Appendix C

Annotated Answers to Review Questions

1 Basics of Java Programming

1.1 *(d)*

A method is an operation defining the behavior for a particular abstraction. Java implements abstractions using classes that have properties and behavior. Behavior is defined by the operations of the abstraction.

1.2 *(b)*

An object is an instance of a class. Objects are created from classes that implement abstractions. The objects that are created are concrete realizations of those abstractions. An object is neither a reference nor a variable.

1.3 *(b)*

(2) is the first line of a constructor declaration. A constructor in Java is declared like a method, but does not specify a return value. (1) is the header of a class declaration, (3) is the first statement in the constructor body, and (4), (5) and (6) are instance method declarations.

1.4 *(b) and (f)*

Two objects and three references are created by the code. Objects are normally created by using the `new` operator. The declaration of a reference creates a variable regardless of whether a reference value is assigned to it.

1.5 *(d)*

An instance member is a field or an instance method. These members belong to an instance of the class rather than to the class as a whole. Members that are not explicitly declared as `static` in a class declaration are instance members.

1.6 *(c)*

An object communicates with another object by calling an instance method of the other object.

1.7 (d) and (f)

Given the declaration `class B extends A {...}`, we can conclude that class `B` extends class `A`, class `A` is the superclass of class `B`, class `B` is a subclass of class `A`, and class `B` inherits from class `A`, which means that objects of class `B` will inherit the field `value1` from class `A`.

1.8 (b), (d), and (g)

A `Train` object can share both the `TrainDriver` and its `Carriage` objects with other `Train` objects, when it is not using them. In other words, they can outlive the `Train` object. This is an example of aggregation. However, a `Train` object owns the array object used for handling its carriages. The lifetime of an array object is nested in the lifetime of its `Train` object. This is an example of composition.

1.9 (d)

The compiler supplied with the JDK is named `javac`. The names of the source files to be compiled are listed on the command line after the command `javac`.

1.10 (a)

Java programs are executed by the Java Virtual Machine (JVM). In the JDK, the command `java` is used to start the execution by the JVM. The `java` command requires the name of a class that has a valid `main()` method. The JVM starts the program execution by calling the `main()` method of the given class. The exact name of the class should be specified, rather than the name of the class file; that is, the `.class` extension in the class file name should not be specified.

1.11 (e)

(a): The JVM must be compatible with the Java Platform on which the program was developed.

(b): The JIT feature of the JVM translates bytecode to machine code.

(c): Other languages, such as Scala, also compile to bytecode and can be executed by a JVM.

(d): A Java program can only create objects; destroying objects occurs at the discretion of the automatic garbage collector.

2 Language Fundamentals

2.1 (c)

`52pickup` is not a legal identifier. The first character of an identifier cannot be a digit. An underscore is treated as a letter in identifier names.

2.2 (b), (c), (d), and (f)

In (b), the underscore is not between digits. In (c), digit 9 is not valid in an octal literal. In (d), the underscore is not between digits. In (f), there is no such escape sequence.

2.3 (e)

In Java, the identifiers `delete`, `thrown`, `exit`, `unsigned`, and `next` are not keywords. Java has a `goto` keyword, but it is reserved and not currently used.

2.4 (e)

Everything from the start sequence (`/*`) of a multiple-line comment to the first occurrence of the end sequence (`*/`) of a multiple-line comment is ignored by the compiler. Everything from the start sequence (`//`) of a single-line comment to the end of the line is ignored by the compiler. In (e), the multiple-line comment ends with the first occurrence of the end sequence (`*/`), leaving the second occurrence of the end sequence (`*/`) unmatched.

2.5 (a) and (d)

`String` is a class, and `"hello"` and `"t"` denote `String` objects. Java has the following primitive data types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`.

2.6 (a), (c), and (e)

(a) is a `boolean` data type, while (c) and (e) are floating-point data types.

2.7 (c)

The bit representation of `int` is 32 bits wide and can hold values in the range -2^{31} through $2^{31} - 1$.

2.8 (a), (c), and (d)

The \u*xxxx* notation can be used anywhere in the source to represent Unicode characters.

2.9 (c)

Local variable `i` is declared but not initialized. The first line of code declares the local variables `i` and `j`. The second line of code initializes the local variable `j`. Local variable `i` remains uninitialized.

2.10 (c)

The local variable of type `float` will remain uninitialized. Fields and static variables are initialized with a default value. An instance variable of type `int[]` is a reference variable that will be initialized with the `null` value. Local variables remain uninitialized unless explicitly initialized. The type of the variable does not affect whether a variable is initialized.

2.11 (e)

The program will compile. The compiler can figure out that the local variable `price` will always be initialized, since the value of the condition in the `if` statement is `true`. The two instance variables and the two static variables are all initialized to the respective default value of their type.

3 Declarations

3.1 *(b)*

Only (b) is a valid method declaration. Methods must specify a return type or must be declared as `void`. This makes (d) and (e) invalid. Methods must specify a list of zero or more comma-separated parameters enclosed by parentheses, `()`. The keyword `void` cannot be used to specify an empty parameter list. This makes (a) and (c) invalid.

3.2 *(a), (b), and (e)*

Non-static methods have an implicit `this` object reference. The `this` reference cannot be changed, as in (c). The `this` reference can be used in a non-static context to refer to both instance and static members. However, it cannot be used to refer to local variables, as in (d).

3.3 *(a) and (e)*

The first and third pairs of methods will compile. The second pair of methods will not compile, since their method signatures do not differ. The compiler has no way of differentiating between the two methods. Note that the return type and the names of the parameters are not a part of the method signature. Both methods in the first pair are named `fly` and have different numbers of parameters, thus overloading this method name. The methods in the last pair do not overload the method name `glide`, since only one method has that name. The method named `Glide` is distinct from the method named `glide`, as identifiers are case sensitive in Java.

3.4 *(a)*

A constructor cannot specify any return type, not even `void`. A constructor cannot be `final`, `static`, or `abstract`.

3.5 *(b) and (e)*

A constructor can be declared as `private`, but this means that this constructor can be used only within the class. Constructors need not initialize all the fields when a class is instantiated. A field will be assigned a default value if not explicitly initialized. A constructor is non-static and, as such, it can directly access both the static and non-static members of the class.

3.6 *(c)*

A compile-time error will occur at (3), since the class does not have a constructor accepting a single argument of type `int`. The declaration at (1) declares a method, not a constructor, since it is declared as `void`. The method happens to have the same name as the class, but that is irrelevant. The class has a default constructor, since the class contains no constructor declarations. This constructor will be invoked to create a `MyClass` object at (2).

3.7 *(d)*

In Java, arrays are objects. Each array object has a `public final` field named `length` that stores the size of the array.

3.8 (a)

Java allows arrays of length zero. Such an array is passed as an argument to the `main()` method when a Java program is run without any program arguments.

3.9 (c)

The `[]` notation can be placed both after the type name and after the variable name in an array declaration. Multidimensional arrays are created by constructing arrays that can contain references to other arrays. The expression `new int[4][]` will create an array of length 4, which can contain references to arrays of `int` values. The expression `new int[4][4]` will also create a two-dimensional array, but will in addition create four more one-dimensional arrays, each of length 4 and of the type `int[]`. References to each of these arrays are stored in the two-dimensional array. The expression `int[][4]` will not work, because the arrays for the dimensions must be created from left to right.

3.10 (b) and (e)

The size of the array cannot be specified, as in (b) and (e). The size of the array is given implicitly by the initialization code. The size of the array is never specified in the declaration of an array reference. The size of an array is always associated with the array instance (on the right-hand side), not the array reference (on the left-hand side).

3.11 (e)

The array declaration is valid, and will declare and initialize an array of length 20 containing `int` values. All the values of the array are initialized to their default value of 0. The `for(;;)` loop will print all the values in the array; that is, it will print `0` twenty times.

3.12 (d)

The program will print `0 false 0 null` at runtime. All the instance variables, and the array element, will be initialized to their default values. When concatenated with a string, the values are converted to their string representation. Notice that the `null` pointer is converted to the string `"null"`, rather than throwing a `NullPointerException`.

3.13 (b)

Evaluation of the actual parameter `i++` yields 0, and increments `i` to 1 in the process. The value 0 is copied into the formal parameter `i` of the method `addTwo()` during method invocation. However, the formal parameter is local to the method, and changing its value does not affect the value in the actual parameter. The value of the variable `i` in the `main()` method remains 1.

3.14 (d)

The variables `a` and `b` are local variables that contain primitive values. When these variables are passed as arguments to another method, the method receives copies of the primitive values in the variables. The actual variables are unaffected by operations performed on the copies of the primitive values within the called method. The variable `bArr` contains a reference value that denotes an array object

containing primitive values. When the variable is passed as a parameter to another method, the method receives a copy of the reference value. Using this reference value, the method can manipulate the object that the reference value denotes. This allows the elements in the array object referenced by `bArr` to be accessed and modified in the method `inc2()`.

