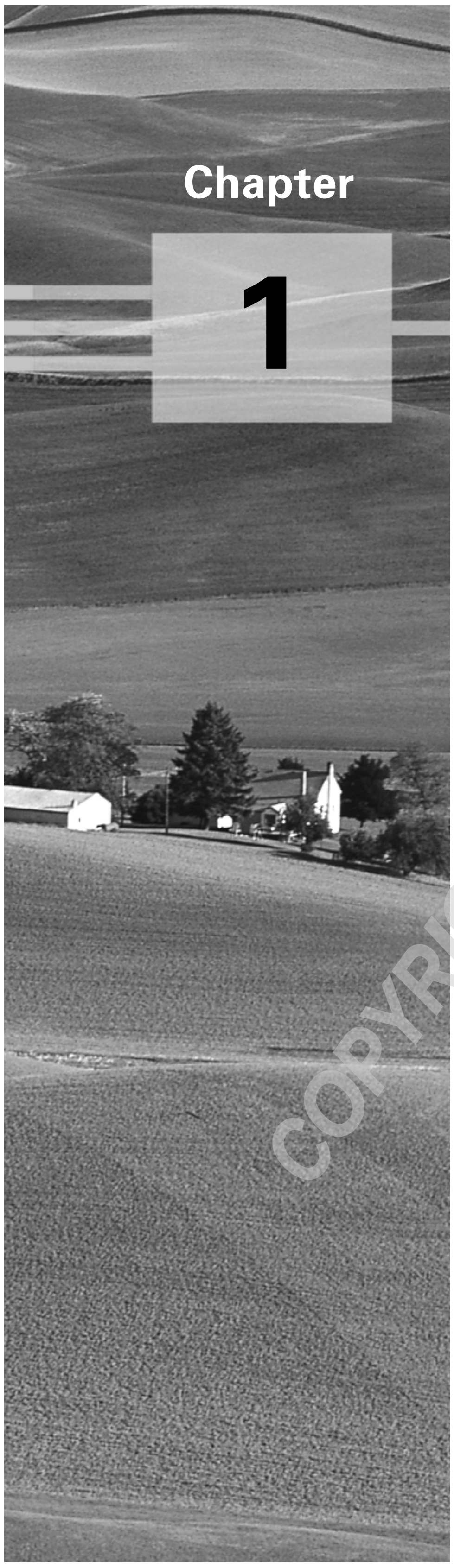


Declarations and Access Control

SUN CERTIFIED PROGRAMMER FOR JAVA 2 PLATFORM EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.**
- **Declare classes, inner classes, methods, instance variables, static variables, and automatic (method local) variables, making appropriate use of all permitted modifiers (such as *public, final, static, abstract,* and so forth). State the significance of each of these modifiers, both singly and in combination, and state the effect of package relationships on declared items qualified by these modifiers.**
- **For a given class, determine if a default constructor will be created and, if so, state the prototype of that constructor.**
- **State the legal return types for any method, given the declarations of all related methods in this or parent classes.**

The common theme of these four objectives is declaration. Declaration tells the compiler that an entity exists and also provides the name and nature of the entity. Since everything you create has to be declared, you must know how to declare correctly and be able to make appropriate use of all the declaration tools available to you. This objective recognizes the importance of declarations by requiring you to know everything about Java declarations.

Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.

This objective addresses your understanding of all aspects of arrays. Arrays are the simplest possible data collection and are supported by all modern programming languages. However, Java's arrays are different from those in other languages, especially C and C++, because they are actually exotic objects. This objective recognizes the importance of being able to make use of all the functionality of arrays. You will be expected to know how to declare, construct, and initialize arrays.

Critical Information

You will need to know about the following aspects of arrays:

- Declaration
- Construction
- Initialization

These three steps are the first stages of an array's life cycle. The exam expects you to be familiar with all aspects of array declaration, construction, and initialization.

