

Identifiers (Objective 1.3)

- ☐ 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.
- ☐ JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with set, get, is, add, or remove.

Bean listener should start with add/remove and should end with Listener and inBetween type of Listener, addXListener and so on.

Declaration Rules (Objective 1.1)

- ☐ 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 multiple imports.
- ☐ The package statement (if any) must be the first (non-comment) line in a source file.
- ☐ The import statements (if any) must come after the package and before the 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 (Objective 1.1)

- ☐ 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) (Objective 1.2)

- ☐ 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 (Objective 1.2)

- ☐ 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.
- ☐ An interface is like a 100-percent abstract class, and is implicitly abstract whether you type the abstract modifier in the declaration or not.
- ☐ An interface can have only abstract methods, no concrete methods allowed.
- ☐ Interface methods are by default public and 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.
- ☐ 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 (Objectives 1.3 and 1.4)

- ☐ Methods and instance (nonlocal) variables are known as "members."
- ☐ Members can use all four access levels: public, protected, default, 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 plus 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' own subclasses.

Local Variables (Objective 1.3)

- ❑ 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 (Objective 1.3)

- ❑ final methods cannot be overridden in a subclass.
- ❑ abstract methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.
- ❑ abstract methods end in a semicolon—no curly braces.
- ❑ Three ways to spot a non-abstract method:
 - ❑ The method is not marked abstract.
 - ❑ The method has curly braces.
 - ❑ The method has code between the curly braces.
- ❑ The first nonabstract (concrete) class to extend an abstract class must implement all of the abstract class' 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 (Objective 1.4)

- ☐ As of Java 5, 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.

Variable Declarations (Objective 1.3)

- ☐ 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 reinitialized once assigned a value.
- ☐ `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
- ☐ `final` reference 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 is immutable.
- ☐ The `transient` modifier applies only to instance variables.
- ☐ The `volatile` modifier applies only to instance variables.

Array Declarations (Objective 1.3)

- ☐ 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 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 (Objective 1.4)

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

Enums (Objective 1.3)

- ☐ 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 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.

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

Encapsulation, IS-A, HAS-A (Objective 5.1)

- ☐ 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.
- ☐ IS-A, "inherits from," and "is a subtype of " are all equivalent expressions.

❑ HAS-A means an instance of one class "has a" reference to an instance of another class or another instance of the same class.

Inheritance (Objective 5.5)

❑ Inheritance allows a class to be a subclass of a superclass, and thereby inherit public and protected variables and methods of the superclass.

❑ Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.

❑ All classes (except class Object), are subclasses of type Object, and therefore they inherit Object's methods.

Polymorphism (Objective 5.2)

❑ 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.

Two-Minute Drill **157**

✓

Overriding and Overloading (Objectives 1.5 and 5.4)

❑ Methods can be overridden or overloaded; constructors can be overloaded but not overridden.

❑ Abstract methods must be overridden by the first concrete (non-abstract) subclass.

❑ With respect to the method it overrides, the overriding method

❑ Must have the same argument list.

❑ Must have the same return type, except that as of Java 5, the return type can be a subclass—this is known as a covariant return.

❑ Must not have a more restrictive access modifier because it reduces visibility and make it difficult to call method on parent type using polymorphism.

It's a fundamental principle in OOP: the child class is a fully-fledged instance of the parent class, and must therefore present *at least the same* interface as the parent class. Making protected/public things less visible would violate this idea; you could make child classes unusable as instances of the parent class.

- ❑ 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.

It means that if a method declares to throw a given exception, the overriding method in a subclass can only declare to throw that exception or its subclass. For example:

```
class A {  
    public void foo() throws IOException {..  
}  
  
class B extends A {  
    @Override  
    public void foo() throws SocketException {..} // allowed  
  
    @Override  
    public void foo() throws SQLException {..} // NOT allowed  
}
```

SocketException extends IOException, but SQLException does not.
This is because of polymorphism:

```
A a = new B();  
try {  
    a.foo();  
} catch (IOException ex) {  
    // forced to catch this by the compiler  
}
```

If B had decided to throw SQLException, then the compiler could not force you to catch it, because you are referring to the instance of B by its superclass - A. On the other hand, any subclass of IOException will be handled by clauses (catch or throws) that handle IOException

The rule that you need to be able to refer to objects by their superclass is the [Liskov Substitution Principle](#).

