☑ **A** and **C** are correct. A uses correct syntax to invoke the class's method, and C uses the correct syntax to invoke the interface's overloaded `default` method.
☒ **B, D, E,** and **F** are incorrect. (OCP Objective 7.5)

3

Assignments

CERTIFICATION OBJECTIVES

- Use Class Members
- Understand Primitive Casting
- Understand Variable Scope
- Differentiate Between Primitive Variables and Reference Variables
- Determine the Effects of Passing Variables into Methods
- Understand Object Lifecycle and Garbage Collection

✓ Two-Minute Drill

Q&A Self Test

Stack and Heap—Quick Review

CERTIFICATION SUMMARY

This chapter covered a wide range of topics. Don't worry if you have to review some of these topics as you get into later chapters. This chapter includes a lot of foundational stuff that will come into play later.

We started the chapter by reviewing the stack and the heap; remember that local variables live on the stack and instance variables live with their objects on the heap.

We reviewed legal literals for primitives and `Strings`, and then we discussed the basics of assigning values to primitives and reference variables and the rules for casting primitives.

Next we discussed the concept of scope, or "How long will this variable live?" Remember the four basic scopes in order of lessening life span: static, instance, local, and block.

We covered the implications of using uninitialized variables and the importance of the fact that local variables MUST be assigned a value explicitly. We talked about some of the tricky aspects of assigning one reference variable to another and some of the finer points of passing variables into methods, including a discussion of "shadowing."

Finally, we dove into garbage collection, Java's automatic memory management feature. We learned that the heap is where objects live and where all the cool garbage collection activity takes place. We learned that in the end, the JVM will perform garbage collection whenever it wants to. You (the programmer) can request a garbage collection run, but you can't force it. We talked about garbage collection only applying to objects that are eligible, and that eligible means "inaccessible from any live thread." Finally, we discussed the rarely useful `finalize()` method and what you'll have to know about it for the exam. All in all, this was one fascinating chapter.

✓ 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()`.

SELF TEST

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// do Stuff` is reached, how many objects are eligible for garbage collection?

A. 0

B. 1

C. 2

D. Compilation fails

E. It is not possible to know

F. An exception is thrown at runtime

2. Given:

```
public class Fishing {
  byte b1 = 4;
  int i1 = 123456;
  long L1 = (long)i1;       // line A
  short s2 = (short)i1;     // line B
  byte b2 = (byte)i1;       // line C
  int i2 = (int)123.456;    // line D
  byte b3 = b1 + 7;         // line E
}
```

Which lines WILL NOT compile? (Choose all that apply.)

A. Line A

B. Line B

C. Line C

D. Line D

E. Line E

3. Given:

```
public class Literally {
  public static void main(String[] args) {
    int i1 = 1_000;       // line A
    int i2 = 10_00;       // line B
    int i3 = _10_000;     // line C
    int i4 = 0b101010;    // line D
    int i5 = 0B10_1010;   // line E
    int i6 = 0x2_a;       // line F
  }
}
```

Which lines WILL NOT compile? (Choose all that apply.)

A. Line A

B. Line B

C. Line C

D. Line D

E. Line E

F. Line F

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails

E. `hi`, followed by an exception

F. `hi hi`, followed by an exception

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails

F. An exception is thrown at runtime

6. Given:

```
public class Mirror {
  int size = 7;
  public static void main(String[] args) {
    Mirror m1 = new Mirror();
    Mirror m2 = m1;
    int i1 = 10;
    int i2 = i1;
    go(m2, i2);
    System.out.println(m1.size + " " + i1);
  }
  static void go(Mirror m, int i) {
    m.size = 8;
    i = 12;
  }
}
```

What is the result?

A. `7 10`

B. `8 10`

C. `7 12`

D. `8 12`

E. Compilation fails

F. An exception is thrown at runtime

7. Given:

```
public class Wind {
  int id;
  Wind(int i) { id = i; }
  public static void main(String[] args) {
    new Wind(3).go();
    // commented line
  }
  void go() {
    Wind w1 = new Wind(1);
    Wind w2 = new Wind(2);
    System.out.println(w1.id + " " + w2.id);
  }
}
```

When execution reaches the commented line, which are true? (Choose all that apply.)

A. The output contains `1`

B. The output contains `2`

C. The output contains `3`

D. Zero `Wind` objects are eligible for garbage collection

E. One `Wind` object is eligible for garbage collection

F. Two `Wind` objects are eligible for garbage collection

G. Three `Wind` objects are eligible for garbage collection

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. `5 7`

B. `5 8`

C. `8 7`

D. `8 8`

E. Compilation fails

F. An exception is thrown at runtime

9. Given:

```
public class Happy {
  int id;
  Happy(int i) { id = i; }
  public static void main(String[] args) {
    Happy h1 = new Happy(1);
    Happy h2 = h1.go(h1);
    System.out.println(h2.id);
  }
  Happy go(Happy h) {
    Happy h3 = h;
    h3.id = 2;
    h1.id = 3;
    return h1;
  }
}
```

What is the result?

A. `1`

B. `2`

C. `3`

D. Compilation fails

E. An exception is thrown at runtime

10. Given:

```
public class Network {
  Network(int x, Network n) {
    id = x;
    p = this;
    if(n != null) p = n;
  }
  int id;
  Network p;
  public static void main(String[] args) {
    Network n1 = new Network(1, null);
    n1.go(n1);
  }
  void go(Network n1) {
    Network n2 = new Network(2, n1);
    Network n3 = new Network(3, n2);
    System.out.println(n3.p.p.id);
  }
}
```

What is the result?

A. `1`

B. `2`

C. `3`

D. `null`

E. Compilation fails

11. Given:

```
3. class Beta { }
4. class Alpha {
5.   static Beta b1;
6.   Beta b2;
7. }
8. public class Tester {
9.   public static void main(String[] args) {
10.    Beta b1 = new Beta();     Beta b2 = new Beta();
11.    Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.    a1.b1 = b1;
13.    a1.b2 = b1;
14.    a2.b2 = b2;
15.    a1 = null;  b1 = null;  b2 = null;
16.    // do stuff
17.  }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

12. Given:

```
public class Telescope {
  static int magnify = 2;
  public static void main(String[] args) {
    go();
  }
  static void go() {
    int magnify = 3;
    zoomIn();
  }
  static void zoomIn() {
    magnify *= 5;
    zoomMore(magnify);
    System.out.println(magnify);
  }
  static void zoomMore(int magnify) {
    magnify *= 7;
  }
}
```

What is the result?

A. `2`

B. `10`

C. `15`

D. `30`

E. `70`

F. `105`

G. Compilation fails

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. ☑ **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated `Short` wrapper object that is also eligible.
   ☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.4)

2. ☑ **E** is correct; compilation of line E fails. When a mathematical operation is performed on any primitives smaller than `int`s, the result is automatically cast to an integer.
   ☒ **A, B, C,** and **D** are all legal primitive casts. (OCA Objective 2.1)

3. ☑ **C** is correct; line **C** will NOT compile. As of Java 7, underscores can be included in numeric literals, but not at the beginning or the end.
   ☒ **A, B, D, E,** and **F** are incorrect. **A** and **B** are legal numeric literals. **D** and **E** are examples of valid binary literals, which were new to

Java 7, and **F** is a valid hexadecimal literal that uses an underscore. (OCA Objective 2.1)

4. ☑ **F** is correct. The `m2` object's `m1` instance variable is never initialized, so when `m5` tries to use it, a `NullPointerException` is thrown.
☒ **A, B, C, D,** and **E** are incorrect based on the above. (OCA Objectives 2.1 and 2.3)

5. ☑ **A** is correct. The references `f1`, `z`, and `f3` all refer to the same instance of `Fizz`. The `final` modifier assures that a reference variable cannot be referred to a different object, but `final` doesn't keep the object's state from changing.
☒ **B, C, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.2)

6. ☑ **B** is correct. In the `go()` method, `m` refers to the single `Mirror` instance, but the `int i` is a new `int` variable, a detached copy of `i2`.
☒ **A, C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 2.2 and 2.3)

7. ☑ **A**, **B,** and **G** are correct. The constructor sets the value of `id` for `w1` and `w2`. When the commented line is reached, none of the three `Wind` objects can be accessed, so they are eligible to be garbage collected.
☒ **C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 1.1, 2.3, and 2.4)

8. ☑ **E** is correct. The parameter declared on line 9 is valid (although ugly), but the variable name `ouch` cannot be declared again on line 11 in the same scope as the declaration on line 9.
☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 1.1 and 2.1)

9. ☑ **D** is correct. Inside the `go()` method, `h1` is out of scope.
☒ **A, B, C,** and **E** are incorrect based on the above. (OCA Objectives 1.1 and 6.1)

10. ☑ **A** is correct. Three `Network` objects are created. The `n2` object has a reference to the `n1` object, and the `n3` object has a reference to the `n2` object. The S.O.P. can be read as, "Use the `n3` object's `Network` reference (the first `p`), to find that object's reference (`n2`), and use that object's reference (the second `p`) to find that object's (`n1`'s) `id`, and print that `id`."

☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objectives, 2.2, 2.3, and 6.4)

11. ☑ **B** is correct. It should be clear that there is still a reference to the object referred to by `a2`, and that there is still a reference to the object referred to by `a2.b2`. What might be less clear is that you can still access the other `Beta` object through the static variable `a2.b1`—because it's static.
☒ **A, C, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.4)

12. ☑ **B** is correct. In the `Telescope` class, there are three different variables named `magnify`. The `go()` method's version and the `zoomMore()` method's version are not used in the `zoomIn()` method. The `zoomIn()` method multiplies the class variable `* 5`. The result (`10`) is sent to `zoomMore()`, but what happens in `zoomMore()` stays in `zoomMore()`. The S.O.P. prints the value of `zoomIn()`'s `magnify`.
☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 1.1 and 6.6)

13. ☑ **E** is correct. In `go1()` the local variable `x` is not initialized.
☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 2.1 and 2.3)

4

Operators

CERTIFICATION OBJECTIVES

- Using Java Operators
- Use Parentheses to Override Operator Precedence
- Test Equality Between Strings and Other Objects Using == and equals( )

✓ Two-Minute Drill

Q&A Self Test

If you've got variables, you're going to modify them. (Unless you're one of those new-fangled "FP" programmers.) You'll increment them, add them together, and compare one to another (in about a dozen different ways). In this chapter, you'll learn how to do all that in Java. As an added bonus, you'll learn how to do things that you'll

rules!

There are three important general rules for determining how Java will evaluate expressions with operators:

- When two operators of the same precedence are in the same expression, Java evaluates the expression from left to right.
- When parts of an expression are placed in parentheses, those parts are evaluated first.
- When parentheses are nested, the innermost parentheses are evaluated first.

A good way to burn these precedence rules into your brain is to—as always—write some test code and play around with it. We've added an example of some test code that demonstrates several of the precedence hierarchy rules listed here. As you can see, we often compared parentheses-free expressions with their parentheses-rich counterparts to prove the rules:

```
System.out.println((-7 - 4) + " " + (-(7 - 4)));        // unary (-7), beats minus
                                                        // output: -11 -3

System.out.println((2 + 3 * 4) + " " + ((2 + 3) * 4));   // * beats +
                                                         // output: 14 20

System.out.println(7 > 5 && 2 > 3);                      // > beats &&
                                                         // output: false

System.out.print((true & false == false | true) + " ");  // == beats & System.out.
print(((true & false) == (false | true)));    // output: true
```

And to repeat, the output is:

```
-11 -3
14 20
false
true false
```

We're so sorry that you need to memorize this stuff, but if you master what's in this short section, you should be able to handle whatever weird precedence-related questions the exam throws at you.

CERTIFICATION SUMMARY

If you've studied this chapter diligently, you should have a firm grasp on Java operators, and you should understand what equality means when tested with the == operator. Let's review the highlights of what you've learned in this chapter.

The logical operators (`&&`, `||`, `&`, `|`, and `^`) can be used only to evaluate two `boolean` expressions. The difference between `&&` and `&` is that the `&&` operator won't bother testing the right operand if the left evaluates to `false`, because the result of the `&&` expression can never be `true`. The difference between `||` and `|` is that the `||` operator won't bother testing the right operand if the left evaluates to `true` because the result is already known to be `true` at that point.

The == operator can be used to compare values of primitives, but it can also be used to determine whether two reference variables refer to the same object.

The `instanceof` operator is used to determine whether the object referred to by a reference variable passes the IS-A test for a specified type.

The + operator is overloaded to perform `String` concatenation tasks and can also concatenate `String`s and primitives, but be careful—concatenation can be tricky.

The conditional operator (a.k.a. the "ternary operator") has an unusual, three-operand syntax—don't mistake it for a complex assert statement.

The ++ and `--` operators will be used throughout the exam, and you must pay attention to whether they are prefixed or postfixed to the variable being updated.

Even though you should use parentheses in real life, for the exam you should memorize Table 4-2 so you can determine how code that doesn't use parentheses for complex expressions will be evaluated, based on Java's operator-precedence hierarchy.

Be prepared for a lot of exam questions involving the topics from this chapter. Even within questions testing your knowledge of another objective, the code will frequently use operators, assignments, object and primitive passing, and so on.

✓ 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.

SELF TEST

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    int i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`

B. `life`

C. `universe`

D. `everything`

E. Compilation fails

F. An exception is thrown at runtime

2. Given:

```
public class Dog {
  String name;
  Dog(String s) { name = s; }
  public static void main(String[] args) {
    Dog d1 = new Dog("Boi");
    Dog d2 = new Dog("Tyri");
    System.out.print((d1 == d2) + " ");
    Dog d3 = new Dog("Boi");
    d2 = d1;
    System.out.print((d1 == d2) + " ");
    System.out.print((d1 == d3) + " ");
  }
}
```

What is the result?

A. `true true true`

B. `true true false`

C. `false true false`

D. `false true true`

E. `false false false`

F. An exception will be thrown at runtime

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`

B. `production live2`

C. `test case live2`

D. Compilation fails

E. An exception is thrown at runtime

4. Given:

```
class Feline {
  public static void main(String[] args) {
    long x = 42L;
    long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. Compilation fails

5. **Note**: Here's another old-style drag-and-drop question…just in case.