Array Declaration

Java supports two formats for array declaration. The first format is the classical C/C++ syntax, in which the element type comes first, followed by the variable name, followed by square brackets. This syntax is illustrated in line 1 as follows. The second format begins with the element type, which is followed by the square brackets and then by the variable name; the second format is illustrated in line 2 as follows.

```
1. int intarr[];
2. int[] intarr;
```

Array elements may be any of the following:

- Primitives (as shown previously)
- Object references
- References to other arrays

Declaration of arrays of primitives is illustrated in the preceding code. The following code shows the two ways to declare an array of object references:

```
1. String myStrings[];
2. String[] myStrings;
```

When an array's elements are references to other arrays, we have a special case of an array of object references. However, the declaration syntax is different from the syntax shown previously. The effect is like a multidimensional array, as shown in the following code. In line 3, the two declaration formats are combined, resulting in a declaration that is legal but difficult to read.

```
1. float[][][][] matrixOfFloats;
2. float matrixOfFloats[][][][];
3. float[][][] matrixOfFloats[];
```

Array Construction

Array declaration is like declaration of any other object reference variable. The declaration only tells the compiler about the type of the variable. No runtime object is created until `new` is invoked. Note that the declaration does not specify the number of elements in the array; the number of elements is supplied at runtime, as shown in the following examples:

```
 1. long longarr[];
 2. longarr = new long[10];
 3. String[] myStrings;
 4. myStrings = new String[22];
 5. double[][] matrixOfDoubles;
 6. matrixOfDoubles = new double[1152][900];
 7. int[][] matrixOfInts;
 8. matrixOfInts = new int[500][];
 9. matrixOfInts[0] = new int[33];
10. matrixOfInts[1] = new int[44];
```

Line 6 illustrates the most common way to construct the equivalent of a two-dimensional array. The `matrixOfDoubles` array looks like a 2-D array and can be treated as such. In reality it is an array of 1152 arrays of doubles. Each of those arrays of doubles has 900 elements. Line 8 constructs an array of 500 arrays of ints; those 500 arrays of ints are not yet allocated, and each of them may have a different length, as shown in lines 9 and 10.

Array Element Initialization

When an array is allocated, all of its elements are initialized. The initialization value depends on the type of array element, as shown in Table 1.1. The values are easy to remember because the numeric types are initialized to zero, and non-numeric types are initialized to values that are similar to zero. Also, these are the same values that are used for construction-time default initialization of an object's fields. The default for `char` is Unicode zero, which is the null character.

TABLE 1.1: Array Element Initialization Values

Element Type	Initial Value
`byte`	0
`short`	0
`int`	0
`long`	0
`char`	'\u0000'
`float`	0.0f
`double`	0.0d
`boolean`	false
Object Reference	null

Declaration, Construction, and Initialization in a Single Statement

An array can be declared, constructed, and initialized in a single statement, as illustrated below:

```
double[] ds = {1.2, 2.3, 3.4, 4.5, Math.PI, Math.E};
```

With this syntax, the invocation of `new` and the size of the array are implicit. The array elements are initialized to the values given between the brackets, rather than their default initialization values.

Exam Essentials

Know how to declare and construct arrays. The declaration includes one empty pair of square brackets for each dimension of the array. The square brackets can appear before or after the array name. Arrays are constructed with the keyword `new`.

Know the default initialization values for all possible types. The initialization values are zero for numeric type arrays, `false` for boolean arrays, and `null` for object reference type arrays.

Know how to declare, construct, and initialize in a single statement. This notation uses initialization values in curly brackets; for example, `int[] intarr = {1, 2, 3};`.

Key Terms and Concepts

Array declaration The square brackets in the declaration can appear before or after the variable name.

Default initialization values An array's elements are initialized to zero for numeric types and to values that resemble zero for non-numeric types, as shown in Table 1.1.

Sample Questions

1. Which of the following are legal array declarations?

A. `int[] z[];`

B. `String[][] z[];`

C. `char[] z;`

D. `char z[];`

E. `float[5] z;`

Answer: All the declarations are legal except E. You cannot specify an array's size in its declaration.