Since unchecked exceptions can be thrown anywhere then they are not subject to this rule. You can add an unchecked exception to the throws clause as a form of documentation if you want, but the compiler doesn't enforce anything about it.

- ❑ 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.
- ☐ 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 superclass can be overloaded in a subclass.

Method overloading can be done by changing:

- The number of parameters in two methods.
- The data types of the parameters of methods.
- The Order of the parameters of methods.

- ☐ 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.

158 Chapter 2: Object Orientation

Reference Variable Casting (Objective 5.2)

- ☐ 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 (Objective 1.2)

- ☐ When you implement an interface, you are fulfilling its contract.
- ☐ You implement an interface by properly and concretely overriding all of the methods defined by the interface.

- ❑ A single class can implement many interfaces.

Return Types (Objective 1.5)

- ❑ 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 (Objectives 1.6 and 5.4)

- ❑ A constructor is always invoked when a new object is created.

Two-Minute Drill 159

- ❑ 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 to either this() (an overloaded constructor) or 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()`).
- ☐ Issues with calls to `this()`
- ☐ May appear only as the first statement in a constructor.
- ☐ The argument list determines which overloaded constructor is called.

160 Chapter 2: Object Orientation

- ☐ Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.
- ☐ Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.

Statics (Objective 1.3)

- ☐ 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();`
- ☐ static methods can't be overridden, but they can be redefined.

Coupling and Cohesion (Objective 5.1)

- ☐ Coupling refers to the degree to which one class knows about or uses members

of another class.

- ❑ Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage.
 - ❑ Tight coupling is the undesirable state of having classes that break the rules of loose coupling.
 - ❑ Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.
 - ❑ High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.
 - ❑ Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.
-

Stack and Heap

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

Literals and Primitive Casting (Objective 1.3)

- ❑ Integer literals can be decimal, octal (e.g. 013), or hexadecimal (e.g. 0x3d).
- ❑ Literals for longs end in L or l.
- ❑ Float literals end in F or f, 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 (Objectives 1.3 and 7.6)

- ❑ 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 (e.g., in a for or an if) live until the block completes.

Basic Assignments (Objectives 1.3 and 7.6)

- ❑ 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 (e.g. +=), 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.

Two-Minute Drill **265**

✓

266 Chapter 3: Assignments

- ☐ When creating a new object, e.g., `Button b = new Button();`, three things happen:

- ☐ Make a reference variable named `b`, of type `Button`
- ☐ Create a new `Button` object
- ☐ Assign the `Button` object to the reference variable `b`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- ☐ 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 (Objective 7.3)

- ☐ 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.

Array Declaration, Construction, and Initialization (Obj. 1.3)

- ☐ Arrays can hold primitives or objects, but the array itself is always an object.

- ☐ When you declare an array, the brackets can be 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 variable 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; e.g., 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`).

Initialization Blocks (Objectives 1.3 and 7.6)

- ❑ Static initialization blocks run once, when the class is first loaded.
- ❑ Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
- ❑ If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

Using Wrappers (Objective 3.1)

- ❑ The wrapper classes correlate to the primitive types.
- ❑ Wrappers have two main functions:
 - ❑ To wrap primitives so that they can be handled like objects
 - ❑ To provide utility methods for primitives (usually conversions)
- ❑ The three most important method families are
 - ❑ `xxxValue()` Takes no arguments, returns a primitive
 - ❑ `parseXxx()` Takes a `String`, returns a primitive, throws `NFE`
 - ❑ `valueOf()` Takes a `String`, returns a wrapped object, throws `NFE`

Two-Minute Drill **267**

268 Chapter 3: Assignments

- ☐ Wrapper constructors can take a String or a primitive, except for Character, which can only take a char.
- ☐ Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

Boxing (Objective 3.1)

- ☐ As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
- ☐ Using == with wrappers created through boxing is tricky; those with the same small values (typically lower than 127), will be ==, larger values will not be ==.