Place the fragments into the code to produce the output `33`. Note that you must use each fragment exactly once.

```
CODE:
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

    System.out.println(x);
  }
}
```

FRAGMENTS:

```
y    y    y    y

y    x    x

-=   *=   *=   *=
```

6. Given:

```
public class Cowboys {
  public static void main(String[] args) {
    int x = 12;
    int a = 5;
    int b = 7;
    System.out.println(x/a + " " + x/b);
  }
}
```

What is the result? (Choose all that apply.)

A. `2 1`

B. `2 2`

C. `3 1`

D. `3 2`

E. An exception is thrown at runtime

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
12.  }
13.  enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors

D. Compilation fails due only to an error on line 7

E. Compilation fails due only to an error on line 8

F. Compilation fails due only to an error on line 11

G. Compilation fails due only to an error on line 13

8. Given:

```
4. public class SpecialOps {
5.   public static void main(String[] args) {
6.     String s = "";
7.     boolean b1 = true;
8.     boolean b2 = false;
9.     if((b2 = false) | (21%5) > 2) s += "x";
10.    if(b1 || (b2 = true))         s += "y";
11.    if(b2 == true)                s += "z";
12.    System.out.println(s);
13.  }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. `x` will be included in the output

C. `y` will be included in the output

D. `z` will be included in the output

E. An exception is thrown at runtime

9. Given:

```
3. public class Spock {
4.   public static void main(String[] args) {
5.     int mask = 0;
6.     int count = 0;
7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )   mask = mask + 1;
8.     if(  (6 > 8) ^ false)                           mask = mask + 10;
9.     if( !(mask > 1) && ++count > 1)                 mask = mask + 100;
10.    System.out.println(mask + " " + count);
11.  }
12. }
```

Which two are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

10. Given:

```
3. interface Vessel { }
4. interface Toy { }
5. class Boat implements Vessel { }
6. class Speedboat extends Boat implements Toy { }
7. public class Tree {
8.   public static void main(String[] args) {
9.     String s = "0";
10.    Boat b = new Boat();
11.    Boat b2 = new Speedboat();
12.    Speedboat s2 = new Speedboat();
13.    if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.    if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.    System.out.println(s);
16.  }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

11. Given:

```
10. boolean b1 = false;
11. boolean b2;
12. int x = 2, y = 5;
13. b1 = 2-12/4 > 5+-7 && b1 != y++>5 == 7%4 > ++x | b1 == true;
14. b2 = (2-12/4 > 5+-7) && (b1 != y++>5) == (7%4 > ++x) | (b1 == true);
15. System.out.println(b1 + " " + b2);
```

What is the result? (This is a tricky one. If you want a hint, go take another look at the operator precedence rant in the chapter.)

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. ☑ **D** is correct. This is a ternary nested in a ternary. Both ternary expressions are false.
☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 3.1 and 3.3)

2. ☑ **C** is correct. The == operator tests for reference variable equality, not object equality.
☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

3. ☑ **E** is correct. Because the short-circuit (`||`) is not used, both operands are evaluated. Since `args[1]` is past the `args` array bounds, an `ArrayIndexOutOfBoundsException` is thrown.
   ☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

4. ☑ **G** is correct. Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers, they are added together.
   ☒ **A, B, C, D, E, F, H,** and **I** are incorrect based on the above. (OCA Objective 3.1)

5. Answer:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x *= x;
    y *= y;
    y *= y;
    x -= y;

    System.out.println(x);
  }
}
```

   Yeah, we know it's kind of puzzle-y, but you might encounter something like it on the real exam if Oracle reinstates this type of question. (OCA Objective 3.1)

6. ☑ **A** is correct. When dividing `int`s, remainders are always rounded down.
   ☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objective 3.1)

7. ☑ **A** is correct. All this syntax is correct. The for-each iterates through the `enum` using the `values()` method to return an array. An `enum` can be compared using either `equals()` or `==`. An `enum` can be used in a ternary operator's boolean test.
   ☒ **B**, **C**, **D**, **E**, **F**, and **G** are incorrect based on the above. (OCA Objectives 3.1, 3.2, and 3.3)

8. ☑ **C** is correct. Line 9 uses the modulus operator, which returns the

remainder of the division, which in this case is `1`. Also, line 9 sets `b2` to `false`, and it doesn't test `b2`'s value. Line 10 would set `b2` to `true`; however, the short-circuit operator keeps the expression `b2 = true` from being executed.
☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objectives 3.1, and 3.2)

9. ☑ **C** and **F** are correct. At line 7 the `||` keeps `count` from being incremented, but the `|` allows `mask` to be incremented. At line 8 the `^` returns true only if exactly one operand is true. At line 9 `mask` is `2` and the `&&` keeps `count` from being incremented.
☒ **A, B, D, E,** and **G** are incorrect based on the above. (OCA Objective 3.1)

10. ☑ **D** is correct. First, remember that `instanceof` can look up through multiple levels of an inheritance tree. Also remember that `instanceof` is commonly used before attempting a downcast; so in this case, after line 15, it would be possible to say `Speedboat s3 = (Speedboat)b2;`.
☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 3.1)

11. ☑ **A** is correct. We're pretty sure you won't encounter anything as horrible as this on the real exam. But if you got this one correct, pat yourself on the back! The way to tackle a problem like this is to evaluate the expression in stages. In this case you might solve it like so:

Stage 1: resolve any use of unary operators
Stage 2: resolve any use of multiplication-related operators
Stage 3: handle addition and subtraction
Stage 4: handle any relationship operators
Stage 5: deal with the equality operators
Stage 6: deal with the logical operators
Stage 7: do the short-circuit operators
Stage 8: finally, do the assignment operators

☑ **B**, **C, D, E,** and **F** are incorrect based on the above. (OCA Objective 3.1)

5

Flow Control and Exceptions

CERTIFICATION OBJECTIVES

- Use if and switch Statements
- Develop for, do, and while Loops
- Use break and continue Statements
- Use try, catch, and finally Statements
- State the Effects of Exceptions
- Recognize Common Exceptions

✓ Two-Minute Drill

Q&A Self Test

an you imagine trying to write code using a language that didn’t give you a way to execute statements conditionally? Flow control is a key part of

CERTIFICATION SUMMARY

This chapter covered a lot of ground, all of which involved ways of controlling your program flow based on a conditional test. First, you learned about `if` and `switch` statements. The `if` statement evaluates one or more expressions to a `boolean` result. If the result is `true`, the program will execute the code in the block that is encompassed by the `if`. If an `else` statement is used and the `if` expression evaluates to `false`, then the code following the `else` will be performed. If no `else` block is defined, then none of the code associated with the `if` statement will execute.

You also learned that the `switch` statement can be used to replace multiple
`if-else` statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`); or it can evaluate `enums`; and as of Java 7, it can evaluate `Strings`. At runtime, the JVM will try to find a match between the expression in the `switch` statement and a constant in a corresponding `case` statement. If a match is found, execution will begin at the matching case and continue on from there, executing code in all the remaining `case` statements until a `break` statement is found or the end of the `switch` statement occurs. If there is no match, then the `default` case will execute, if there is one.

You've learned about the three looping constructs available in the Java language. These constructs are the `for` loop (including the basic `for` and the enhanced `for`, which was new to Java 5), the `while` loop, and the `do` loop. In general, the `for` loop is used when you know how many times you need to go through the loop. The `while` loop is used when you do not know how many times you want to go through, whereas the `do` loop is used when you need to go through at least once. In the `for` loop and the `while` loop, the expression has to evaluate to `true` to get inside the block and will check after every iteration of the loop. The `do` loop does not check the condition until after it has gone through the loop once. The major benefit of the `for` loop is the ability to initialize one or more variables and increment or decrement those variables in the `for` loop definition.

The `break` and `continue` statements can be used in either a labeled or unlabeled fashion. When unlabeled, the `break` statement will force the program to stop processing the innermost looping construct and start with the line of code following the loop. Using an unlabeled `continue` command will cause the program to stop execution of the current iteration

of the innermost loop and proceed with the next iteration. When a `break` or a `continue` statement is used in a labeled manner, it will perform in the same way, with one exception: the statement will not apply to the innermost loop; instead, it will apply to the loop with the label. The `break` statement is used most often in conjunction with the `switch` statement. When there is a match between the `switch` expression and the `case` constant, the code following the `case` constant will be performed. To stop execution, a `break` is needed.

You've seen how Java provides an elegant mechanism in exception handling. Exception handling allows you to isolate your error-correction code into separate blocks so the main code doesn't become cluttered by error-checking code. Another elegant feature allows you to handle similar errors with a single error-handling block, without code duplication. Also, the error handling can be deferred to methods further back on the call stack.

You learned that Java's `try` keyword is used to specify a guarded region—a block of code in which problems might be detected. An exception handler is the code that is executed when an exception occurs. The handler is defined by using Java's `catch` keyword. All `catch` clauses must immediately follow the related `try` block.

Java also provides the `finally` keyword. This is used to define a block of code that is always executed, either immediately after a `catch` clause completes or immediately after the associated `try` block in the case that no exception was thrown (or there was a `try` but no `catch`). Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. A `finally` block is not required, but if there is one, it must immediately follow the last `catch`. (If there is no `catch` block, the `finally` block must immediately follow the `try` block.) It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

An exception object is an instance of class `Exception` or one of its subclasses. The `catch` clause takes, as a parameter, an instance of an object of a type derived from the `Exception` class. Java requires that each method either catches any checked exception it can throw or else declares that it throws the exception. The exception declaration is part of the method's signature. To declare that an exception may be thrown, the `throws` keyword is used in a method definition, along with a list of all checked exceptions that might be thrown.

Runtime exceptions are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they do not need to be handled or declared, and thus are known as "unchecked" exceptions.

Errors are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they do not need to be handled or declared. Checked exceptions include any exception types that are not of type `RuntimeException` or `Error`. If your code fails either to handle a checked exception or declare that it is thrown, your code won't compile. But with unchecked exceptions or objects of type `Error`, it doesn't matter to the compiler whether you declare them or handle them, do nothing about them, or do some combination of declaring and handling. In other words, you're free to declare them and handle them, but the compiler won't care one way or the other. It's not good practice to handle an `Error`, though, because you can rarely recover from one.

Finally, remember that exceptions can be generated by the JVM or by a programmer.

✓ 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.

SELF TEST

1. Given that `toLowerCase()` is an aptly named `String` method that returns a `String`, and given the code:

```
public class Flipper {
  public static void main(String[] args) {
    String o = "-";
    switch("RED".toLowerCase()) {
      case "yellow":
        o += "y";
      case "red":
        o += "r";
      case "green":
        o += "g";
    }
    System.out.println(o);
  }
}
```

What is the result?

A. -

B. -r

C. -rg

D. Compilation fails

E. An exception is thrown at runtime

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. -

B. `-c`

C. `-c2`

D. `-2c`

E. `-c22b`

F. `-2c2b`

G. `-2c2bc`

H. Compilation fails

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`

B. `IllegalStateException`

C. `NumberFormatException`

D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

4. Given:

```
public class Flip2 {
  public static void main(String[] args) {
    String o = "-";
    String[] sa = new String[4];
    for(int i = 0; i < args.length; i++)
      sa[i] = args[i];
    for(String n: sa) {
      switch(n.toLowerCase()) {
        case "yellow": o += "y";
        case "red":    o += "r";
        case "green":  o += "g";
      }
    }
    System.out.print(o);
  }
}
```

And given the command-line invocation:

```
Java Flip2 RED Green YeLLow
```

Which are true? (Choose all that apply.)

A. The string `rgy` will appear somewhere in the output
B. The string `rgg` will appear somewhere in the output
C. The string `gyr` will appear somewhere in the output
D. Compilation fails
E. An exception is thrown at runtime

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7.   }
8. }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`
B. `for(x : int y) {`
C. `int y = 0; for(y : x) {`
D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`
E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`
F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

7. Given:

```
3. class SubException extends Exception { }
4. class SubSubException extends SubException { }
5.
6. public class CC { void doStuff() throws SubException { } }
7.
8. class CC2 extends CC { void doStuff() throws SubSubException { } }
9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

8. Given:

```
3. public class Ebb {
4.   static int x = 7;
5.   public static void main(String[] args) {
6.     String s = "";
7.     for(int y = 0; y < 3; y++) {
8.       x++;
9.       switch(x) {
10.         case 8: s += "8 ";
11.         case 9: s += "9 ";
12.         case 10: { s+= "10 "; break; }
13.         default: s += "d ";
```

```
14.          case 13: s+= "13 ";
15.        }
16.      }
17.      System.out.println(s);
18.    }
19.    static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

9. Given:

```
3. class Infinity { }
4. public class Beyond extends Infinity {
5.   static Integer i;
6.   public static void main(String[] args) {
7.     int sw = (int)(Math.random() * 3);
8.     switch(sw) {
9.       case 0: {  for(int x = 10; x > 5; x++)
10.                   if(x > 10000000) x = 10;
11.                break; }
12.      case 1: {  int y = 7 * i;  break;  }
13.      case 2: {  Infinity inf = new Beyond();
14.                 Beyond b = (Beyond)inf;  }
15.    }
16.  }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A `ClassCastException` might be thrown

C. A `StackOverflowError` might be thrown

D. A `NullPointerException` might be thrown

E. An `IllegalStateException` might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

10. Given:

```
3. public class Circles {
4.   public static void main(String[] args) {
5.     int[] ia = {1,3,5,7,9};
6.     for(int x : ia) {
7.       for(int j = 0; j < 3; j++) {
8.         if(x > 4 && x < 8) continue;
9.         System.out.print(" " + x);
10.        if(j == 1) break;
11.        continue;
12.      }
13.      continue;
14.    }
15.  }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

11. Given:

```
3. public class OverAndOver {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     try {
7.       s += "1";
8.       throw new Exception();
9.     } catch (Exception e) { s += "2";
10.    } finally { s += "3"; doStuff(); s += "4";
11.    }
12.    System.out.println(s);
13.  }
14.  static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

12. Given:

```
3. public class Wind {
4.   public static void main(String[] args) {
5.     foreach:
6.     for(int j=0; j<5; j++) {
7.       for(int k=0; k< 3; k++) {
8.         System.out.print(" " + j);
9.         if(j==3 && k==1) break foreach;
10.        if(j==0 || j==2) break;
11.      }
12.    }
13.  }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

13. Given:

```
3. public class Gotcha {
4.   public static void main(String[] args) {
5.     // insert code here
6.
7.   }
8.   void go() {
9.     go();
10.  }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();

II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I–III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile
B. They will all compile
C. All will complete normally
D. None will complete normally
E. Only one will complete normally
F. Two of them will complete normally

14. Given the code snippet:

```
String s = "bob";
String[] sa = {"a", "bob"};
final String s2 = "bob";
StringBuilder sb = new StringBuilder("bob");

// switch(sa[1]) {           // line 1
// switch("b" + "ob") {      // line 2
// switch(sb.toString()) {   // line 3

// case "ann":  ;            // line 4
// case s:      ;            // line 5
// case s2:     ;            // line 6
}
```

And given that the numbered lines will all be tested by uncommenting one `switch` statement and one `case` statement together, which line(s) will FAIL to compile? (Choose all that apply.)