2. What are the default initialization values for an array of type `char`?

Answer: '\u0000' (the null character). Table 1.1 lists initialization values for arrays of all primitive types.

3. What are the default initialization values for an array of type `boolean`?

   Answer: `false`. Table 1.1 lists initialization values for arrays of all primitive types.

4. Which of the following are legal ways to declare, construct, and initialize an array in a single line?

   A. `char[3] cs = {'a', 'b', 'c'};`

   B. `char[] cs = ['a', 'b', 'c'];`

   C. `char[] cs = {'a', 'b', 'c', 'd'};`

   D. `char cs[] = {'a', 'b', 'c', 'd'};`

   Answer: C and D are both legal. A is illegal because the array size is stated explicitly. B is illegal because it uses square brackets where curly brackets are required.

Declare classes, inner classes, methods, instance variables, static variables, and automatic (method local) variables, making appropriate use of all permitted modifiers (such as *public, final, static, abstract,* and so forth). State the significance of each of these modifiers, both singly and in combination, and state the effect of package relationships on declared items qualified by these modifiers.

This objective covers a lot of subject matter (and paper!). The big idea is modifiers. This section reviews Java's modifiers, including access modifiers, which relate to the phrase in the objective that mentions "the effect of package relationships."

It is possible to write Java programs that make little or no use of modifiers. However, appropriate use of modifiers—especially access modifiers—is essential for creating classes that are secure, object-oriented, and maintainable. This objective recognizes the importance of a working knowledge of Java's identifiers.

Critical Information

A *modifier* is a Java keyword that affects the behavior of the feature it precedes. (A feature of a class is the class itself, or a method, variable, or inner class of the class.) Java's modifiers are listed as follows:

- `private`
- `protected`
- `public`
- `final`
- `abstract`
- `static`
- `synchronized`
- `transient`
- `native`
- `volatile`

The first three of these modifiers (`private`, `protected`, and `public`) are known as *access modifiers*. The remaining modifiers do not fall into any clear-cut categories. (Another modifier, `strictfp`, is a new addition to Java 2. It is discussed briefly in Chapter 4, "Language Fundamentals," but does not appear on the exam.)

The Access Modifiers

There are four possible access modes for Java features: public, protected, default, and private. Three of these modes (public, protected, and private) correspond to access modifiers. The fourth mode (default) is the default and has no corresponding keyword modifier; a feature is default if it is not marked with `private`, `protected`, or `public`. Access modifiers generally dictate which *classes*, and not which *instances*, have access to features. You may use one access modifier at most to modify a feature. Automatic variables (that is, variables defined within the scope of a method or code block) may not take access modifiers.

The most restrictive access modifier is *private*. Only methods, data, and inner classes may be private; private classes are not permitted. A private method may only be called within the class that defines the method. A private variable may only be read and written within the class that defines the variable. A private inner class may only be accessed by the containing class; subclasses of the containing class may not access a private inner class. The granularity of private access is the class level, not the instance level: if a class has a private feature, then an instance of the class may access that private feature of *any instance* of the class.

"Default" is the default mode in the absence of a `private`, `protected`, or `public` modifier. This mode has no corresponding Java keyword. Classes, methods, and data may be default. A default class may be accessed by any class within its class's package. A default method of a class may be called from anywhere within the class's package. A default variable of a class may be read and written from anywhere within the class's package. A default inner class may be accessed by a subclass of the containing class, provided the containing class and the subclass are in the same package.

The *protected* access mode grants access permission to all members of the owning class's package, just as default access does. Moreover, additional access is granted if the class that owns a protected feature has a subclass in a different package. In this case an instance of the subclass may access protected data and call protected methods inherited from the parent instance. A protected inner class may be accessed by any subclass of the containing class, even if the subclass is in a different package from the containing class.

The most accessible access modifier is *public*. A public class may be accessed by any class. A public method may be called by any class (provided the calling class may access the class that owns the method in question). A public variable may be read and written by any class (provided the reading/writing class may access the class that owns the variable in question). A public inner class, like a protected inner class, may be accessed by any subclass of the containing class; moreover, public inner classes are a bit more available for manipulation via Java's reflection mechanism. The exact details are beyond the scope of the exam.