Advanced Overloading (Objectives 1.5 and 5.4)

- ☐ Primitive widening uses the "smallest" method argument possible.
- ☐ Used individually, boxing and var-args are compatible with overloading.
- ☐ You CANNOT widen from one wrapper type to another. (IS-A fails.)
- ☐ You CANNOT widen and then box. (An int can't become a Long.)
- ☐ You can box and then widen. (An int can become an Object, via an Integer.)
- ☐ You can combine var-args with either widening or boxing.

Garbage Collection (Objective 7.4)

- ☐ 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 GCed, even though they refer to each other.
- ☐ Request garbage collection with System.gc(); (only before the SCJP 6).
- ☐ Class Object 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 uneligibilize an object for GC from within finalize().

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

Relational Operators (Objective 7.6)

- ❑ 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 (Objective 7.6)

- ❑ `instanceof` is for reference variables only, and checks for 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 (Objective 7.6)

- ❑ There are four primary math operators: add, subtract, multiply, and divide.
- ❑ The remainder 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 `-`.

Two-Minute Drill **311**

✓

String Concatenation Operator (Objective 7.6)

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

Increment/Decrement Operators (Objective 7.6)

- ❑ Prefix operators (`++` and `--`) run before the value is used in the expression.

- ❑ Postfix operators (++ and --) 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) (Objective 7.6)

- ❑ Returns one of two values based on whether a boolean expression is true or false.
- ❑ Returns the value after the ? if the expression is true.
- ❑ Returns the value after the : if the expression is false.

Logical Operators (Objective 7.6)

- ❑ The exam covers six "logical" operators: &, |, ^, !, &&, and ||.
- ❑ Logical operators 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.

312 Chapter 4: Operators

You might want to loop through them several times.

Writing Code Using if and switch Statements (Obj. 2.1)

- ❑ 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 variable.
- ❑ 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, and

char data types. You can't say,
long s = 30;
switch(s) { }

- ❑ The case constant must be a literal or final variable, or a constant expression, including an enum. 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 case matches, the default block will be entered, and 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.

Two-Minute Drill **397**

✓

398 Chapter 5: Flow Control, Exceptions, and Assertions

Writing Code Using Loops (Objective 2.2)

- ❑ 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 separated by a comma.
- ❑ An enhanced for statement (new as of Java 6), 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.

- ❑ You cannot use a number (old C-style language construct) 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 enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (Objective 2.2)

- ❑ An unlabeled break statement will cause the current iteration of the innermost looping construct 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 similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6)

- ❑ 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.
- ❑ If you use an optional finally block, it will always be invoked, regardless of whether an exception in the corresponding try is thrown or not, and regardless of whether a thrown exception is caught or not.
- ❑ 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 create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.

❑ 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! The compiler will stop you from defining catch clauses that can never be reached.

❑ Some exceptions are created by programmers, some by the JVM.

Two-Minute Drill **399**

400 Chapter 5: Flow Control, Exceptions, and Assertions

Working with the Assertion Mechanism (Objective 2.3)

❑ Assertions give you a way to test your assumptions during development and debugging.

❑ Assertions are typically enabled during testing but disabled during deployment.

❑ You can use `assert` as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses `assert` as an identifier (for example, a method name), use the `-source 1.3` command-line flag to `javac`.

❑ Assertions are disabled at runtime by default. To enable them, use a command-line flag `-ea` or `-enableassertions`.

❑ Selectively disable assertions by using the `-da` or `-disableassertions` flag.

❑ If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.

❑ You can enable and disable assertions on a class-by-class basis, using the following syntax:

```
java -ea -da:MyClass TestClass
```

❑ You can enable and disable assertions on a package-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).

❑ Do not use assertions to validate arguments to public methods.