A. line 1
B. line 2
C. line 3
D. line 4
E. line 5
F. line 6

G. All six lines of code will compile

15. Given that `IOException` is in the `java.io` package and given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 2, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a `try/catch` block around line 4 will cause compilation to fail

16. Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I–IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

SELF TEST ANSWERS

1. ☑ **C** is correct. As of Java 7 it's legal to switch on a String, and remember that switches use "entry point" logic.

☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objective 3.4)

2. ☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

3. ☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (that is, a broader exception).

☒ **A, B, E,** and **F** are not in `NumberFormatException`'s class hierarchy. (OCA Objective 8.5)

4. ☑ **E** is correct. As of Java 7 the syntax is legal. The `sa[]` array receives only three arguments from the command line, so on the last iteration through `sa[]`, a `NullPointerException` is thrown.

☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 1.3, 5.2, and 8.5)

5. ☑ **A**, **D,** and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.

   ☒ **B, C,** and **E** are incorrect. **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (OCA Objective 5.2)

6. ☑ **E** is correct. There is no problem nesting `try/catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, and then the code in the `finally` block runs.

   ☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

7. ☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

   ☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

8. ☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-through" logic and that fall-through logic also applies to the default case, which is used when no other case matches.

   ☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (OCA Objective 3.4)

9. ☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw a `NullPointerException`. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.

   ☒ **A, B, C, E,** and **G** are incorrect based on the above. (OCA Objectives 3.4 and 8.5)

10. ☑ **D** is correct. The basic rule for unlabeled `continue` statements is that the current iteration stops early and execution jumps to the next iteration. The last two `continue` statements are redundant!

    ☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

11. ☑ **H** is correct. It's true that the value of `String s` is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not*

guaranteed to complete, and in this case `finally()` never completes, so the `System.out.println` (S.O.P) never executes.

☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 8.2 and 8.5)

12. ☑ **C** is correct. A `break` breaks out of the current innermost loop and carries on. A labeled `break` breaks out of and terminates the labeled loops.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

13. ☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a `StackOverflowError`. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.

☒ **A, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.1, 8.2, and 8.4)

14. ☑ **E** is correct. A `switch`'s `cases` must be compile-time constants or `enum` values.

☒ **A, B, C, D, F,** and **G** are incorrect based on the above. (OCA Objective 3.4)

15. ☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an `IOException`, you have to import the java.io package or declare the exception with a fully qualified name.

☒ **A, B, C,** and **E** are incorrect. **A, B,** and **C** are incorrect based on the above. **E** is incorrect because it's okay both to handle and declare an exception. (OCA Objectives 8.2 and 8.5)

16. ☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However, an overriding method *can* throw `RuntimeExceptions` not thrown by the overridden method.

☒ **A, B, E,** and **F** are incorrect based on the above. (OCA Objective 8.1)

6

Strings, Arrays, ArrayLists, Dates, and Lambdas

CERTIFICATION OBJECTIVES

- Create and Manipulate Strings
- Manipulate Data Using the StringBuilder Class and Its Methods
- Create and Use Calendar Data
- Declare, Instantiate, Initialize, and Use a One-Dimensional Array
- Declare, Instantiate, Initialize, and Use a Multidimensional Array
- Declare and Use an ArrayList
- Use Wrapper Classes
- Use Encapsulation for Reference Variables
- Use Simple Lambda Expressions

✓ Two-Minute Drill

Q&A Self Test

```
// ==== LEGAL LAMBDAS ======================

    m1.go(x -> 7 < 5);                                  // extra terse
    m1.go(x -> { return adder(2, 1) > 5; });            // block
    m1.go((Lamb2 x) -> { int y = 5;
                         return adder(y, 7) > 8; }); // multi-stmt block
    m1.go(x -> { int y=5; return adder(y,6) > 8; }); // no arg type, block
    int a = 5; int b = 6;
    m1.go(x -> { return adder(a, b) > 8; });            // in scope vars
    m1.go((Lamb2 x) -> adder(a, b) > 13);               // arg type, no block

// ==== ILLEGAL LAMBDAS ======================

    // m1.go(x ->   return adder(2, 1) > 5;  );       // return w/o block
    // m1.go(Lamb2 x -> adder(2, 3) > 7);             // type needs parens
    // m1.go(() -> adder(2, 3) > 7);                  // Predicate needs 1 arg
    // m1.go(x -> { adder(4, 2) > 9 });               // blocks need statements
    // m1.go(x -> { int y = 5; adder(y, 7) > 8; });   // block needs return
  }
  void go(Predicate<Lamb2> e) {                        // go() takes a predicate
    Lamb2 m2 = new Lamb2();
    System.out.println(e.test(m2) ? "ternary true"    // ternary uses boolean expr
                                  : "ternary false");
  }
  static int adder(int x, int y) { return x + y; }    // complex calculation
 }
```

This code is mostly about valid and invalid syntax, but let's look a little more closely at the `go()` method. The test is mainly concerned with the code to be passed to a method, but it's useful to look (but not TOO closely) at a method that receives lambda code. In both the Dogs code and the code directly above, the receiving method took a `Predicate`. Inside the receiving methods, we created an object of the type we're working with, which we pass to the `Predicate.test()` method. The receiving method expects the `test()` method to return a `boolean`.

We have to admit that lambdas are a bit tricky to learn. Again, we expect we've left you with some unanswered questions, but we think Oracle did a reasonable job of slicing out a piece of the lambda puzzle to start with. If you understand the bits we've covered, you should be able to handle the lambda-related questions Oracle throws you.

CERTIFICATION SUMMARY

The most important thing to remember about `Strings` is that `String`

objects are immutable, but references to `String`s are not! You can make a new `String` by using an existing `String` as a starting point, but if you don't assign a reference variable to the new `String`, it will be lost to your program—you will have no way to access your new `String`. Review the important methods in the `String` class.

The `StringBuilder` class was added in Java 5. It has exactly the same methods as the old `StringBuffer` class, except `StringBuilder`'s methods aren't thread-safe. Because `StringBuilder`'s methods are not thread-safe, they tend to run faster than `StringBuffer` methods, so choose `StringBuilder` whenever threading is not an issue. Both `StringBuffer` and `StringBuilder` objects can have their value changed over and over without your having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable `String` objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

Next we discussed key classes and interfaces in the new Java 8 calendar and time-related packages. Similar to `String`s, all of the calendar classes we studied create immutable objects. In addition, these classes use factory methods exclusively to create new objects. The keyword `new` cannot be used with these classes. We looked at some of the powerful features of these classes, like calculating the amount of time between two different dates or times. Then we took a look at how the `DateTimeFormatter` class is used to parse `String`s into calendar objects and how it is used to beautify calendar objects.