Table 1.2 summarizes Java's four access modes.

TABLE 1.2: Java's Access Modes

Access Mode	Java Keyword	Methods and Fields	Inner Classes
Private	`private`	Accessible only by owning class	Accessible only by owning class
Default	(no keyword)	Accessible by all classes in package of owning class	Accessible by same-package subclasses of enclosing class
Protected	`protected`	Accessible by all classes in package of owning class and by subclasses of owning class	Accessible by all subclasses of enclosing class
Public	`public`	Universal access	Accessible by all subclasses of enclosing class, plus some reflection

Table 1.3 shows which access modes are available to which feature types.

TABLE 1.3: Features and Access Modes

Access Mode	Classes	Methods	Class Variables	Inner Classes
Private	NO	YES	YES	YES
Default	YES	YES	YES	YES
Protected	NO	YES	YES	YES
Public	YES	YES	YES	YES

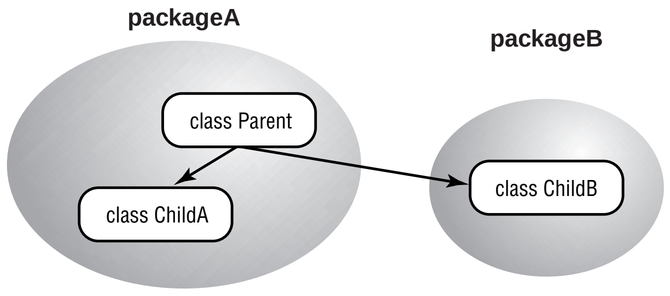
Access and Packages

A package defines a namespace for classes: Every class within a package must have a unique name. Packages are hierarchical: A package may contain other packages. Package naming uses a period (`.`) to separate the components of a package hierarchy name. Thus, for example, the `java` package contains an `awt` subpackage, which in turn contains a subpackage called `event`. The fully qualified name of this last package is `java.awt.event`. Within this package is a class called `AdjustmentEvent`, whose fully qualified name is `java.awt.event. AdjustmentEvent`. No other class named `AdjustmentEvent` may exist in the `java.awt.event` package; however, it is legal for a class named `AdjustmentEvent` to exist in any other package.

One way to create a Java class that resides inside a package is to put a `package` declaration at the beginning of your source file. (Further details on package creation are not required knowledge for the Programmer's Exam and are beyond the scope of this book.) If you do not explicitly use a `package` declaration, a package might be implicitly created for you. At runtime, the Java environment creates a "default package" that contains all classes in the current working directory that do not explicitly belong to other packages.

Figure 1.1 shows a superclass, `Parent`, which is part of the `packageA` package. The `Parent` superclass has two subclasses. One subclass, `childA`, is also in the `packageA` package. The other subclass, `childB`, is in a different package, called `packageB`.

FIGURE 1.1: Package and subclass relationships



The `childB` subclass has access to all features of the `Parent` superclass that are public or protected. The `childA` subclass also has access to the public and protected features of `Parent`. In addition, since `childA` and `Parent` are in the same package, `childA` has access to all default features of `Parent`; this access would be the same if `childA` were not a subclass of `Parent`.

Default Access in Interfaces

In a class definition, the default access mode for a method is, of course, "default." This is not the case for a method in an interface. All interface methods are inherently public, so the default access mode for a method in an interface is public. Thus the following two interface versions are functionally identical:

```
Interface Inter {    // no access modifier
  double getWeight(Animal theAnimal);
}

Interface Inter {    // explicitly public
  public double getWeight(Animal theAnimal);
}
```

You are not allowed to apply any access modifier other than `public` to a method in an interface. The following interface generates a compiler error.

```
Interface BadAccess {
  protected double x();  // Compiler error
}
```

Access Examples

This section presents several examples of the use of access modifiers. All the examples refer to a simple class called `packageA.Parent`, which is listed below:

```
package packageA;

public class Parent {
  private int    iPrivate;
  int            iDefault;
  protected int  iProtected;
  public int     iPublic;
}
```

The class definitions that follow illustrate each of the four Java access modes. All the code compiles; lines that illustrate illegal access attempts have been commented out.