- ❑ Do not use assert expressions that cause side effects. Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.
- ❑ Do use assertions—even in public methods—to validate that a particular code block will never be reached. You can use `assert false`; for code that should never be reached, so that an assertion error is thrown immediately if the assert statement is executed.

O-MINUTE DRILL

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

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- ❑ 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—its methods can't be overridden.
- ❑ When the JVM finds a String literal, it is added to the String literal pool.
- ❑ Strings have a method: `length()`; arrays have an attribute named `length`.
- ❑ The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- ❑ StringBuilder methods should run faster than StringBuffer methods.
- ❑ All of the following bullets apply to both StringBuffer and StringBuilder:
 - ❑ They are mutable—they can change without creating a new object.
 - ❑ StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
 - ❑ StringBuffer `equals()` is not overridden; it doesn't compare values.
 - ❑ 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()`.
 - ❑ StringBuffer methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

File I/O (Objective 3.2)

- ❑ The classes you need to understand in `java.io` are `File`, `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, `PrintWriter`, and `Console`.
- ❑ A new File object doesn't mean there's a new file on your hard drive.

❑ File objects can represent either a file or a directory.

Two-Minute Drill **511**

✓

❑ The File class lets you manage (add, rename, and delete) files and directories.

❑ The methods `createNewFile()` and `mkdir()` add entries to your file system.

❑ `FileWriter` and `FileReader` are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.

❑ Classes in `java.io` are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)

❑ It's very common to "wrap" a `BufferedReader` around a `FileReader` or a `BufferedWriter` around a `FileWriter`, to get access to higher-level (more convenient) methods.

❑ `PrintWriters` can be used to wrap other `Writers`, but as of Java 5 they can be built directly from `Files` or `Strings`.

❑ Java 5 `PrintWriters` have new `append()`, `format()`, and `printf()` methods.

❑ `Console` objects can read non-echoed input and are instantiated using `System.console()`.

Serialization (Objective 3.3)

❑ The classes you need to understand are all in the `java.io` package; they include: `ObjectOutputStream` and `ObjectInputStream` primarily, and `FileOutputStream` and `FileInputStream` because you will use them to create the low-level streams that the `ObjectXxxStream` classes will use.

❑ A class must implement `Serializable` before its objects can be serialized.

❑ The `ObjectOutputStream.writeObject()` method serializes objects, and the `ObjectInputStream.readObject()` method deserializes objects.

❑ If you mark an instance variable `transient`, it will not be serialized even though the rest of the object's state will be.

❑ You can supplement a class's automatic serialization process by implementing

the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.

❑ If a superclass implements `Serializable`, then its subclasses do automatically.

❑ If a superclass doesn't implement `Serializable`, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its `superconstructor(s)`.

❑ `DataInputStream` and `DataOutputStream` aren't actually on the exam, in spite of what the Sun objectives say.

512 Chapter 6: Strings, I/O, Formatting, and Parsing

Dates, Numbers, and Currency (Objective 3.4)

- ☐ The classes you need to understand are `java.util.Date`, `java.util.Calendar`, `java.text.DateFormat`, `java.text.NumberFormat`, and `java.util.Locale`.
- ☐ Most of the `Date` class's methods have been deprecated.
- ☐ A `Date` is stored as a long, the number of milliseconds since January 1, 1970.
- ☐ `Date` objects are go-betweens the `Calendar` and `Locale` classes.
- ☐ The `Calendar` provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
- ☐ Create `Calendar` instances using static factory methods (`getInstance()`).
- ☐ The `Calendar` methods you should understand are `add()`, which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and `roll()`, which works like `add()` but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the `Calendar`'s year value.)
- ☐ `DateFormat` instances are created using static factory methods (`getInstance()` and `getDateInstance()`).
- ☐ There are several format "styles" available in the `DateFormat` class.
- ☐ `DateFormat` styles can be applied against various `Locales` to create a wide array of outputs for any given date.
- ☐ The `DateFormat.format()` method is used to create `Strings` containing properly formatted dates.
- ☐ The `Locale` class is used in conjunction with `DateFormat` and `NumberFormat`.
- ☐ Both `DateFormat` and `NumberFormat` objects can be constructed with a specific, immutable `Locale`.
- ☐ For the exam you should understand creating `Locales` using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting (Objective 3.5)