The next topic was arrays. We talked about declaring, constructing, and initializing one-dimensional and multidimensional arrays. We talked about anonymous arrays and the fact that arrays of objects are actually arrays of references to objects.

Next, we discussed the basics of `ArrayList`s. `ArrayList`s are like arrays with superpowers that allow them to grow and shrink dynamically and to make it easy for you to insert and delete elements at locations of your choosing within the list. We discussed the idea that `ArrayList`s cannot hold primitives, and that if you want to make an `ArrayList` filled with a given type of primitive values, you use "wrapper" classes to turn a primitive value into an object that represents that value. Then we discussed how with autoboxing, turning primitives into wrapper objects, and vice versa, is done automatically.

Finally, we discussed a specific subset of the topic of lambdas, using the `Predicate` interface. The basic idea of lambdas is that you can pass a

bit of code from one method to another. The `Predicate` interface is one of many "functional interfaces" provided in the Java 8 API. A functional interface is one that has only one method to be implemented. In the case of the `Predicate` interface, this method is called `test()`, and it takes a single argument and returns a `boolean`. To wrap up our discussion of lambdas, we covered some of the tricky syntax rules you need to know to write valid lambdas.

✓ TWO-MINUTE DRILL

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

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.

SELF TEST

1. Given:

```
public class Mutant {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("abc");
    String s = "abc";
    sb.reverse().append("d");
    s.toUpperCase().concat("d");
    System.out.println("." + sb + ". ." + s + ".");
  }
}
```

Which two substrings will be included in the result? (Choose two.)

A. .abc.
B. .ABCd.
C. .ABCD.
D. .cbad.
E. .dcba.

2. Given:

```
public class Hilltop {
  public static void main(String[] args) {
    String[] horses = new String[5];
    horses[4] = null;
    for(int i = 0; i < horses.length; i++) {
      if(i < args.length)
        horses[i] = args[i];
      System.out.print(horses[i].toUpperCase() + " ");
    }
  }
}
```

And, if the code compiles, the command line:

```
java Hilltop eyra vafi draumur kara
```

What is the result?

A. `EYRA VAFI DRAUMUR KARA`

B. `EYRA VAFI DRAUMUR KARA null`

C. An exception is thrown with no other output

D. `EYRA VAFI DRAUMUR KARA`, and then a `NullPointerException`

E. `EYRA VAFI DRAUMUR KARA`, and then an `ArrayIndexOutOfBoundsException`

F. Compilation fails

3. Given:

```
public class Actors {
  public static void main(String[] args) {
    char[] ca = {0x4e, \u004e, 78};
    System.out.println((ca[0] == ca[1]) + " " + (ca[0] == ca[2]));
  }
}
```

What is the result?

A. `true true`

B. `true false`

C. `false true`

D. `false false`

E. Compilation fails

4. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result? (Choose all that apply.)

A. `2`

B. `4`

C. An exception is thrown at runtime

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 5

F. Compilation fails due to an error on line 6

G. Compilation fails due to an error on line 7

5. Given:

```
import java.util.*;
public class Sequence {
  public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("apple");
    myList.add("carrot");
    myList.add("banana");
    myList.add(1, "plum");
    System.out.print(myList);
  }
}
```

What is the result?

A. `[apple, banana, carrot, plum]`

B. `[apple, plum, carrot, banana]`

C. `[apple, plum, banana, carrot]`

D. `[plum, banana, carrot, apple]`

E. `[plum, apple, carrot, banana]`

F. `[banana, plum, carrot, apple]`

G. Compilation fails

6. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
8.     Dozens [] da = new Dozens[3];
9.     da[0] = new Dozens();
10.    Dozens d = new Dozens();
11.    da[1] = d;
12.    d = null;
13.    da[1] = null;
14.    // do stuff
15.  }
16. }
```

Which two are true about the objects created within `main()`, and which are eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

7. Given:

```
public class Tailor {
  public static void main(String[] args) {
    byte[][] ba = {{1,2,3,4}, {1,2,3}};
    System.out.println(ba[1].length + " " + ba.length);
  }
}
```

What is the result?

A. `2 4`

B. `2 7`

C. `3 2`

D. `3 7`

E. `4 2`

F. `4 7`

G. Compilation fails

8. Given:

```
3. public class Theory {
4.   public static void main(String[] args) {
5.     String s1 = "abc";
6.     String s2 = s1;
7.     s1 += "d";
8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
9.
10.    StringBuilder sb1 = new StringBuilder("abc");
11.    StringBuilder sb2 = sb1;
12.    sb1.append("d");
13.    System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The first line of output is `abc abc true`

C. The first line of output is `abc abc false`

D. The first line of output is `abcd abc false`

E. The second line of output is `abcd abc false`

F. The second line of output is `abcd abcd true`

G. The second line of output is `abcd abcd false`

9. Given:

```
public class Mounds {
 public static void main(String[] args) {
   StringBuilder sb = new StringBuilder();
   String s = new String();
   for(int i = 0; i < 1000; i++) {
     s = " " + i;
     sb.append(s);
   }
   // done with loop
 }
}
```

If the garbage collector does NOT run while this code is executing, approximately how many objects will exist in memory when the loop is done?

A. Less than 10

B. About 1000

C. About 2000

D. About 3000

E. About 4000

10. Given:

```
3. class Box {
4.   int size;
5.   Box(int s) { size = s; }
6. }
7. public class Laser {
8.   public static void main(String[] args) {
9.     Box b1 = new Box(5);
10.    Box[] ba = go(b1, new Box(6));
11.    ba[0] = b1;
12.    for(Box b : ba) System.out.print(b.size + " ");
13.  }
14.  static Box[] go(Box b1, Box b2) {
15.     b1.size = 4;
16.     Box[] ma = {b2, b1};
17.     return ma;
18.  }
19. }
```

What is the result?

A. 4 4

B. 5 4

C. 6 4

D. 4 5

E. 5 5

F. Compilation fails

11. Given:

```
public class Hedges {
  public static void main(String[] args) {
    String s = "JAVA";
    s = s + "rocks";
    s = s.substring(4,8);
    s.toUpperCase();
    System.out.println(s);
  }
}
```

What is the result?

A. JAVA

B. `JAVAROCKS`

C. `rocks`

D. `rock`

E. `ROCKS`

F. `ROCK`

G. Compilation fails

12. Given:

```
1. import java.util.*;
2. class Fortress {
3.   private String name;
4.   private ArrayList<Integer> list;
5.   Fortress() { list = new ArrayList<Integer>(); }
6.
7.   String getName() { return name; }
8.   void addToList(int x) { list.add(x); }
9.   ArrayList getList() { return list; }
10. }
```

Which lines of code (if any) break encapsulation? (Choose all that apply.)

A. Line 3

B. Line 4

C. Line 5

D. Line 7

E. Line 8

F. Line 9

G. The class is already well encapsulated

13. Given:

```
import java.util.function.Predicate;
public class Sheep {
  public static void main(String[] args) {
    Sheep s = new Sheep();
    s.go(() -> adder(5, 1) < 7);    // line A
    s.go(x -> adder(6, 2) < 9);     // line B
    s.go(x, y -> adder(3, 2) < 4);  // line C
  }
  void go(Predicate<Sheep> e) {
    Sheep s2 = new Sheep();
    if(e.test(s2))
      System.out.print("true ");
    else
      System.out.print("false ");
  }
  static int adder(int x, int y) {
    return x + y;
  }
}
```

What is the result?