3.15 *(a) and (f)*

A value can be assigned to a `final` variable only once. A `final` formal parameter is assigned the value of the actual parameter at method invocation. Within the method body, it is illegal to reassign or modify the value stored in a `final` parameter. This causes `a++` and `c = d` to fail. Whether the actual parameter is `final` does not constrain the client that invoked the method, since the actual parameter values are assigned to the formal parameters.

3.16 *(a), (d), and (f)*

The ellipses (`...`) must be specified before the parameter name. Only one variable arity parameter is permitted, and it must be the last parameter in the formal parameter list.

3.17 *(c)*

In (a) and (b), the arguments are encapsulated as elements in the implicitly created array that is passed to the method. In (c), the `int` array object itself is encapsulated as an element in the implicitly created array that is passed to the method. (a), (b) and (c) are fixed arity calls. Note that `int[]` is not a subtype of `Object[]`. In (d), (e), and (f), the argument is a subtype of `Object[]`, and the argument itself is passed without the need of an implicitly created array; that is, these are fixed arity method calls. However, in (d) and (e), the compiler issues a warning that both fixed arity and variable arity method calls are feasible, but chooses fixed arity method calls.

4 Access Control

4.1 *(a) and (c)*

Bytecode of all reference type declarations in the file is placed in the designated package, and all reference type declarations in the file can access the imported types.

4.2 *(e)*

Both classes are in the same package `app`, so the first two import statements are unnecessary. The package `java.lang` is always imported in all compilation units, so the next two import statements are unnecessary. The last static import statement is necessary to access the static variable `frame` in the `Window` class by its simple name.

4.3 *(b), (c), (d), and (e)*

(a): The import statement imports types from the `mainpkg` package, but `Window` is not one of them.

(b): The import statement imports types from the `mainpkg.subpkg1` package, and `Window` is one of them.

(c): The import statement imports types from the `mainpkg.subpkg2` package, and `Window` is one of them.

(d): The first import statement is type-import-on-demand and the second import statement is single-type-import. Both import the type `Window`. The second overrides the first one.

(e): The first import statement is single-type-import and the second import statement is type-import-on-demand. Both import the type `Window`. The first overrides the second one.

(f): Both import statements import the type `Window`, making the import ambiguous.

(g): Both single-type-import statements import the type `Window`. The second import statement causes a conflict with the first.

4.4 (c) and (e)

The name of the class must be fully qualified. A parameter list after the method name is not permitted. (c) illustrates single static import and (e) illustrates static import on demand.

4.5 (b), (d), and (f)

In (a), the file `A.class` will be placed in the same directory as the file `A.java`. There is no `-D` option for the `javac` command, as in (c). The compiler maps the package structure to the file system, creating the necessary (sub)directories.

4.6 (b) and (d)

In (a) and (c), class `A` cannot be found. In (e) and (f), class `B` cannot be found—there is no package under the current directory `/top/wrk/pkg` to search for class `B`. Note that specifying `pkg` in the classpath in (d) is superfluous. The *parent* directory of the package must be specified, meaning the *location* of the package.

4.7 (d) and (f)

The *parent* directory (or *location*) of the package must be specified. Only (d) and (f) do that. (d) specifies the current directory first, but there is no file `top/sub/A.class` under the current directory. Searching under `../bin` (i.e., `/proj/bin`) finds the file `top/sub/A.class`.

4.8 (c) and (d)

A class or interface name can be referred to by using either its fully qualified name or its simple name. Using the fully qualified name will always work, but to use the simple name it has to be imported. When `net.basemaster.*` is imported, all the type names from the package `net.basemaster` will be imported and can now be referred to using simple names. Importing `net.*` will not import the subpackage `basemaster`.

4.9 (c)

Any non-final class can be declared as `abstract`. A class cannot be instantiated if the class is declared as `abstract`. The declaration of an `abstract` method cannot provide an implementation. The declaration of a non-`abstract` method must provide an implementation. If any method in a class is declared as `abstract`, then the class must be declared as `abstract`, so (a) is invalid. The declaration in (b) is not valid, since it omits the keyword `abstract` in the method declaration. The declaration in

(d) is not valid, since it omits the keyword `class`. In (e), the return type of the method is missing.

4.10 (e)

Only a `final` class cannot be extended, as in (d). (c) and (e) will also not compile. The keyword `native` can be used only for methods, not for classes and fields. A class cannot be declared as both `final` and `abstract`.

4.11 (b)

Outside the package, the member `j` is accessible to any class, whereas the member `k` is accessible only to subclasses of `MyClass`.

The field `i` has package accessibility, and is accessible to only classes inside the package. The field `j` has public accessibility, and is accessible from anywhere. The field `k` has protected accessibility, and is accessible from any class inside the package and from subclasses anywhere. The field `l` has private accessibility, and is accessible only within its own class.

4.12 (c)

The default accessibility for members is more restrictive than protected accessibility, but less restrictive than private accessibility. Members with default accessibility are accessible only within the class itself and from classes in the same package. Protected members are, in addition, accessible from subclasses anywhere. Members with private accessibility are accessible only within the class itself.

4.13 (b)

A private member is accessible only within the class of the member. If no accessibility modifier has been specified for a member, the member has default accessibility, also known as package accessibility. The keyword `default` is not an accessibility modifier. A member with package accessibility is accessible only from classes in the same package. Subclasses in other packages cannot access a member with default accessibility.

4.14 (a), (c), (d), (e), and (h)

The lines (1), (3), (4), (5), and (8) will compile. A `protected` member of a superclass is always inherited by a subclass. Direct access to the `protected` field `pf` is permitted in subclasses `B` and `C` at (1) and (5), respectively.

A subclass in another package can access protected members in the superclass only via references of its own type or its subtypes. In `packageB`, the subclass `B` can access the protected field `pf` in the superclass `packageA.A` via references of type `B` (i.e., parameter `obj2`) and references of its subclass `C` (i.e., parameter `obj3`). However, the subclass `C` can access the protected field `pf` in the superclass `packageA.A` only via references of type `C` (i.e., parameter `obj3`). This is the case at (3), (4), and (8).

The class `D` does not have any inheritance relationship with any of the other classes, and therefore the `protected` field `pf` is not accessible in the class `D`. This rules out the lines from (9) to (12).

4.15 (b) and (e)

If no accessibility modifier (`public`, `protected`, or `private`) is given in the member declaration of a class, the member is accessible only to classes in the same package.

A subclass does not have access to members with default accessibility declared in a superclass, unless they are in the same package.

Local variables cannot be declared as `static` or have an accessibility modifier.

4.16 (c)

Line (3) `void k() { i++; }` can be reinserted without introducing errors. Reinserting line (1) will cause the compilation to fail, since `MyOtherClass` will try to override a `final` method. Reinserting line (2) will fail, since `MyOtherClass` will no longer have the (no-argument) default constructor. The `main()` method needs to call the no-argument constructor. Reinserting line (3) will work without any problems, but reinserting line (4) will fail, since the method will try to access a `private` member of the superclass.

4.17 (b)

The keyword `this` can be used only in non-static code, as in non-static methods, constructors, and instance initializer blocks. Only one occurrence of each static variable of a class is created, when the class is loaded by the JVM. This occurrence is shared among all the objects of the class (and for that matter, by other clients). Local variables are accessible only within the block scope, regardless of whether the block scope is defined within a static context.

4.18 (c)

The declaration in (c) is not legal, as variables cannot be declared as `abstract`. The keywords `static` and `final` are valid modifiers for both field and method declarations. The modifiers `abstract` and `native` are valid for methods, but not together. They cannot be specified for fields.

4.19 (a) and (c)

Abstract classes can declare both `final` methods and non-`abstract` methods. Non-`abstract` classes cannot, however, contain `abstract` methods. Nor can `abstract` classes be `final`. Only methods can be declared `native`.

4.20 (a)

The keyword `transient` signifies that the fields should not be stored when objects are serialized. Constructors cannot be declared as `abstract`. When an array object is created, as in (c), the elements in the array object are assigned the default value corresponding to the type of the elements. Whether the reference variable denoting the array object is a local variable or a member variable is irrelevant. Abstract methods from a superclass need not be implemented by a subclass, but the subclass must then be declared as `abstract`. Static methods can also be accessed in a non-static context—for example, in instance methods, constructors, and instance initializer blocks.

5 Operators and Expressions

5.1 (a)

A value of type `char` can be assigned to a variable of type `int`. A widening conversion will convert the value to an `int`.