- ☐ `regex` is short for regular expressions, which are the patterns used to search for data within large data sources.
- ☐ `regex` is a sub-language that exists in Java and other languages (such as Perl).
- ☐ `regex` lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

Two-Minute Drill **513**

514 Chapter 6: Strings, I/O, Formatting, and Parsing

- ☐ Study the `\d`, `\s`, `\w`, and `.` metacharacters
- ☐ regex provides for *quantifiers* which allow you to specify concepts like: "look for one or more digits in a row."
- ☐ Study the `?`, `*`, and `+` greedy quantifiers.
- ☐ Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance `String s = "\\d";`
- ☐ The Pattern and Matcher classes have Java's most powerful regex capabilities.
- ☐ You should understand the Pattern `compile()` method and the Matcher `matches()`, `pattern()`, `find()`, `start()`, and `group()` methods.
- ☐ You WON'T need to understand Matcher's replacement-oriented methods.
- ☐ You can use `java.util.Scanner` to do simple regex searches, but it is primarily intended for tokenizing.
- ☐ Tokenizing is the process of splitting delimited data into small pieces.
- ☐ In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- ☐ Tokenizing can be done with the Scanner class, or with `String.split()`.
- ☐ Delimiters are single characters like commas, or complex regex expressions.
- ☐ The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- ☐ The Scanner class allows you to tokenize Strings or streams or files.
- ☐ The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- ☐ New to Java 5 are two methods used to format data for output. These methods are `format()` and `printf()`. These methods are found in the `PrintStream` class, an instance of which is the `out` in `System.out`.
- ☐ The `format()` and `printf()` methods have identical functionality.
- ☐ Formatting data with `printf()` (or `format()`) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- ☐ The `format()` method allows you to mix literals in with your format strings.
- ☐ The format string values you should know are
- ☐ Flags: `-`, `+`, `0`, `"`, `"`, and `(`
- ☐ Conversions: `b`, `c`, `d`, `f`, and `s`
- ☐ If your conversion character doesn't match your argument type, an exception will be thrown.

Here are some of the key points from this chapter.

Overriding hashCode() and equals() (Objective 6.2)

- ☐ equals(), hashCode(), and toString() are public.
- ☐ Override toString() so that System.out.println() or other methods can see something useful, like your object's state.
- ☐ Use == to determine if two reference variables refer to the same object.
- ☐ Use equals() to determine if two objects are meaningfully equivalent.
- ☐ If you don't override equals(), your objects won't be useful hashing keys.
- ☐ If you don't override equals(), different objects can't be considered equal.
- ☐ Strings and wrappers override equals() and make good hashing keys.
- ☐ When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
- ☐ When overriding equals(), compare the objects' significant attributes.
- ☐ Highlights of the equals() contract:
 - ☐ Reflexive: x.equals(x) is true.
 - ☐ Symmetric: If x.equals(y) is true, then y.equals(x) must be true.
 - ☐ Transitive: If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
 - ☐ Consistent: Multiple calls to x.equals(y) will return the same result.
 - ☐ Null: If x is not null, then x.equals(null) is false.
 - ☐ If x.equals(y) is true, then x.hashCode() == y.hashCode() is true.
 - ☐ If you override equals(), override hashCode().
- ☐ HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- ☐ An appropriate hashCode() override sticks to the hashCode() contract.
- ☐ An efficient hashCode() override distributes keys evenly across its buckets.
- ☐ An overridden equals() must be at least as precise as its hashCode() mate.
- ☐ To reiterate: if two objects are equal, their hashcodes must be equal.
- ☐ It's legal for a hashCode() method to return the same value for all instances (although in practice it's very inefficient).

Two-Minute Drill **631**

✓

632 Chapter 7: Generics and Collections

- ☐ Highlights of the hashCode() contract:
 - ☐ Consistent: multiple calls to x.hashCode() return the same integer.
 - ☐ If x.equals(y) is true, x.hashCode() == y.hashCode() is true.
 - ☐ If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.