The first example illustrates the private access mode. Since only the `Parent` class may access a private feature of the `Parent` class, our example is an expansion of `Parent`.

```
package packageA;

public class Parent {
  private int    iPrivate;
  int            iDefault;
  protected int  iProtected;
  public int     iPublic;

```

```
 9.   void xxx() {
10.     iPrivate = 10;         // My own
11.     Parent other = new Parent();
12.     other.iPrivate = 20;   // Someone else's
13.   }
14. }
```

Line 10 is an obvious use of the private access mode. The current instance of the `Parent` class is modifying its own version of `iPrivate`. Line 12 is less obvious: the current instance is modifying the `iPrivate` of a different instance of `Parent`. This example illustrates the principle that access modifiers grant access to *classes*, and not to *instances*.

The next example illustrates the default access mode, which grants access permission to all classes in the same package as the class that owns the default feature. Here we will create a second class in the `packageA` package.

```
 1.  package packageA;
 2.
 3.  public class InSamePackage {
 4.  void makeItSo() {
 5.    Parent parent = new Parent();
 6.    // parent.iPrivate = 10;
 7.    parent.iDefault = 20;
 8.    parent.iProtected = 30;
 9.    parent.iPublic = 40;
10.   }
11. }
```

Line 6 has to be commented out, because the attempted access to `iPrivate` in another class would not be allowed. Line 7 is legal because `iDefault` is a default variable in a class in the current package. Line 8 is legal because `iProtected` is a protected variable in a class in the current package. Line 9 is legal because `iPublic` is a public variable.

The next example illustrates the protected access mode, which grants access permission to all classes in the same package as the class that owns the protected feature, as well as to subclasses of the class that owns the feature. Here we will create a class in a second package, called `packageB`.

```
 1. package packageB;
 2.
 3. public class InDifferentPackage
 4.   extends packageA.Parent {
 5.  void aMethod() {
 6.     packageA.Parent parent =
          new packageA.Parent();
 7.     // parent.iPrivate = 10;
 8.     // parent.iDefault = 20;
 9.     // parent.iProtected = 30;
10.     iProtected = 40;
11.     parent.iPublic = 50;
12.   }
13. }
```

Lines 7 and 8 have to be commented out for reasons that were illustrated in the previous examples. It may be surprising that line 9 does not compile and also must be commented out. The `iProtected` variable is protected; an instance of a subclass in a different package (such as the current instance of `packageB.InDifferentPackage` here) does not have access to the `iProtected` of every instance of `packageA.Parent`. Rather, an instance of the subclass may access the one instance of `iProtected` that the subclass instance inherits by virtue of extending `Parent`. The current instance of `InDifferentPackage` may access *its own* `iProtected`, as in line 10. Of course, line 11 compiles because it represents access to a public feature.

The last example in this section illustrates access and inner classes. Consider the following superclass:

```
 1. package packageX;
 2.
 3. public class Parent {
 4.   protected class Prot { public Prot() {} }
 5. }
```

The superclass has a subclass in a different package:

```
 6. package packageY;
 7. import packageX.*;
 8.
 9. public class Child extends Parent {
10.   void xxx() {
11.     Prot p = new Prot();
12.   }
13. }
```

The `Prot` inner class is protected, so it is accessible from a subclass (`Child` in `packageY`) of the enclosing class (`Parent` in `packageX`), even though the parent and child classes reside in different packages. If the inner class were private or default, line 11 would not compile.

The Miscellaneous Modifiers

The remainder of this section examines the Java modifiers that have nothing to do with access. They are considerably simpler than the access modifiers. We will review each of the following keywords in turn:

- `final`
- `abstract`
- `static`
- `native`
- `transient`

- synchronized
- volatile

final

The final keyword conveys the sense that a feature may not be altered. Classes, methods, and variables may be final. A final class may not be subclassed, and a final method may not be overridden.