5.2 (d)

An assignment statement is an expression statement. The value of the expression statement is the value of the expression on the right-hand side. Since the assignment operator is right associative, the statement `a = b = c = 20` is evaluated as follows: `(a = (b = (c = 20)))`. This results in the value 20 being assigned to `c`, then the same value being assigned to `b` and finally to `a`. The program will compile, and print `20` at runtime.

5.3 (c)

Strings are objects. The variables `a`, `b`, and `c` are references that can denote such objects. Assigning to a reference only changes the reference value; it does not create a copy of the source object or change the object denoted by the old reference value in the target reference. In other words, assignment to references affects only which object the target reference denotes. The reference value of the `"cat"` object is first assigned to `a`, then to `b`, and later to `c`. The program prints the string denoted by `c`, `"cat"`. The local `final String` variable `b` is initialized only once in the code.

5.4 (a), (d), and (e)

A binary expression with any floating-point operand will be evaluated using floating-point arithmetic. Expressions such as `2/3`, where both operands are integers, will use integer arithmetic and evaluate to an integer value. In (e), the result of `(0x10 * 1L)` is promoted to a floating-point value.

5.5 (b)

The `/` operator has higher precedence than the `+` operator. This means that the expression is evaluated as `((1/2) + (3/2) + 0.1)`. The associativity of the binary operators is from left to right, giving `(((1/2) + (3/2)) + 0.1)`. Integer division results in `((0 + 1) + 0.1)`, which evaluates to 1.1.

5.6 (e)

`0x10` is a hexadecimal literal equivalent to the decimal value 16. `10` is a decimal literal. `010` is an octal literal equivalent to the decimal value 8. `0b10` is a binary literal equivalent to the decimal value 2. The `println()` method will print the sum of these values, which is 36, in decimal form.

5.7 (b), (c), and (f)

The unary `+` and `-` operators with right associativity are used in the valid expressions (b), (c), and (f). Expression (a) tries to use a nonexistent unary `-` operator with left associativity, expression (d) tries to use a decrement operator (`--`) on an expression that does not resolve to a variable, and expression (e) tries to use a nonexistent unary `*` operator. (c) compiles because the unary operators cannot be interpreted as increment (`++`) or decrement (`--`) operators: `(+(-(+(-(+(-1))))))`.

5.8 (b)

The expression evaluates to –6. The whole expression is evaluated as `(((-(-1)) - ((3 * 10) / 5)) - 1)` according to the precedence and associativity rules.

5.9 (a), (b), (d), and (e)

In (a), the conditions for implicit narrowing conversion are fulfilled: The source is a constant expression of type `int`, the destination type is of type `short`, and the value of the source (12) is in the range of the destination type. The assignments in (b), (d), and (e) are valid, since the source type is narrower than the target type and an implicit widening conversion will be applied. The expression (c) is not valid. Values of type `boolean` cannot be converted to other types.

5.10 (a), (c), and (d)

The left associativity of the `+` operator makes the evaluation of `(1 + 2 + "3")` proceed as follows: `(1 + 2) + "3"` → `3 + "3"` → `"33"`. Evaluation of the expression `("1" + 2 + 3)`, however, will proceed as follows: `("1" + 2) + 3` → `"12" + 3` → `"123"`. `(4 + 1.0f)` evaluates as `4.0f + 1.0f` → `5.0f` and `(10/9)` performs integer division, resulting in the value 1. The operand `'a'` in the expression `('a' + 1)` will be promoted to `int`, and the resulting value will be of type `int`.

5.11 (d)

The expression `++k + k++ + + k` is evaluated as `((++k) + (k++)) + (+k)` → `((2) + (2) + (3))`, resulting in the value `7`.

5.12 (d)

The types `char` and `int` are both integral. A `char` value can be assigned to an `int` variable since the `int` type is wider than the `char` type and an implicit widening conversion will be done. An `int` type cannot be assigned to a `char` variable because the `char` type is narrower than the `int` type. The compiler will report an error about a possible loss of precision in (4).

5.13 (c)

Variables of the type `byte` can store values in the range –128 to 127. The expression on the right-hand side of the first assignment is the `int` literal 128. Had this literal been in the range of the `byte` type, an implicit narrowing conversion would have been applied to convert it to a `byte` value during assignment. Since 128 is outside the range of the type `byte`, the program will not compile.

5.14 (a)

First, the expression `++i` is evaluated, resulting in the value 2. Now the variable `i` also has the value 2. The target of the assignment is now determined to be the element `array[2]`. Evaluation of the right-hand expression, `--i`, results in the value 1. The variable `i` now has the value 1. The value of the right-hand expression, 1, is then assigned to the array element `array[2]`, causing the array contents to become `{4, 8, 1}`. The program computes and prints the sum of these values, `13`.

5.15 (a) and (c)

In (a) and (e), both operands are evaluated, with (a) yielding `true`, but (e) yielding `false`. The `null` literal can be compared, so `(null != null)` yields `false`. The expression `(4 <= 4)` is `true`. `(!true)` is `false`.

5.16 (c) and (e)

The remainder operator `%` is not limited to integral values, but can also be applied to floating-point operands. Short-circuit evaluation occurs only with the conditional operators (`&&`, `||`). The operators `*`, `/`, and `%` have the same level of precedence. The data type `short` is a 16-bit signed two's complement integer, so the range of values is from `-32768` to `+32767`, inclusive. `(+15)` is a legal expression using the unary `+` operator.

5.17 (a), (c), and (e)

The `!=` and `^` operators, when used on boolean operands, will return `true` if and only if one operand is `true`, and `false` otherwise. This means that `d` and `e` in the program will always be assigned the same value, given any combination of truth values in `a` and `b`. The program will, therefore, print `true` four times.

5.18 (b)

The element referenced by `a[i]` is determined based on the current value of `i`, which is 0—that is, the element `a[0]`. The expression `i = 9` will evaluate to the value 9, which will be assigned to the variable `i`. The value 9 is also assigned to the array element `a[0]`. After the execution of the statement, the variable `i` will contain the value 9, and the array `a` will contain the values 9 and 6. The program will print `9 9 6` at runtime.

5.19 (c) and (d)

Note that the logical and conditional operators have lower precedence than the relational operators. Unlike the `&` and `|` operators, the `&&` and `||` operators short-circuit the evaluation of their operands if the result of the operation can be determined from the value of the first operand. The second operand of the `||` operator in the program is never evaluated because of short-circuiting. All the operands of the other operators are evaluated. Variable `i` ends up with the value 3, which is the first digit printed, and `j` ends up with the value 1, which is the second digit printed.

5.20 (d) and (f)

`&&=` and `%%` are not operators in Java. The operators `%`, `&&`, `%=` , `<=`, and `->` are called remainder, conditional AND, remainder compound assignment, relational less than or equal, and arrow operator, respectively.

5.21 (c), (e), and (f)

In (a), the third operand has the type `double`, which is not assignment compatible with the type `int` of the variable `result1`. Blocks are not legal operands in the conditional operator, as in (b). In (c), the last two operands result in wrapper objects with type `Integer` and `Double`, respectively, which are assignment compatible with the type `Number` of the variable `number`. The evaluation of the conditional expression results in the reference value of an `Integer` object, with value `20` being assigned to

the `number` variable. All three operands of the operator are mandatory, which is not the case in (d). In (e), the last two operands are of type `int`, and the evaluation of the conditional expression results in an `int` value (`21`), whose string representation is printed. In (f), the value of the second operand is boxed into a `Boolean`. The evaluation of the conditional expression results in a string literal (`"i not equal to j"`), which is printed. The `println()` method creates and prints a string representation of any object whose reference value is passed as parameter.

5.22 *(d)*

The condition in the outer conditional expression is `false`. The condition in the nested conditional expression is `true`, resulting in the value of `m1` (i.e., 20) being printed.

6 Control Flow

6.1 *(d)*

The program will display the letter `b` when run. The second `if` statement is evaluated since the boolean expression of the first `if` statement is `true`. The `else` clause belongs to the second `if` statement. Since the boolean expression of the second `if` statement is `false`, the `if` block is skipped and the `else` clause is executed.

6.2 *(a), (b), and (e)*

The condition of an `if` statement can be any expression, including method calls, as long as it evaluates or can be unboxed to a value of type `boolean`. The expression `(a = b)` does not compare the variables `a` and `b`, but assigns the value of `b` to the variable `a`. The result of the expression is the value being assigned. Since `a` and `b` are either `boolean` or `Boolean` variables, the value returned by the expression is also either `boolean` or `Boolean`. This allows the expression to be used as the condition for an `if` statement. An `if` statement must always have an `if` block, but the `else` clause is optional. The expression `if (false) ; else ;` is legal. In this case, both the `if` block and the `else` block are simply the empty statement.