- ❑ transient variables aren't appropriate for equals() and hashCode().

Collections (Objective 6.1)

- ❑ Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- ❑ Three meanings for "collection":
 - ❑ **collection** Represents the data structure in which objects are stored
 - ❑ **Collection** java.util interface from which Set and List extend
 - ❑ **Collections** A class that holds static collection utility methods
- ❑ Four basic flavors of collections include Lists, Sets, Maps, Queues:
- ❑ **Lists of things** Ordered, duplicates allowed, with an index.
- ❑ **Sets of things** May or may not be ordered and/or sorted; duplicates not allowed.
- ❑ **Maps of things with keys** May or may not be ordered and/or sorted; duplicate keys are not allowed.
- ❑ **Queues of things to process** Ordered by FIFO or by priority.
- ❑ Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
- ❑ **Ordered** Iterating through a collection in a specific, non-random order.
- ❑ **Sorted** Iterating through a collection in a sorted order.
- ❑ Sorting can be alphabetic, numeric, or programmer-defined.

Key Attributes of Common Collection Classes (Objective 6.1)

- ❑ ArrayList: Fast iteration and fast random access.
- ❑ Vector: It's like a slower ArrayList, but it has synchronized methods.
- ❑ LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- ❑ HashSet: Fast access, assures no duplicates, provides no ordering.
- ❑ LinkedHashSet: No duplicates; iterates by insertion order.
- ❑ TreeSet: No duplicates; iterates in sorted order.
- ❑ HashMap: Fastest updates (key/values); allows one null key, many null values.
- ❑ Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
- ❑ LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
- ❑ TreeMap: A sorted map.
- ❑ PriorityQueue: A to-do list ordered by the elements' priority.

Using Collection Classes (Objective 6.3)

- ❑ Collections hold only Objects, but primitives can be autoboxed.
- ❑ Iterate with the enhanced for, or with an Iterator via hasNext() & next().

- ❑ `hasNext()` determines if more elements exist; the `Iterator` does NOT move.
- ❑ `next()` returns the next element AND moves the `Iterator` forward.
- ❑ To work correctly, a `Map`'s keys must override `equals()` and `hashCode()`.
- ❑ `Queues` use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.
- ❑ As of Java 6 `TreeSets` and `TreeMaps` have new navigation methods like `floor()` and `higher()`.
- ❑ You can create/extend "backed" sub-copies of `TreeSets` and `TreeMaps`.

Sorting and Searching Arrays and Lists (Objective 6.5)

- ❑ Sorting can be in natural order, or via a `Comparable` or many `Comparators`.
- ❑ Implement `Comparable` using `compareTo()`; provides only one sort order.
- ❑ Create many `Comparators` to sort a class many ways; implement `compare()`.
- ❑ To be sorted and searched, a `List`'s elements must be *comparable*.
- ❑ To be searched, an array or `List` must first be sorted.

Utility Classes: Collections and Arrays (Objective 6.5)

- ❑ Both of these `java.util` classes provide
- ❑ A `sort()` method. Sort using a `Comparator` or sort using natural order.
- ❑ A `binarySearch()` method. Search a pre-sorted array or `List`.

Two-Minute Drill 633

634 Chapter 7: Generics and Collections

- ❑ `Arrays.asList()` creates a `List` from an array and links them together.
- ❑ `Collections.reverse()` reverses the order of elements in a `List`.
- ❑ `Collections.reverseOrder()` returns a `Comparator` that sorts in reverse.
- ❑ `Lists` and `Sets` have a `toArray()` method to create arrays.