A final variable, once initialized, may not be written. The declaration and initialization of a final variable may appear in the same statement or in different statements. If they appear in different statements, those statements are not required to be consecutive, as illustrated in the following code sample:

```
final int j;        // Declare j
final int k = 10;   // Intervening statement
j = 20;             // Initialize Java
```

The final keyword, unlike the access modifiers, can be applied to the automatic variables and arguments of a method. A final automatic variable may not be written after it is initialized. A final argument may not be written at all. The following code sample illustrates final data in a method:

```
void aMethod(int x, final double z) {
  final char c = 'c';
  // etc.
}
```

abstract

The abstract keyword conveys the sense that a feature is somehow incomplete and cannot be used until further information is provided. Classes and methods may be final.

When you declare a method to be abstract, the class that contains the method has no definition for that method. Instead, the method definition is deferred to one or more subclasses. After the method name and parenthetical argument list, an abstract method has only a semicolon, where a non-abstract method provides the method body enclosed in curly brackets. Subclasses provide the body of an abstract method.

An abstract class may not be instantiated. A class must be declared abstract if any of the following conditions apply:

- The class contains one or more abstract methods.
- The class does not provide an implementation for each of the abstract methods of its superclass.
- The class declares that it implements an interface, and the class does not provide an implementation for each method of the interface.

The following code illustrates abstract classes.

```
 1. abstract class Parent {
 2.   abstract void x(int a);
 3. }
 4.
 5. class ChildA extends Parent {
 6.   void x(int a) {
 7.     System.out.println("a = " + a);
 8.   }
 9. }
10.
11. class ChildB extends Parent {
12.   void x(int a) {
13.     System.out.println("I did it my way" + a);
14.   }
15. }
16.
17. abstract class ChildC extends Parent { }
```

The class `Parent` must be declared abstract on line 1, because it contains an abstract method. The classes `ChildA` and `ChildB` do not have to be abstract, since they provide implementations for the parent class's abstract method. Class `ChildC` on line 17 does have to be abstract, since it does not provide an implementation for its parent's abstract method.

static

Data and methods may be declared static. Static features belong to the class in which they are declared, rather than belonging to individual instances of that class.

A class's static variable is allocated and initialized when the owning class is loaded. A class's static variable may be referenced via a reference to any instance of the owning class, or via the name of the class itself. For example, suppose class `C` has a static variable `v`. If `Cref1` and `Cref2` are references to instances of class `C`, then the static variable can be referenced as `C.v`, `Cref1.v`, or `Cref2.v`.

A method that is declared static must observe the following restrictions:

- The static method may only access those variables of its owning class that are declared static; the class's nonstatic variables may not be accessed.
- The static method may only call those methods of its owning class that are declared static; the class's nonstatic methods may not be called.
- The static method has no `this` reference.
- The static method may not be overridden.

It is legal for a class to contain static code that does not exist within a method body. Such code is known as *static initializer* code; it is executed when the owning class is loaded, after static variables are allocated and initialized. Static initializer code has no `this` reference. The code listed below illustrates a static initializer:

```
public class StaticDemo {
static int i=5;

  static { i++; }

  public static void main(String[] args) {
    System.out.println("i = " + i);
  }
}
```

When the application is started, the `StaticDemo` class is loaded. During the class loading process, the static variable `i` is allocated and initialized to 5. Later in the class loading process, `i` is incremented to 6. When the `main()` method executes, it prints out "i = 6".

synchronized

The `synchronized` modifier applies only to code. The modifier requires a thread to acquire a lock before executing the code. This topic is covered in Chapter 7, "Threads."

transient, native, volatile

The keywords `transient`, `native`, and `volatile` modify features in ways that are beyond the scope of the Programmer's Exam. They are covered briefly here, because the exam only requires you to be aware of their syntax.

The `transient` modifier applies only to class variables. During serialization, an object's transient variables are not serialized.