A. `true true false`

B. Compilation fails due only to an error at line A

C. Compilation fails due only to an error at line B

D. Compilation fails due only to an error at line C

E. Compilation fails due only to errors at lines A and B

F. Compilation fails due only to errors at lines A and C

G. Compilation fails due only to errors at lines A, B, and C

H. Compilation fails for reasons not listed

14. Given:

```
import java.time.*;
import java.time.format.*;
public class Shiny {
  public static void main(String[] args) {
    DateTimeFormatter f1 =
              DateTimeFormatter.ofPattern("MMM dd, yyyy");
    LocalDate d = LocalDate.of(2018, Month.JANUARY, 15);
    LocalDate d2 = d.plusDays(1);
    System.out.print(f1.format(d) + " ");
    System.out.println(d2.format(f1));
  }
}
```

What is the result?

A. `2018-01-15 2018-01-15`

B. `2018-01-15 2018-01-16`

C. `Jan 15, 2018 Jan 15, 2018`

D. `Jan 15, 2018 Jan 16, 2018`

E. Compilation fails

F. An exception is thrown at runtime

15. Given:

```
import java.util.*;
public class Jackets {
  public static void main(String[] args) {
    List<Integer> myList = new ArrayList<>();  // line 5
    myList.add(new Integer(5));
    myList.add(42);                            // line 7
    myList.add("113");                         // line 8
    myList.add(new Integer("7"));              // line 9
    System.out.println(myList);
  }
}
```

What is the result?

A. `[5, 42, 113, 7]`

B. Compilation fails due only to an error on line 5

C. Compilation fails due only to an error on line 8

D. Compilation fails due only to errors on lines 5 and 8

E. Compilation fails due only to errors on lines 7 and 8

F. Compilation fails due only to errors on lines 5, 7, and 8

G. Compilation fails due only to errors on lines 5, 7, 8, and 9

16. Given that `adder()` returns an `int`, which are valid `Predicate` lambdas? (Choose all that apply.)

A.
```
x, y -> 7 < 5
```
B.
```
x -> { return adder(2, 1) > 5; }
```
C.
```
x -> return adder(2, 1) > 5;
```
D.
```
x -> { int y = 5;
       int z = 7;
       adder(y, z) > 8; }
```
E.
```
x -> { int y = 5;
       int z = 7;
       return adder(y, z) > 8; }
```
F.
```
(MyClass x) -> 7 > 13
```
G.
```
(MyClass x) -> 5 + 4
```

17. Given:

```
import java.util.*;
public class Baking {
  public static void main(String[] args) {
    ArrayList<String> steps = new ArrayList<String>();
    steps.add("knead");
    steps.add("oil pan");
    steps.add("turn on oven");
    steps.add("roll");
    steps.add("turn on oven");
    steps.add("bake");
    System.out.println(steps);
  }
}
```

What is the result?

A. `[knead, oil pan, roll, turn on oven, bake]`

B. `[knead, oil pan, turn on oven, roll, bake]`

C. `[knead, oil pan, turn on oven, roll, turn on oven, bake]`

D. The output is unpredictable

E. Compilation fails

F. An exception is thrown at runtime

18. Given:

```
import java.time.*;
public class Bachelor {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(2018, 8, 15);
    d = d.plusDays(1);
    LocalDate d2 = d.plusDays(1);
    LocalDate d3 = d2;
    d2 = d2.plusDays(1);
    System.out.println(d + " " + d2 + " " + d3);  // line X
  }
}
```

Which are true? (Choose all that apply.)

A. The output is: `2018-08-16 2018-08-17 2018-08-18`

B. The output is: `2018-08-16 2018-08-18 2018-08-17`

C. The output is: `2018-08-16 2018-08-17 2018-08-17`

D. At line X, zero `LocalDate` objects are eligible for garbage collection

E. At line X, one `LocalDate` object is eligible for garbage collection

F. At line X, two `LocalDate` objects are eligible for garbage collection

G. Compilation fails

19. Given that `e` refers to an object that implements `Predicate`, which could be valid code snippets or statements? (Choose all that apply.)

A. `if(e.test(m))`

B. `switch (e.test(m))`

C. `while(e.test(m))`

D. `e.test(m) ? "yes" : "no";`

E. `do {} while(e.test(m));`

F. `System.out.print(e.test(m));`

G. `boolean b = e.test(m));`

SELF TEST ANSWERS

1. ☑ **A** and **D** are correct. The `String` operations are working on a new (lost) `String` not `String s`. The `StringBuilder` operations work from left to right.

☒ **B**, **C**, and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

2. ☑ **D** is correct. The `horses` array's first four elements contain `Strings`, but the fifth is null, so the `toUpperCase()` invocation for the fifth element throws a `NullPointerException`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objectives 4.1 and 1.3)

3. ☑ **E** is correct. The Unicode declaration must be enclosed in single quotes: `'\u004e'`. If this were done, the answer would be **A**, but that equality isn't on the OCA exam.

☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 2.1 and 4.1)

4. ☑ **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]`, not an `int[]`. If line 7 were removed, the output would be 4.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 4.2)

5. ☑ **B** is correct. `ArrayList` elements are automatically inserted in the order of entry; they are not automatically sorted. `ArrayLists` use zero-based indexes, and the last `add()` inserts a new element and shifts the remaining elements back.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 9.4)

6. ☑ **C** and **F** are correct. `da` refers to an object of type "`Dozens` array," and each `Dozens` object that is created comes with its own "`int` array" object. When line 14 is reached, only the second `Dozens` object (and its "`int` array" object) are not reachable.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (OCA Objectives 4.1 and 2.4)

7. ☑ **C** is correct. A two-dimensional array is an "array of arrays." The length of `ba` is 2 because it contains 2 one-dimensional arrays. Array indexes are zero-based, so `ba[1]` refers to `ba`'s second array.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 4.2)

8. ☑ **D** and **F** are correct. Although `String` objects are immutable, references to `Strings` are mutable. The code `s1 += "d";` creates a new `String` object. `StringBuilder` objects are mutable, so the `append()` is changing the single `StringBuilder` object to which both `StringBuilder` references refer.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

9. ☑ **B** is correct. `StringBuilders` are mutable, so all of the `append()` invocations are acting on the same `StringBuilder` object over and over. `Strings`, however, are immutable, so every `String` concatenation operation results in a new `String` object. Also, the string `" "` is created once and reused in every loop iteration.

☒ **A, C, D,** and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

10. ☑ **A** is correct. Although `main()`'s `b1` is a different reference variable than `go()`'s `b1`, they refer to the same `Box` object.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 4.1, 6.1, and 6.6)

11. ☑ **D** is correct. The `substring()` invocation uses a zero-based index and the second argument is exclusive, so the character at index 8 is NOT included. The `toUpperCase()` invocation makes a new `String` object that is instantly lost. The `toUpperCase()` invocation does NOT affect the `String` referred to by `s`.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (OCA Objective 9.2)

12. ☑ **F** is correct. When encapsulating a mutable object like an `ArrayList`, your getter must return a reference to a copy of the object, not just the reference to the original object.