6.3 *(f)*

There is nothing wrong with the code. The `case` and `default` labels do not have to be specified in any specific order. The use of the `break` statement is not mandatory, and without it the control flow will simply fall through the labels of the `switch` statement.

6.4 *(c)*

The `case` label value `2 * iLoc` is a constant expression whose value is 6, the same as the `switch` expression. Fall-through results in the program output shown in (c).

6.5 *(b)*

The `switch` expression, when unboxed, has the value 5. The statement associated with the `default` label is executed, and the fall-through continues until the `break` statement.

6.6 (a), (b), (f), and (j)

In (a), (b), (f), and (j), the string expression involves constant values and evaluates to `"TomTom"`. Program output is `"Hi, TomTom!"` In (i), the constant string expression evaluates to `"304Tom"` (`84+111+109+"Tom"`). The first three literals are of type `char`, and their `int` values are added before being concatenated with last `String` operand. Program output is `"Whatever!"` In (c), (d), (e), (g), and (h), the `case` label is not a constant string expression, and the program will not compile.

6.7 (e)

The loop body is executed twice and the program will print `3`. The first time the loop is executed, the variable `i` changes from 1 to 2 and the variable `b` changes from `false` to `true`. Then the loop condition is evaluated. Since `b` is `true`, the loop body is executed again. This time the variable `i` changes from 2 to 3 and the variable `b` changes from `true` to `false`. The loop condition is then evaluated again. Since `b` is now `false`, the loop terminates and the current value of `i` is printed.

6.8 (b) and (e)

Both the first and second numbers printed will be `10`. Both the loop body and the update expression will be executed exactly 10 times. Each execution of the loop body will be directly followed by an execution of the update expression. Afterward, the condition `j < 10` is evaluated to see whether the loop body should be executed again.

6.9 (c)

Only (c) contains a valid `for` loop. The initialization in a `for(;;)` statement can contain either declarations or a list of expression statements, but not both as attempted in (a). The loop condition must be of type `boolean`. (b) tries to use an assignment of an `int` value (notice the use of `=` rather than `==`) as a loop condition and, therefore, is not valid. The loop condition in the `for` loop (d) tries to use the uninitialized variable `i`, and the `for(;;)` loop in (e) is syntactically invalid, as there is only one semicolon.

6.10 (f)

The code will compile without error, but will never terminate when run. All the sections in the `for` header are optional and can be omitted (but not the semicolons). An omitted loop condition is interpreted as being `true`. Thus, a `for(;;)` loop with an omitted loop condition will never terminate, unless an appropriate control transfer statement is encountered in the loop body. The program will enter an infinite loop at (4).

6.11 (b), (d), and (e)

The loop condition in a `while` statement is not optional. It is missing in (a). It is not possible to break out of the `if` statement in (c). Notice that if this `if` statement had been placed within a `switch` statement or a loop, the usage of `break` would be valid. Inside a labeled block, a labeled `break` statement would be required.

6.12 (a) and (d)

"`i=1, j=0`" and "`i=2, j=1`" are part of the output. The variable `i` iterates through the values 0, 1, and 2 in the outer loop, while `j` toggles between the values 0 and 1 in the inner loop. If the values of `i` and `j` are equal, the printing of the values is skipped and the execution continues with the next iteration of the outer loop. The following can be deduced when the program is run: Variables `i` and `j` are both 0 and the execution continues with the update expression of the outer loop. "`i=1, j=0`" is printed and the next iteration of the inner loop starts. Variables `i` and `j` are both 1 and the execution continues with the update expression of the outer loop. "`i=2, j=0`" is printed and the next iteration of the inner loop starts. "`i=2, j=1`" is printed, `j` is incremented, `j < 2` is `false`, and the inner loop ends. Variable `i` is incremented, `i < 3` is `false`, and the outer loop ends.

6.13 (b)

The code will fail to compile, since the condition of the `if` statement is not of type `boolean`. The variable `i` is of type `int`. There is no conversion between `boolean` and other primitive types.

6.14 (c) and (d)

The element type of the array `nums` must be assignment compatible with the type of the loop variable, `int`. Only the element type in (c), `Integer`, can be automatically unboxed to an `int`. The element type in (d) is `int`.

6.15 (d) and (e)

In the header of a `for(:)` loop, we can declare only one local variable. This rules out (a) and (b), as they specify two local variables. Also the array expression in (a), (b), and (c) is not valid. Only (d) and (e) specify a legal `for(:)` header.

6.16 (d)

The program will print `1`, `4`, and `5`, in that order. The expression `5/k` will throw an `ArithmeticException`, since `k` equals 0. Control is transferred to the first `catch` clause, since it is the first clause that can handle the arithmetic exceptions. This exception handler simply prints `1`. The exception has now been caught and normal execution can resume. Before leaving the `try` statement, the `finally` clause is executed. This clause prints `4`. The last statement of the `main()` method prints `5`.

6.17 (b) and (e)

If run with no arguments, the program will print `The end`. If run with one argument, the program will print the given argument followed by "`The end`". The `finally` clause will always be executed, no matter how control leaves the `try` block.

6.18 (c) and (d)

Normal execution will resume only if the exception is caught by the method. The uncaught exception will propagate up the JVM stack until some method handles it. An overriding method need simply declare that it can throw a subset of the checked exceptions that the overridden method can throw. The `main()` method can declare that it throws checked exceptions just like any other method. The `finally` clause will always be executed, no matter how control leaves the `try` block.

6.19 (a)

The program will print `2` and throw an `InterruptedException`. An `InterruptedException` is thrown in the `try` block. There is no `catch` clause to handle the exception, so it will be sent to the caller of the `main()` method—that is, to the default exception handler. Before this happens, the `finally` clause is executed. The code to print `3` is never reached.

6.20 (b)

The only thing that is wrong with the code is the ordering of the `catch` and `finally` clauses. If present, the `finally` clause must always appear last in a `try-catch-finally` construct.

6.21 (a)

Overriding methods can specify all, none, or a subset of the checked exceptions that the overridden method declares in its `throws` clause. The `InterruptedException` is the only checked exception specified in the `throws` clause of the overridden method. The overriding method `compute()` need not specify the `InterruptedException` from the `throws` clause of the overridden method, because the exception is not thrown here.

7 Object-Oriented Programming

7.1 (a) and (b)

The `extends` clause is used to specify that a class extends another class. A subclass can be declared as `abstract` regardless of whether the superclass was declared as `abstract`. Private, overridden, and hidden members from the superclass are not inherited by the subclass. A class cannot be declared as both `abstract` and `final`, since an `abstract` class needs to be extended to be useful, and a `final` class cannot be extended. The accessibility of the class is not limited by the accessibility of its members. A class with all the members declared `private` can still be declared as `public`.

7.2 (b) and (e)

The `Object` class has a `public` method named `equals`, but it does not have any method named `length`. Since all classes are subclasses of the `Object` class, they all inherit the `equals()` method. Thus, all Java objects have a `public` method named `equals`. In Java, a class can extend only a single superclass, but there is no limit on how many subclasses can extend a superclass.

7.3 (a), (b), and (d)

`Bar` is a subclass of `Foo` that overrides the method `g()`. The statement `a.j = 5` is not legal, since the member `j` in the class `Bar` cannot be accessed through a `Foo` reference. The statement `b.i = 3` is not legal either, since the `private` member `i` cannot be accessed from outside of the class `Foo`.

7.4 (g)

It is not possible to invoke the `doIt()` method in `A` from an instance method in class `C`. The method in `C` needs to call a method in a superclass two levels up in the inher-

itance hierarchy. The `super.super.doIt()` strategy will not work, since `super` is a keyword and cannot be used as an ordinary reference, nor can it be accessed like a field. If the member to be accessed had been a field, the solution would be to cast the `this` reference to the class of the field and use the resulting reference to access the field. Field access is determined by the declared type of the reference, whereas the instance method to execute is determined by the actual type of the object denoted by the reference at runtime.

7.5 (e)

The code will compile without errors. None of the calls to a `max()` method are ambiguous. When the program is run, the `main()` method will call the `max()` method on the `C` object referred to by the reference `b` with the parameters `13` and `29`. This method will call the `max()` method in `B` with the parameters `23` and `39`. The `max()` method in `B` will in turn call the `max()` method in `A` with the parameters `39` and `23`. The `max()` method in `A` will return `39` to the `max()` method in `B`. The `max()` method in `B` will return `29` to the `max()` method in `C`. The `max()` method in `C` will return `29` to the `main()` method.