A native method calls code in a library specific to the underlying hardware. A native method is like an abstract method in the sense that the implementation exists somewhere other than the class in which the method is declared. The following code illustrates the syntax of a native method; note the semicolon in the place where an ordinary method would have a method body enclosed in curly brackets:

```
native int callnat(char c, String s);
```

The `volatile` modifier applies only to class variables. Volatile data is protected from certain kinds of corruption under multithreaded conditions.

Exam Essentials

Understand the four access modes and the corresponding keywords. You should know the significance of public, default, protected, and private access when applied to data, methods, and inner classes.

Understand how Java classes are organized into packages, so that you can understand the default and protected modes. A package is a namespace containing classes. You should know how the default and protected modes grant access to classes within the same package.

Know the effect of declaring a final class, variable, or method. A final class cannot be subclassed; a final variable cannot be modified after initialization; a final method cannot be overridden.

Know the effect of declaring an abstract class or method. An abstract class cannot be instantiated; an abstract method's definition is deferred to a subclass.

Understand the effect of declaring a static variable or method. Static variables belong to the class; static methods have no `this` pointer and may only access static variables and methods of their class.

Know how to reference a static variable or method. A static feature may be referenced through the class name or through a reference to any instance of the class.

Be able to recognize static initializer code. Static initializer code appears in curly brackets with no method declaration. Such code is executed once, when the class is loaded.

Key Terms and Concepts

Abstract class An abstract class may not be instantiated.

Abstract method An abstract method contains no body, deferring definition to non-abstract subclasses.

Default class A class with default access may be accessed by any class in the same package as the default class.

Default inner class A default inner class may be accessed from the enclosing class and from subclasses of the enclosing class that reside in the same package as the enclosing class.

Default method A method with default access may be called by any class in the same package as the class that owns the default method.

Default variable A variable with default access may be read and written by any class in the same package as the class that owns the default variable.

Final class A final class may not be subclassed.

Final method A final method may not be overridden.

Final variable A final variable, once initialized, may not be modified.

Private inner class A private inner class may only be accessed from the enclosing class.

Private method A private method may only be called by an instance of the class that owns the method.

Private variable A private variable may only be accessed by an instance of the class that owns the variable.

Protected inner class A protected inner class may be accessed by the enclosing class and by any subclass of the enclosing class.

Protected variable and protected method Protected access expands on default access by allowing any subclass to read and write protected data and to call protected methods, even if the subclass is in a different package from its superclass.

Public class, protected variable, and protected method Public access makes a feature accessible to all classes without restriction.

Public inner class A public inner class has the same access as a protected inner class and can be more manipulated by reflection.

Static initializer Static initializer code executes during class-load time, after static variables are allocated and initialized.

Static method A static method may only access the static variables and methods of its class.

Static variable Static data belongs to its class, rather than to any instance of the class. Static variables are allocated and initialized at class-load time.

Sample Questions

1. Which of the following are access modifiers?

A. `abstract`

B. `final`

C. `private`

D. `protected`

E. `public`

F. `static`

G. `synchronized`

Answer: C, D, and E. The other modifiers are not access modifiers. The fourth access mode, "default," has no corresponding modifier keyword.

2. If class `ClassY` has private method `z()`, can an instance of `ClassY` call the `z()` method of a different instance of `ClassY`?

Answer: Yes. A private feature may be accessed by any instance of the class that owns the feature.

3. Which access mode is more restricting: default or protected?

Answer: Default. Protected access is default plus some subclass access.

4. Can a static method write a nonstatic variable of the class that owns the static method?

   Answer: No. A static method can only access the static data and methods of its class.

5. Can a non-abstract class contain an abstract method?

   Answer: No. A class that contains any abstract methods must itself be declared abstract.

6. Which of the following statements are true?

   A. A final method may be overridden.

   B. A final method may not be overridden.

   C. A final class may be subclassed.

   D. A final class may not be subclassed.

   Answer: B and D. Final methods may not be overridden, and final classes may not be subclassed.