Generics (Objective 6.4)

- ❑ Generics let you enforce compile-time type safety on `Collections` (or other classes and methods declared using generic type parameters).
- ❑ An `ArrayList<Animal>` can accept references of type `Dog`, `Cat`, or any other subtype of `Animal` (subclass, or if `Animal` is an interface, implementation).
- ❑ When using generic collections, a cast is not needed to get (declared type) elements

out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
```

```
List list = new ArrayList();
```

```
// more code
```

```
String s = gList.get(0); // no cast needed
```

```
String s = (String)list.get(0); // cast required
```

❑ You can pass a generic collection into a method that takes a non-generic collection,

but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.

❑ If the compiler can recognize that non-type-safe code is potentially endangering

something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a `List<String>` into a method declared as `void foo(List aList) { aList.add(anInteger); }`

You'll get a warning because `add()` is potentially "unsafe".

❑ "Compiles without error" is not the same as "compiles without warnings." A compilation *warning* is not considered a compilation *error* or *failure*.

❑ Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.

❑ Polymorphic assignments applies only to the base type, not the generic type

parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>(); // yes
```

You can't say

```
List<Animal> aList = new ArrayList<Dog>(); // no
```

❑ The polymorphic assignment rule applies everywhere an assignment can be

made. The following are NOT allowed:

```
void foo(List<Animal> aList) { } // cannot take a List<Dog>
```

```
List<Animal> bar() { } // cannot return a List<Dog>
```

❑ Wildcard syntax allows a generic method, accept subtypes (or supertypes) of

the declared type of the method argument:

```
void addD(List<Dog> d) { } // can take only <Dog>
```

```
void addD(List<? extends Dog>) { } // take a <Dog> or <Beagle>
```

❑ The wildcard keyword `extends` is used to mean either "extends" or "implements."

So in `<? extends Dog>`, `Dog` can be a class or an interface.

❑ When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.

❑ When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.

❑ `List<Object>` refers only to a `List<Object>`, while `List<?>` or

`List<? extends Object>` can hold any type of object, but for access only.

❑ Declaration conventions for generics use `T` for type and `E` for element:

```
public interface List<E> // API declaration for List
```

```
boolean add(E o) // List.add() declaration
```

❑ The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { } // a class
```

```
T anInstance; // an instance variable
```

```
Foo(T aRef) {} // a constructor argument
```

```
void bar(T aRef) {} // a method argument
```

```
T baz() {} // a return type
```

The compiler will substitute the actual type.

❑ You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

❑ You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

is NOT using T as the return type. This method has a void return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

Here are some of the key points from this chapter.

Inner Classes

❑ A "regular" inner class is declared *inside* the curly braces of another class, but

outside any method or other code block.

❑ An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers. (Never both abstract and final together— remember that abstract *must* be subclassed, whereas final *cannot* be subclassed).

❑ An inner class instance shares a special relationship with an instance of the

enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked private.

❑ To instantiate an inner class, you must have a reference to an instance of the

outer class.

❑ From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

❑ From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names,

and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();
```

```
MyOuter.MyInner inner = mo.new MyInner();
```

❑ From code within the inner class, the keyword `this` holds a reference to the inner class instance. To reference the *outer* `this` (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword `this` with the outer class name as follows: `MyOuter.this`;

Method-Local Inner Classes

❑ A method-local inner class is defined within a method of the enclosing class.

❑ For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but *after* the class definition code.

❑ A method-local inner class cannot use variables declared within the method

(including parameters) unless those variables are marked `final`.

Two-Minute Drill **683**

✓

❑ The only modifiers you can apply to a method-local inner class are `abstract`

and `final`. (Never both at the same time, though.)

Anonymous Inner Classes

❑ Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.

❑ An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace.

This

is a rare case in Java, a curly brace followed by a semicolon.

❑ Because of polymorphism, the only methods you can call on an anonymous

inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer

of the reference variable type.

❑ An anonymous inner class can extend one subclass *or* implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous

inner class cannot do both. In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.

❑ An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: `});`

Static Nested Classes

❑ Static nested classes are inner classes marked with the `static` modifier.

- ❑ A static nested class is *not* an inner class, it's a top-level nested class.
- ❑ Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- ❑ Instantiating a static nested class requires using both the outer and nested class names as follows:
`BigOuter.Nested n = new BigOuter.Nested();`
- ❑ A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an *outer* this reference).