7.6 (c)

The simplest way to print the message in the class `Message` would be to use `msg.text`. The `main()` method creates an instance of `MyClass`, which results in the creation of a `Message` instance. The field `msg` denotes this `Message` object in `MySuperclass` and is inherited by the `MyClass` object, as this field has default accessibility. Thus, the message in the `Message` object can be accessed directly by `msg.text` in the `print()` method of `MyClass`, and also by `this.msg.text` and `super.msg.text`.

7.7 (g)

In the class `Car`, the static method `getModelName()` hides the static method of the same name in the superclass `Vehicle`. In the class `Car`, the instance method `getRegNo()` overrides the instance method of the same name in the superclass `Vehicle`. The declared type of the reference determines the method to execute when a static method is called, but the actual type of the object at runtime determines the method to execute when an overridden method is called.

7.8 (e)

The class `MySuper` does not have a no-argument constructor. This means that constructors in subclasses must explicitly call the superclass constructor and provide the required parameters. The supplied constructor accomplishes this by calling `super(num)` in its first statement. Additional constructors can accomplish this either by calling the superclass constructor directly using the `super()` call, or by calling another constructor in the same class using the `this()` call, which in turn calls the superclass constructor. (a) and (b) are not valid, since they do not call the superclass constructor explicitly. (d) fails, since the `super()` call must always be the first statement in the constructor body. (f) fails, since the `super()` and `this()` calls cannot be combined.

7.9 (b)

In a subclass without any declared constructors, the default constructor will call `super()`. The use of the `super()` and `this()` statements are not mandatory as long as the superclass has a default constructor. If neither `super()` nor `this()` is declared as the first statement in the body of a constructor, then the default `super()` will implicitly be the first statement. A constructor body cannot have both a `super()` and a `this()` statement. Calling `super()` will not always work, since a superclass might not have a default constructor.

7.10 (d)

The program will print `12` followed by `Test`. When the `main()` method is executed, it will create a new instance of `B` by passing `"Test"` as an argument. This results in a call to the constructor of `B`, which has one `String` parameter. The constructor does not explicitly call any superclass constructor or any overloaded constructor in `B` using a `this()` call; instead, the no-argument constructor of the superclass `A` is called implicitly. The no-argument constructor of `A` calls the constructor in `A` that has two `String` parameters, passing it the argument list `("1", "2")`. This constructor calls the constructor with one `String` parameter, passing the argument `"12"`. This constructor prints the argument, after implicitly invoking the no-argument constructor of the superclass `Object`. Now the execution of all the constructors in `A` is completed, and execution continues in the constructor of `B`. This constructor now prints the original argument `"Test"` and returns to the `main()` method.

7.11 (b) and (c)

Interface declarations do not provide any method implementations and permit only multiple interface inheritance. An interface can extend any number of interfaces and can be extended by any number of interfaces. Fields in interfaces are always `static`, and can be declared as `static` explicitly. Abstract method declarations in interfaces are always non-`static`, and cannot be declared static.

Interfaces allow only multiple interface inheritance. An interface can extend any number of interfaces, and can be extended by any number of interfaces. Fields in interfaces are always `static`, and can be declared as `static` explicitly. Static methods, of course, can be declared as `static`. Abstract method declarations in interfaces are always non-`static`, and cannot be declared as `static`.

7.12 (a), (d), (e), and (f)

The keywords `protected`, `private`, and `final` cannot be applied to interface methods. The keyword `public` is implied, but can be specified for all interface methods. The keywords `default`, `abstract`, and `static` can be specified for default, abstract, and static methods, respectively. The keywords `default` and `static` are required for default and static methods, respectively, but the keyword `abstract` is optional and is implicitly implied for abstract methods.

7.13 (a), (f), and (g)

Only the keywords `public`, `static`, and `final` are implicitly implied for interface variables.

7.14 (e)

(1): The final static constant is not initialized.

(2): The abstract method cannot have an implementation.

(3): The static method is missing the implementation.

(4): The default method cannot be final.

7.15 (b) and (c)

The default instance method `printSlogan()` is inherited by the class `Company`.

(a): It can be called from a non-static context (instance method `testSlogan()`) by its simple name, but not from a static context (static method `main()`).

(b), (c): An instance method can be invoked on an instance via a reference, regardless of whether it is in a static or non-static context.

(d), (e): An instance method cannot be invoked via a reference type, but only on an instance via a reference; that is, you cannot make a static reference to a non-static method.

7.16 (e)

The `static` method `printSlogan()` is *not* inherited by the class `Firm`. It can be invoked by using a static reference, the name of the interface in which it is declared, regardless of whether the call is in a static or a non-static context.

7.17 (c)

The instance method at (3) overrides the default method at (1). The static method at (2) is not inherited by the class `RaceA`. The instance method at (4) does not override the static method at (2).

The method to be invoked by the call at (5) is determined at runtime by the object type of the reference, which in this case is `Athlete`, resulting in the method at (3) being invoked. Similarly, the call at (6) will invoke the instance method at (4).

7.18 (a)

The program will not compile, because the overriding method at (2) cannot have narrower accessibility than the overridden method at (1). The method at (1) has `public` accessibility, whereas the method at (2) has package accessibility.

7.19 (a), (c), and (d)

Fields in interfaces declare named constants, and are always `public`, `static`, and `final`. None of these modifiers is mandatory in a constant declaration. All named constants must be explicitly initialized in the declaration.

7.20 (a) and (d)

The keyword `implements` is used when a class implements an interface. The keyword `extends` is used when an interface inherits from another interface or a class inherits from another class.

7.21 (d)

The code will compile without errors. The class `MyClass` declares that it implements the interfaces `Interface1` and `Interface2`. Since the class is declared as `abstract`, it does not need to implement all abstract method declarations defined in these

interfaces. Any non-`abstract` subclasses of `MyClass` must provide the missing method implementations. The two interfaces share a common abstract method declaration `void g()`. `MyClass` provides an implementation for this abstract method declaration that satisfies both `Interface1` and `Interface2`. Both interfaces provide declarations of constants named `VAL_B`. This can lead to ambiguity when referring to `VAL_B` by its simple name from `MyClass`. The ambiguity can be resolved by using the qualified names: `Interface1.VAL_B` and `Interface2.VAL_B`. However, there are no problems with the code as it stands.

7.22 (a) and (c)

Declaration (b) fails, since it contains an illegal forward reference to its own named constant. The type of the constant is missing in declaration (d). Declaration (e) tries (illegally) to use the `protected` modifier, even though named constants always have `public` accessibility. Such constants are implicitly `public`, `static`, and `final`.

7.23 (c)

The program will throw a `java.lang.ClassCastException` in the assignment at (3) at runtime. The statement at (1) will compile, since the assignment is done from a subclass reference to a superclass reference. The cast at (2) assures the compiler that `arrA` refers to an object that can be cast to type `B[]`. This will work when run, since `arrA` will refer to an object of type `B[]`. The cast at (3) assures the compiler that `arrA` refers to an object that can be cast to type `B[]`. This will not work when run, since `arrA` will refer to an object of type `A[]`.

7.24 (d) and (f)

(4) and (6) will cause a compile-time error, since an attempt is made to assign a reference value of a supertype object to a reference of a subtype. The type of the source reference value is `MyClass` and the type of the destination reference is `MySubclass`. (1) and (2) will compile, since the reference is assigned a reference value of the same type. (3) will also compile, since the reference is assigned a reference value of a subtype.

7.25 (e)

Only the assignment `I1 b = obj3` is valid. The assignment is allowed, since `C3` extends `C1`, which implements `I1`. The assignment `obj2 = obj1` is not legal, since `C1` is not a subclass of `C2`. The assignments `obj3 = obj1` and `obj3 = obj2` are not legal, since neither `C1` nor `C2` is a subclass of `C3`. The assignment `I1 a = obj2` is not legal, since `C2` does not implement `I1`. Assignment `I2 c = obj1` is not legal, since `C1` does not implement `I2`.

7.26 (b)

The compiler will allow the statement, as the cast is from the supertype (`Super`) to the subtype (`Sub`). However, if at runtime the reference `x` does not denote an object of the type `Sub`, a `ClassCastException` will be thrown.

7.27 (b)

The expression `(o instanceof B)` will return `true` if the object referred to by `o` is of type `B` or a subtype of `B`. The expression `(!(o instanceof C))` will return `true` unless

the object referred to by `o` is of type `C` or a subtype of `C`. Thus, the expression `(o instanceof B) && (!(o instanceof C))` will return `true` only if the object is of type `B` or a subtype of `B` that is not `C` or a subtype of `C`. Given objects of the classes `A`, `B`, and `C`, this expression will return `true` only for objects of class `B`.