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

This objective requires you to know about Java's behind-the-scenes constructor behavior. This objective is important because most classes have to be instantiated to be useful, and instantiation means invoking a constructor.

Critical Information

A *default constructor* is a constructor with an empty argument list. For example, the default constructor for a class named `MyClass` would have the following format:

```
MyClass() { ... }
```

Every class must have at least one constructor. If you create a class that has no explicit constructors, then a default constructor is automatically generated by the compiler. In this case, the access mode of the constructor depends on the access mode of the class. If the class is public, the automatically generated constructor is also public. If the class is not public, the automatically generated constructor has default access. (Access is slightly different for inner classes, but this level of detail is not covered on the exam.)

Exam Essentials

Know that the compiler generates a default constructor when a class has no explicit constructors. When a class has constructor code, no default constructor is generated.

Know the access mode of the generated constructor. Public for public classes; default for classes with any other access mode.

Key Term and Concept

Default constructor A constructor with an empty argument list.

Sample Questions

1. What is the prototype for the automatically generated constructor of the following class?

   ```
   public class Kat extends Mammal { }
   ```

   Answer: `public Kat()`. Automatically generated constructors always have empty argument lists. For a public class such as this one, the automatically generated constructor is public.

2. Does the compiler automatically generate a constructor for the following class?

```
Class Kat extends Mammal {
  Float weight;
  Kat(float f) { weight = f; }
}
```

Answer: No. The class has an explicit constructor, so the compiler does not automatically generate a default constructor.

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

This objective concerns certain aspects of overloading and overriding. These are essential concepts in object-oriented programming, and the exam recognizes their importance in this objective and in the objectives covered in Chapter 6 ("Overloading, Overriding, Runtime Type, and Object Orientation").

You will be expected to know Java's rules for method overloading and overriding. The rule for access of overridden methods has already been discussed (in the section of this chapter that covers the long objective that begins "Declare classes…"). Here we will review the other factors that relate to overloading and overriding.

Critical Information

Overloading is reuse of a method name within a class. *Overriding* is reuse of a method name in a subclass.

To determine the legality of an attempt to overload or override a method, consider three things:

- The method's name

- The method's argument list
- The method's return type

Method Overloading

Overloaded methods share a common name but have different argument lists. Overloaded methods may have the same or different return types. Thus, within a single class the following methods may appear and are examples of legal overloading:

```
1. void aaa(int i, double d) { ... }
2. private String aaa(long z) { ... }
3. String aaa(long z, double d) { ... }
```

These three methods have the same name and different argument lists. The access modes and return types are irrelevant.

If a class contains the three methods shown previously, then it would be illegal to add the following method into that class:

```
4. double aaa(long z, double d) { ... }
```

The new method has the same name and argument list as the method on line 3, so it is illegal even though it has a different return type.

Method Overriding

When a method is overridden, the version in the subclass must match the name, argument list, and return type of the version in the superclass. Compilation fails if the subclass version has the same name and argument list as the superclass version but has a different return type.

Compilation succeeds if the subclass version has the same name as the superclass version but has a different argument list, whether or not the return type is different. However, this is not really method overriding; it is overloading of the method inherited from the superclass.

Exam Essentials

Know the legal return types for overloaded and overridden methods. There are no restrictions for an overloaded method; an overriding method must have the same return type as the overridden version.

Key Terms and Concepts

Overloading Reuse of a method name within a class. The methods must have different argument lists and/or return types.

Overriding Reuse of a method name in a subclass. The subclass version must have the same name, argument list, and return type as the superclass version.

Sample Questions

1. If two methods in a single class have the same name and different argument lists, can they have different return types?

   Answer: Yes. As long as the argument lists are different, the overloading is legal.

2. Is it legal for two methods in a single class to have the same name and the same argument list, but different return types?

   Answer: No. Two methods in a class may not have the same name and argument list.

3. When is it legal for a method to have the same name and argument list as a method in the superclass?

   Answer: If the two methods have the same return type, then we have an example of legal overriding. If the return types are different, the code will not compile.