☒ **A, B, C, D, E,** and **G** are incorrect based on the above. (OCA Objective 6.5)

13. ☑ **F** is correct. Predicate lambdas take exactly one parameter; the rest of the code is correct.

☒ **A, B, C, D, E, G,** and **H** are incorrect based on the above. (OCA Objective 9.5)

14. ☑ **D** is correct. Invoking the `plusDays()` method creates a new object, and both `LocalDate` and `DateTimeFormatter` have `format()` methods.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 9.3)

15. ☑ **C** is correct. The only error in this code is attempting to add a `String` to an `ArrayList` of `Integer` wrapper objects. Line 7 uses autoboxing, and lines 6 and 9 demonstrate using a wrapper class's two constructors.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 2.5 and 9.4)

16. ☑ **B, E** and **F** use correct syntax.

☒ **A, C, D**, and **G** are incorrect. **A** passes two parameters. **C**, a `return`, must be in a code block, and code blocks must be in curly braces. **D**, a block, must have a `return` statement. **G,** the result, is not a `boolean`. (OCA Objective 9.5)

17. ☑ **C** is correct. `ArrayLists` can have duplicate entries.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objective 9.4)

18. ☑ **B** and **E** are correct. A total of four `LocalDate` objects are created, but the one created using the `of()` method is abandoned on the next line of code when its reference variable is assigned to the new `LocalDate` object created via the first `plusDays()` invocation. The reference variables are swapped a bit, which accounts for the dates not printing in chronological order.

☒ **A, C, D, F,** and **G** are incorrect based on the above. (OCA Objectives 2.4 and 9.3)

19. ☑ **A, C, D, E, F,** and **G** are correct; they all require a `boolean`.

☒ **B** is incorrect. A `switch` doesn't take a `boolean`. (OCA Objective 9.5)

A

About the Download

This e-book comes with free downloadable Oracle Press Practice Exam Software, which can be downloaded using the links provided in this appendix. This software is easy to install on any Mac or Windows computer and must be installed to access the Practice Exam feature.

System Requirements

Windows

- 2.33GHz or faster x86-compatible processor, or Intel Atom™ 1.6GHz or faster processor for netbook class devices
- Microsoft® Windows Server 2008, Windows 7, Windows 8.1 Classic or Windows 10
- 512MB of RAM (1GB recommended)

Mac OS

- Intel® Core™ Duo 1.83GHz or faster processor
- Mac OS X v10.7 and above
- 512MB of RAM (1GB recommended)

Downloading from McGraw-Hill Professional's Media Center

To download the glossary, additional content, and Oracle Press Practice Exam Software, visit McGraw-Hill Professional's Media Center by clicking the link below and entering this e-book's ISBN and your e-mail address. You will then receive an e-mail message with a download link for the additional content.

http://mhprofessional.com/mediacenter

This e-book's ISBN is 1260011380.

Once you've received the e-mail message from McGraw-Hill Professional's Media Center, click the link included to download the practice exams. If you do not receive the e-mail, be sure to check your spam folder.

Installing the Practice Exam Software

Follow the instructions below for Windows or Mac OS.

Windows

Step 1 Open the InstallerforPC.zip file. You will need to unzip the file and extract or copy and paste the contents to your hard drive.
Step 2 Locate the Installer.exe file and double click the file. After a few moments, the installer will open.
Step 3 Follow the onscreen instructions to install the application.

Mac OS

Step 1 Open the InstallerforMac.zip file. You will need to unzip the file and extract or copy and paste the contents to your hard drive.
Step 2 After a few moments, the contents of the .zip file will be displayed.

Step 3 Double click on Installer to begin installation.
Step 4 Follow the onscreen instructions to install the application.

NOTE: If you get an error while installing the software please ensure your anti-virus or internet security programs are disabled and try installing the software again. You may enable the antivirus or internet security program again after installation is complete.

Running the Practice Exam Software

Follow the instructions below after you have completed the software installation.

Windows

After installing, you can start the application using *either* of the two methods below:

1. Double-click the Oracle Press Java SE 8 Practice icon on your desktop, or
2. Go to the Start menu and click Programs or All Programs. Click Oracle Press Java SE 8 Practice to start the application.

Mac OS

Open the Oracle Press Java SE 8 Practice folder inside your Mac's application folder and double-click the Oracle Press Java SE 8 Practice icon to run the application.

Practice Exam Software Features

The Practice Exam Software provides you with a simulation of the actual exam. The software also features a custom mode that can be used to generate quizzes by exam objective domain. Quiz mode is the default mode. To launch an exam simulation, select one of the OCA exam buttons at the top of the screen, or check the Exam Mode check box at the bottom of the screen and select the OCA exam in the custom window.

The number of questions, types of questions, and the time allowed on the exam simulation are intended to be a representation of the live exam.

The custom exam mode includes hints and references, and in-depth answer explanations are provided through the Feedback feature.

When you launch the software, a digital clock display will appear in the upper-right corner of the question window. The clock will continue to count unless you choose to end the exam by selecting Grade The Exam.

Removing Installation

The Practice Exam Software is installed on your hard drive. For best results for removal of programs using a Windows PC use the Control Panel | Uninstall A Program option and then choose Oracle Press Java SE 8 Practice to uninstall.

For best results for removal of programs using a Mac go to the Oracle Press Java SE 8 Practice folder inside your applications folder and drag the "Oracle Press Java SE 8 Practice" icon to the trash.

Help

A help file is provided through the Help button on the main page in the top-right corner. A readme file is also included in the Bonus Content folder.

Technical Support

Technical Support information is provided in the following sections by feature.

Windows 8 Troubleshooting

The following known errors on Windows 8 have been reported. Please see below for information on troubleshooting these known issues.

If you get an error while installing the software, such as "The application could not be installed because the installer file is damaged. Try obtaining the new installer from the application author," you may need to disable your anti-virus or Internet security programs and try installing the software again. You may enable the antivirus or Internet security program again after installation is complete.

For more information on how to disable anti-virus programs in Windows, please visit the web site of the software provider of your anti-virus program. For example, if you use Norton or MacAfee products, you may need to visit the Norton or the MacAfee web site and search for "how to disable antivirus in Windows 8." Anti-virus programs are different from firewall technology, so be sure to disable the anti-virus program, and be sure to re-enable to program after you have installed the practice exam software.

While Windows doesn't include default antivirus software, Windows can often detect antivirus software installed by you or the manufacturer of your computer and typically displays the status of any such software in the Action Center, which is located in the Control Panel under System and Security (select Review Your Computer's Status). Window's help feature can also provide more information on how to detect your anti-virus software. If the anti-virus software is on, check the Help feature that came with that software for information on how to disable it.

Windows will not detect all anti-virus software. If your anti-virus software isn't displayed in the Action Center you can try typing the name of the software or the publisher in the Start Menu's search field.

McGraw-Hill Education Content Support

For questions regarding the Glossary or the additional bonus content, e-mail techsolutions@mhedu.com or visit **http://mhp.softwareassist.com.**

For questions regarding book content, e-mail **hep_customer.service@mheducation.com**. For customers outside the United States, e-mail **international_cs@mheducation.com.**