7.28 (d)

The program will print all the letters `I`, `J`, `C`, and `D` at runtime. The object referred to by the reference `x` is of class `D`. Class `D` extends class `C` and implements `J`, and class `C` implements interface `I`. This makes `I`, `J`, and `C` supertypes of class `D`. The reference value of an object of class `D` can be assigned to any reference of its supertypes and, therefore, is an `instanceof` these types.

7.29 (a)

The signatures `yingyang(Integer[])` and `yingyang(Integer...)` are equivalent and, therefore, are not permitted in the same class.

7.30 (c)

The calls to the `compute()` method in the method declarations at (2) and at (3) are to the `compute()` method declaration at (1), as the argument is always an `int[]`.

The method call at (4) calls the method at (2). The signature of the call at (4) is

```
compute(int[], int[])
```

which matches the signature of the method at (2). No implicit array is created.

The method call in (5) calls the method at (1). An implicit array of int is created to store the argument values.

The method calls in (6) and (7) call the method in (3). Note the type of the variable arity parameter in (3): an `int[][]`. The signature of the calls at (6) and (7) is

```
compute(int[], int[][])
```

which matches the signature of the method at (3). No implicit array is created.

7.31 (e)

The program will print `2` when `System.out.println(ref2.f())` is executed. The object referenced by `ref2` is of class `C`, but the reference is of type `B`. Since `B` contains a method `f()`, the method call will be allowed at compile time. During execution it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method in `C` to be executed.

7.32 (c)

The program will print `1` when run. The `f()` methods in `A` and `B` are `private`, and are not accessible by the subclasses. Because of this, the subclasses cannot overload or override these methods, but simply define new methods with the same signature. The object being called is of class `C`. The reference used to access the object is of type `B`. Since `B` contains a method `g()`, the method call will be allowed at compile time. During execution it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method `g()` in `B` to be executed. This method calls

a method named `f`. It can be determined during compilation that this can refer to only the `f()` method in `B`, since the method is `private` and cannot be overridden. This method returns the value `1`, which is printed.

7.33 (b), (c), and (d)

The code as it stands will compile. The use of inheritance in this code defines a `Planet` *is-a* `Star` relationship. The code will fail if the name of the field `starName` is changed in the `Star` class, since the subclass `Planet` tries to access it using the name `starName`. An instance of `Planet` is not an instance of `HeavenlyBody`. Neither `Planet` nor `Star` implements `HeavenlyBody`.

7.34 (b)

The code will compile. The code will not fail to compile if the name of the field `starName` is changed in the `Star` class, since the `Planet` class does not try to access the field by name, but instead uses the `public` method `describe()` in the `Star` class for that purpose. An instance of `Planet` is not an instance of `HeavenlyBody`, since it neither implements `HeavenlyBody` nor extends a class that implements `HeavenlyBody`.

7.35 (e)

(a) to (f) are all true; therefore (e) is not.

8 Fundamental Classes

8.1 (b)

The method `hashCode()` in the `Object` class returns a hash code value of type `int`.

8.2 (e)

All arrays are genuine objects and inherit all the methods defined in the `Object` class, including the `clone()` method. Neither the `hashCode()` method nor the `equals()` method is declared as `final` in the `Object` class, and it cannot be guaranteed that implementations of these methods will differentiate among all objects.

8.3 (a)

The `clone()` method of the `Object` class will throw a `CloneNotSupportedException` if the class of the object does not implement the `Cloneable` interface.

8.4 (a), (c), and (d)

The class `java.lang.Void` is considered a wrapper class, although it does not wrap any value. There is no class named `java.lang.Int`, but there is a wrapper class named `java.lang.Integer`. A class named `java.lang.String` also exists, but it is not a wrapper class since all strings in Java are objects.

8.5 (c) and (d)

The classes `Character` and `Boolean` are non-numeric wrapper classes and do not extend the `Number` class. The classes `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are numeric wrapper classes that extend the abstract `Number` class.

8.6 (a), (b), and (d)

All instances of concrete wrapper classes are immutable. The `Number` class is an `abstract` class.

8.7 (b) and (c)

All instances of wrapper classes except `Void` and `Character` have a constructor that accepts a single `String` parameter. The class `Object` has only a no-argument constructor.

8.8 (e)

While all numeric wrapper classes have the methods `byteValue()`, `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()`, only the `Boolean` class has the `booleanValue()` method. Likewise, only the `Character` class has the `charValue()` method.

8.9 (b) and (d)

`String` is not a wrapper class. All wrapper classes except `Void` have a `compareTo()` method. Only the numeric wrapper classes have an `intValue()` method. The `Byte` class, like all other numeric wrapper classes, extends the `Number` class.

8.10 (a)

Using the `new` operator creates a new object. Boxing also creates a new object if one is not already interned from before.

8.11 (b) and (e)

The operators `-` and `&` cannot be used in conjunction with a `String` object. The operators `+` and `+=` perform concatenation on strings, and the dot operator accesses members of the `String` object.

8.12 (d)

The expression `str.substring(2,5)` will extract the substring `"kap"`. The method extracts the characters from index 2 to index 4, inclusive.

8.13 (d)

The program will print `str3str1` when run. The `concat()` method will create and return a new `String` object, which is the concatenation of the current `String` object and the `String` object given as an argument. The expression statement `str1.concat(str2)` creates a new `String` object, but its reference value is not stored after the expression is evaluated. Therefore this `String` object gets discarded.

8.14 (c)

The `trim()` method of the `String` class returns a string where both the leading and trailing whitespace of the original string have been removed.

8.15 (a) and (c)

The `String` class and all wrapper classes are declared as `final` and, therefore, cannot be extended. The `clone()` method is declared as `protected` in the `Object` class. `String` objects and wrapper class objects are immutable and, therefore, cannot be modi-

fied. The class `String` and `char` array types are unrelated, resulting in a compile-time error.

8.16 (d)

The constant expressions `"ab" + "12"` and `"ab" + 12` will, at compile time, be evaluated to the string-valued constant `"ab12"`. Both variables `s` and `t` are assigned a reference to the same interned `String` object containing `"ab12"`. The variable `u` is assigned a new `String` object, created by using the `new` operator.

8.17 (a), (c), (d), (f), and (j)

The `String` class has constructors with the parameter lists given in (a), (c), (d), (f), and (j).

8.18 (e)

The `String` class has no `reverse()` method.

8.19 (b)

The reference value in the reference `str1` never changes; it always refers to the string literal `"lower"`. The calls to `toUpperCase()` and `replace()` return a new `String` object whose reference value is ignored.

8.20 (d)

The call to the `put0()` method does not change the `String` object referred to by the `s1` reference in the `main()` method. The reference value returned by the call to the `concat()` method is ignored.

8.21 (a)

The code will fail to compile, since the expression `(s == sb)` is illegal. It compares references of two classes that are not related.

8.22 (e)

The program will compile without errors and will print `have a` when run. The contents of the string buffer are truncated to 6 characters by the method call `sb.setLength(6)`.

8.23 (a), (b), (d), and (e)

The `StringBuilder` class has only constructors with the parameter lists given in (a), (b), (d), and (e).

8.24 (a)

The `StringBuilder` class does not define a `trim()` method.

8.25 (b)

The references `sb1` and `sb2` are not aliases. The `StringBuilder` class does not override the `equals()` method; hence the answer is (b).

8.26 (a)

The `StringBuilder` class does not override the `hashCode()` method, but the `String` class does. The references `s1` and `s2` refer to a `String` object and a `StringBuilder` object, respectively. The hash values of these objects are computed by the `hashCode()`

method in the `String` and the `Object` class, respectively—giving different results. The references `s1` and `s3` refer to two different `String` objects that are equal; hence they have the same hash value.

8.27 *(b)*

The call to the `put0()` method changes the `StringBuilder` object referred to by the `s1` reference in the `main()` method. So does the call to the `append()` method.

9 Object Lifetime

9.1 *(e)*

An object is eligible for garbage collection only if all remaining references to the object are from other objects that are also eligible for garbage collection. Therefore, if an object `obj2` is eligible for garbage collection and object `obj1` contains a reference to it, then object `obj1` must also be eligible for garbage collection. Java does not have a keyword `delete`. An object will not necessarily be garbage collected immediately after it becomes unreachable, but the object will be eligible for garbage collection. Circular references do not prevent objects from being garbage collected; only reachable references do. An object is not eligible for garbage collection as long as the object can be accessed by any live thread. An object that is eligible for garbage collection can be made non-eligible if the `finalize()` method of the object creates a reachable reference to the object.

9.2 *(b)*

Before (1), the `String` object initially referenced by `arg1` is denoted by both `msg` and `arg1`. After (1), the `String` object is denoted by only `msg`. At (2), reference `msg` is assigned a new reference value. This reference value denotes a new `String` object created by concatenating the contents of several other `String` objects. After (2), there are no references to the `String` object initially referenced by `arg1`. The `String` object is now eligible for garbage collection.

9.3 *(d)*

It is difficult to say how many objects are eligible for garbage collection when control reaches (1), because some of the eligible objects may have already been finalized.

9.4 *(a)*

All the objects created in the loop are reachable via `p`, when control reaches (1).

9.5 *(b)*

The `Object` class defines a `protected finalize()` method. All classes inherit from `Object`; thus, all objects have a `finalize()` method. Classes can override the `finalize()` method and, as with all overriding, the new method must not reduce the accessibility. The `finalize()` method of an eligible object is called by the garbage collector to allow the object to do any cleaning up before the object is destroyed. When the garbage collector calls the `finalize()` method, it will ignore any exceptions thrown by the `finalize()` method. If the `finalize()` method is called

explicitly, normal exception handling occurs when an exception is thrown during the execution of the `finalize()` method; that is, exceptions are not simply ignored. Calling the `finalize()` method does not in itself destroy the object. Chaining of the `finalize()` method is not enforced by the compiler, and it is not mandatory to call the overridden `finalize()` method.

9.6 (d)

The `finalize()` method is like any other method: It can be called explicitly if it is accessible. However, such a method is intended to be called by the garbage collector to clean up before an object is destroyed. Overloading the `finalize()` method is allowed, but only the method with the original signature will be called by the garbage collector. The `finalize()` method in the `Object` class is `protected`. This means that any overriding method must be declared as either `protected` or `public`. The `finalize()` method in the `Object` class specifies a `Throwable` object in its `throws` clause. An overriding definition of this method can throw any type of `Throwable`. Overriding methods can limit the range of throwables to *unchecked* exceptions or specify no exceptions at all. Further overriding definitions of this method in subclasses will then *not* be able to throw *checked* exceptions.

9.7 (d) and (g)

(a), (b), (c), (j), (k), and (l) reduce the visibility of the inherited method. In (e), (f), (h), and (i), the call to the `finalize()` method of the superclass can throw a `Throwable`, which is not handled by the method. The `Throwable` superclass is not assignable to the `Exception` subclass.

9.8 (e)

It is not guaranteed if and when garbage collection will occur, nor in which order the objects will be finalized. However, it is guaranteed that the finalization of an object will be run only once. Hence, (e) cannot possibly be a result from running the program.

9.9 (c) and (e)

It is not guaranteed if and when garbage collection will occur, nor in which order the objects will be finalized. Thus, the program may not print anything. If garbage collection does take place, the `MyString` object created in the program may get finalized before the program terminates. In that case, the `finalize()` method will print `A`, as the string in the field `str` is not changed by the `concat()` method. Keep in mind that a `String` object is immutable.

9.10 (c), (e), and (f)

The static initializer blocks (a) and (b) are not legal, since the fields `alive` and `STEP` are non-`static` and `final`, respectively. (d) is not a syntactically legal static initializer block. The static block in (e) will have no effect, as it executes the empty statement. The static block in (f) will change the value of the static field `count` from 5 to 1.

9.11 (c)

The program will compile, and print `50, 70, 0, 20, 0` at runtime. All fields are given default values unless they are explicitly initialized. Field `i` is assigned the

value 50 in the static initializer block that is executed when the class is initialized. This assignment will override the explicit initialization of field `i` in its declaration statement. When the `main()` method is executed, the static field `i` is 50 and the static field `n` is 0. When an instance of the class is created using the `new` operator, the value of static field `n` (i.e., 0) is passed to the constructor. Before the body of the constructor is executed, the instance initializer block is executed, which assigns the values 70 and 20 to the fields `j` and `n`, respectively. When the body of the constructor is executed, the fields `i`, `j`, `k`, and `n` and the parameter `m` have the values 50, 70, 0, 20, and 0, respectively.

9.12 *(f)*

This class has a blank final `boolean` instance variable `active`. This variable must be initialized when an instance is constructed, or else the code will not compile. This also applies to blank `final static` variables. The keyword `static` is used to signify that a block is a static initializer block. No keyword is used to signify that a block is an instance initializer block. (a) and (b) are not instance initializers blocks, and (c), (d), and (e) fail to initialize the blank final variable `active`.

9.13 *(c)*

The program will compile, and print `2`, `3`, and `1` at runtime. When the object is created and initialized, the instance initializer block is executed first, printing `2`. Then the instance initializer expression is executed, printing `3`. Finally, the constructor body is executed, printing `1`. The forward reference in the instance initializer block is legal, as the use of the field `m` is on the left-hand side of the assignment.

9.14 *(c) and (e)*

Line A will cause an illegal redefinition of the field `width`. Line B uses an illegal forward reference to the fields `width` and `height`. The assignment in Line C is legal. Line D is an assignment statement, so it is illegal in this context. Line E declares a local variable inside an initializer block, with the same name as the instance variable `width`, which is allowed. The simple name in this block will refer to the local variable. To access the instance variable `width`, the `this` reference must be used in this block.

10 The ArrayList<E> Class and Lambda Expressions

10.1 *(h)*

The method `remove()` can be used to delete an element at a specific index in an `ArrayList`.

The method `clear()` can be used to delete all elements in an `ArrayList`.

The method `add(int, E)` can be used to insert an element at a specific index in an `ArrayList`.

The method `add()` can be used to append an element at the end of an `ArrayList`.

The method `set()` can be used to replace the element at a specific index with another element in an `ArrayList`.

The method `contains()` can be used to determine whether an element is in an `ArrayList`.

There is no method to determine the current capacity of an `ArrayList`.

10.2 (e)

The `for(;;)` loop correctly increments the loop variable so that all the elements in the list are traversed. Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

10.3 (b) and (c)

In the method `doIt1()`, one of the common elements (`"Ada"`) between the two lists is reversed. The value `null` is added to only one of the lists but not the other.

In the method `doIt2()`, the two lists have common elements. Swapping the elements in one list does not change their positions in the other list.

10.4 (c)

The element at index 2 has the value `null`. Calling the `equals()` method on this element throws a `NullPointerException`.

10.5 (f)

Deleting elements when traversing a list requires care, as the size changes and any elements to the right of the deleted element are shifted left. Incrementing the loop variable after deleting an element will miss the next element, as is the case with the last occurrence of `"Bob"`. Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

10.6 (f)

The `while` loop will execute as long as the `remove()` methods returns `true`—that is, as long as there is an element with the value `"Bob"` in the list. The `while` loop body is the empty statement. The `remove()` method does not throw an exception if an element value is `null`, or if it is passed a `null` value.

10.7 (f)

A functional interface can be implemented by lambda expressions and classes.

A functional interface declaration can have only one abstract method declaration.

In the body of a lambda expression, all members in the enclosing class can be accessed.

In the body of a lambda expression, only effectively final local variables in the enclosing scope can be accessed.

A lambda expression in a program can implement more than one functional interface. For example, the lambda expression `(i -> i%2 == 0)` can be the target type of both the functional interfaces `IntPredicate` and `Predicate<Integer>`.

10.8 (a) and (c)

(1) redeclares the local variable `p` from the enclosing scope, which is not legal.

In (2), the `equals()` method of the `String` class is called, because it is invoked on the textual representation of the parameter. In the other statements, the `equals()` method of the object referred to by the parameter is called.

The lambda body in (3) is a statement block with an expression whose value must be returned by the `return` statement.

(4) and (5) access static members in the class, which is legal.

In (6), the parameter name `lock2` shadows the static variable by the same name, but is a local variable in the lambda expression. The static variable is referred to using the class name.

10.9 (e), (f), (g), and (i)

Assignments in (5), (6), (7), and (9) will not compile. We must check whether the function type of the target type and the type of the lambda expression are compatible. The function type of the target type `p1` in the assignment statements from (1) to (5) is `String -> void`, or a `void` return. The function type of the target type `p2` in the assignment statements from (6) to (10) is `String -> String`, or a non-`void` return. In the following code, the functional type of the target type is shown in a comment with the prefix `LHS` (left-hand side), and the type of the lambda expression for each assignment from (1) to (10) is shown in a comment with the prefix `RHS` (right-hand side).

```
    Funky1 p1;                                  //     LHS: String -> void
    p1 = s -> System.out.println(s);            // (1) RHS: String -> void
    p1 = s -> s.length();                       // (2) RHS: String -> int
    p1 = s -> s.toUpperCase();                  // (3) RHS: String -> String
    p1 = s -> { s.toUpperCase(); };             // (4) RHS: String -> void
//  p1 = s -> { return s.toUpperCase(); };      // (5) RHS: String -> String

    Funky2 p2;                                  //     LHS: String -> String
//  p2 = s -> System.out.println(s);            // (6) RHS: String -> void
//  p2 = s -> s.length();                       // (7) RHS: String -> int
    p2 = s -> s.toUpperCase();                  // (8) RHS: String -> String
//  p2 = s -> { s.toUpperCase(); };             // (9) RHS: String -> void
    p2 = s -> { return s.toUpperCase(); };      // (10)RHS: String -> String
```

The non-`void` return of a lambda expression with an *expression statement* as the body can be interpreted as a `void` return, if the function type of the target type returns `void`. This is the case in (2) and (3). The return value is ignored. The type `String -> String` of the lambda expression in (5) is not compatible with the function type `String -> void` of the target type `p1`.

The type of the lambda expression in (6), (7), and (9) is not compatible with the function type `String -> String` of the target type `p2`.

10.10 (d)

The lambda expression filters all integer values that are both negative and even numbers. These values are replaced with their absolute values in the integer array. The functional interface `java.util.function.IntPredicate` has the abstract method: `boolean test(int i)`.

10.11 (d)

The three interfaces are functional interfaces. `InterfaceB` explicitly provides an abstract method declaration of the `public` method `equals()` from the `Object` class, but such declarations are excluded from the definition of a functional interface. Thus `InterfaceB` effectively has only one abstract method. A functional interface can be implemented by a concrete class, such as `Beta`. The function type of the target type in the assignments (1) to (3) is `void -> void`. The type of the lambda expression in (1) to (3) is also `void -> void`. The assignments (1) to (3) are legal.

The assignment in (4) is legal. Subtype references are assigned to supertype references. References `o`, `a`, and `c` refer to the lambda expression in (3).

The assignment in (5) is legal. The reference `b` has the type `InterfaceB`, and class `Beta` implements this interface.

(6), (7), and (8) invoke the method `doIt()`. (6) evaluates the lambda expression in (3), printing `Jingle|`. (7) invokes the `doIt()` method on an object of class `Beta`, printing `Jazz|`. (8) also evaluates the lambda expression in (3), printing `Jingle|`.

In (9), the reference `o` is cast down to `InterfaceA`. The reference `o` actually refers to the lambda expression in (3), which has target type `InterfaceC`. This interface is a subtype of `InterfaceA`. The subtype is cast to a supertype, which is allowed, so no `ClassCastException` is thrown at runtime. Invoking the `doIt()` method again results in evaluation of the lambda expression in (3), printing `Jingle|`.

Apart from the declarations of the lambda expressions, the rest of the code is plain-vanilla Java. Note also that the following assignment that defines a lambda expression would not be valid, since the `Object` class is not a functional interface and therefore cannot provide a target type for the lambda expression:

```
Object obj = () -> System.out.println("Jingle");      // Compile-time error!
```

11 Date and Time

11.1 (e)

The `LocalDateTime` class does not provide the `isLeapYear()` method.

The `LocalTime` class does not provide the `isEqual()` method.

The `Period` class does not provide the `withWeeks()` method, but does provide the `ofWeeks()` static method.

Both the `Period` and `LocalTime` classes do not provide the `plusWeeks()` method.

11.2 (e)

The `date` reference never gets updated, as the return value is ignored. If it had been updated, the correct answer would have been (c). The `LocalDate.getMonth()` method returns a `Month` enum constant—in this case, `Month.MARCH`. The `LocalDate.getMonthValue()` method returns the month as a value between 1 and 12—in this case, `3`.

11.3 (b), (c), (e), and (g)

(a): The month numbers start with 1. August has month value 8.

(d): Invalid month (0) and day (0) arguments in the call to the `of()` method result in a `DateTimeException` being thrown at runtime.

(f): The `LocalDate` class does not provide a `public` constructor.

11.4 (c), (d), and (f)

(a): Invalid argument for the minutes (0–59).

(b): The `LocalTime` class does not provide a `public` constructor.

(c): The time assigned is 09:00.

(d): The time assigned is 00:00.

(e): There is no `withHours()` method, but there is a `withHour()` method in the `LocalTime` class.

(f): The time assigned is 11:45.

11.5 (c)

Both the hour and minutes are normalized by the `plus` methods, and the time of day wraps around midnight. The calculation of `time.plusHours(10).plusMinutes(120)` proceeds as follows:

```
12:00 + 10 hours ==> 22:00 + 120 min (i.e., 2 hrs.) ==> 00:00
```

11.6 (d)

The calculation of `p1.plus(p2).plus(p1)` proceeds as follows:

```
P1Y1M1D + P2Y12M30D ==> P3Y13M31D + P1Y1M1D ==> P4Y14M32D
```

11.7 (c)

The calculation of `date.withYear(5).plusMonths(14)` proceeds as follows:

```
2015-01-01 with year 5 ==> 0005-01-01 + 14 months (i.e., 1 year 2 months) ==>
0006-03-01
```

11.8 (a), (d), (e), (g), and (i)

The `between()` and `until()` methods return a `Period`, which can be negative. The `isAfter()`, `isBefore()`, `between()`, and `until()` methods are strict in the sense that the end date is excluded. The `compareTo()` method returns `0` if the two dates are equal, a negative value if `date1` is less than `date2`, and a positive value if `date1` is greater than `date2`.

11.9 (e)

(a): The `DateTimeFormatter` class provides factory methods to obtain both predefined and customized formatters.

(b): The styles defined by the `java.time.format.FormatStyle` enum type are locale sensitive.

(c): The `ofLocalizedDate()` method of the `DateTimeFormatter` class returns a formatter that is based on a format style (a constant of the `FormatStyle` enum type) passed as an argument to the method.

(d): The pattern `"yy-mm-dd"` cannot be used to create a formatter that can format a `LocalDate` object. The letter `m` stands for minutes of the hour, which is not a part of a date.

11.10 (a), (b), (c), and (f)

(a), (b), (c): The input string matches the pattern. The input string specifies the mandatory parts of both a date and a time, needed by the respective method to construct either a `LocalTime`, a `LocalDate`, or a `LocalDateTime`.

To use the pattern for formatting, the temporal object must provide the parts corresponding to the pattern letters in the pattern. The `LocalTime` object in (d) does not have the date part required by the pattern. The `LocalDate` object in (e) does not have the time part required by the pattern. Both (d) and (e) will throw an `UnsupportedTemporalTypeException`. Only the `LocalDateTime` object in (f) has both the date and time parts required by the pattern.

11.11 (b), (e), and (f)

The input string matches the pattern. It specifies the date-based values that can be used to construct a `LocalDate` object in (b), based on the date-related pattern letters in the pattern. No time-based values can be interpreted from the input string, as this pattern has only date-related pattern letters. (a) and (c), which require a time part, will throw a `DateTimeParseException`.

To use the pattern for formatting, the temporal object must provide values for the parts corresponding to the pattern letters in the pattern. The `LocalTime` object in (d) does not have the date part required by the pattern. (d) will throw an `UnsupportedTemporalTypeException`. The `LocalDate` object in (e) has the date part required by the pattern, as does the `LocalDateTime` object in (f). In (f), only the date part of the `LocalDateTime` object is formatted.

11.12 (e)

(a), (b), (c), (d), and (f) result in a `DateTimeParseException` when parsing.

(a): The pattern letter `h` represents hour in the day, but requires AM/PM information to resolve the hour in a 24-hour clock (i.e., pattern letter `a`), which is missing.

(b): The pattern letter `M` is interpreted correctly as month of the year (value 5). Matching the pattern letter `h` is the problem, as explained for (a).

(c), (d): The pattern letter `a` cannot be resolved from the input string, as an AM/PM marker is missing in the input string.

(e): The parse succeeds, with the `LocalTime` object having the value 09:05. Formatting this object with the formatter results in the output string: `5 minutes past 9`.

(f): The letter pattern `mm` cannot be resolved, as the minutes value has only one digit (i.e., 5) in the input string.

(g): The parse succeeds, with the resulting `LocalTime` object having the value 09:00. The month value 5 is ignored. Formatting this object with the formatter results in an `UnsupportedTemporalTypeException`, because now the pattern letter `M` requires a month value, which is not part of a `LocalTime` object.

11.13 (d)

(a): The formatter will format a `LocalTime` object, or the time part of a `LocalDateTime` object, but not a `LocalDate` object, as it knows nothing about formatting the date part.

(b): The formatter will format a `LocalDate` object, or the date part of a `LocalDateTime` object, but not a `LocalTime` object, as it knows nothing about formatting the time part.
(c): The formatter will format a `LocalDateTime` object, but not a `LocalDate` object or a `LocalTime` object, as it will format only temporal objects with both date and time parts.

The program throws a `java.time.temporal.UnsupportedTemporalTypeException` in all cases.