2 Language Fundamentals

Exam Objectives

- Identify correctly constructed `package` declarations, `import` statements, class declarations (of all forms, including inner classes), interface declarations and implementations (for `java.lang.Runnable` or other interface described in the test), method declarations (including the `main` method that is used to start execution of a class), variable declarations and identifiers.
    - *For defining and using packages, see Section 4.5.*
    - *For class declarations, see Section 4.2.*
    - *For inner classes, see Chapter 7.*
    - *For interface declarations and implementations, see Section 6.4.*
    - *For method declarations, see Section 4.3.*
- State the correspondence between index values in the argument array passed to a `main` method and command line arguments.
    - *See Section 3.23.*
- Identify all Java programming language keywords and correctly constructed identifiers.
- State the effect of using a variable or array element of any kind, when no explicit assignment has been made to it.
    - *For array elements, see Section 4.1.*
- State the range of all primitive data types, and declare literal values for `String` and all primitive types using all permitted formats, bases and representations.
- Write code to implement listener classes and methods, and in listener methods extract information from the event to determine the affected component, mouse position, nature and time of the event. State the event class name for any specified event listener interface in the `java.awt.event` package.
    - *See Chapter 14.*

Supplementary Objectives

- State the wrapper classes for primitive data types.

2.1 Language Building Blocks

Like any other programming language, the Java programming language is defined by *grammar rules* that specify how *syntactically* legal constructs can be formed using the language elements, and by a *semantic definition* that specifies the *meaning* of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called *lexical tokens* (or just *tokens* for short) and are the building blocks for more complex constructs. Identifiers, operators and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods and classes.

Identifiers

A name in a program is called an *identifier*. Identifiers can be used to denote classes, methods and variables.

In Java an *identifier* is composed of a sequence of characters, where each character can be either a *letter*, a *digit*, a *connecting punctuation* (such as *underscore* _) or any *currency symbol* (such as `$`, `¢`, `¥` or `£`), and cannot start with a digit. Since Java programs are written in the Unicode character set (p. 24), the definitions of letter and digit are interpreted according to this character set.

Note that Java is case-sensitive, e.g. `price` and `Price` are two different identifiers.

Examples of legal identifiers:

```
number, Number, sum_$, bingo, $$_100, mål, grüß
```

Examples of illegal identifiers:

```
48chevy, all/clear, get-lost-fred
```

Keywords

Keywords are reserved identifiers that are predefined in the language, and cannot be used to denote other entities. Incorrect usage results in compilation errors.

Keywords currently defined in the language are listed in Table 2.1. In addition, three identifiers are reserved as predefined *literals* in the language: `null`, `true`, `false` (Table 2.3). Keywords currently reserved, but not in use, are listed in Table 2.2. All these reserved words cannot be used as identifiers. The index contains references to relevant sections where currently defined keywords are explained.

Table 2.1 *Keywords in Java*

`abstract`	`do`	`import`	`public`	`transient`
`boolean`	`double`	`instanceof`	`return`	`try`
`break`	`else`	`int`	`short`	`void`
`byte`	`extends`	`interface`	`static`	`volatile`
`case`	`final`	`long`	`super`	`while`
`catch`	`finally`	`native`	`switch`	
`char`	`float`	`new`	`synchronized`	
`class`	`for`	`package`	`this`	
`continue`	`if`	`private`	`throw`	
`default`	`implements`	`protected`	`throws`	

Table 2.2 *Reserved Keywords not currently in use*

`const`	`goto`

Table 2.3 *Reserved Literals in Java*

`null`	`true`	`false`

Literals

A *literal* denotes a constant value. This value can be numerical (integer or floating-point), character, boolean or a string. In addition there is the null literal (`null`) which represents the null reference.

Table 2.4 *Examples of Literals*

Integer	`2000`	`0`	`-7`		
Floating-point	`3.14`	`-3.14`	`.5`	`0.5`	
Character	`'a'`	`'A'`	`'0'`	`'*'`	`')'`
Boolean	`true`	`false`			
String	`"abba"`	`"3.14"`	`"for"`	`"a piece of the action"`	

Integer Literals

Integer datatypes are comprised of the following primitive types: `int`, `long`, `byte` and `short`.

The default type of an integer literal is `int`, but it can be specified as `long` by appending the suffix `L` (or `l`) to the integer value; for example `2000L`, `0l`. There is no way to specify a `short` or a `byte` literal.

Octal Numbers and Hexadecimal Numbers

In addition to the decimal number system, integer literals can also be specified in octal (*base* `8`) and hexadecimal (*base* `16`) number systems. Table 2.5 lists the integers from `0` to `16`, showing their equivalents in the octal and hexadecimal number systems.

Table 2.5 *Number Systems*

Decimal numbers	Octal numbers	Hexadecimal numbers
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	10	8
9	11	9
10	12	a
11	13	b
12	14	c
13	15	d
14	16	e
15	17	f
16	20	10

In Java, octal and hexadecimal numbers are specified with `0` and `0x` prefix respectively. Some examples of octal and hexadecimal literals are shown in Table 2.6.

Converting Octal and Hexadecimal Numbers to Decimals

Octal and hexadecimal numbers can be easily converted to their decimal equivalents:

$0132 = 1*8^2 + 3*8^1 + 2*8^0 = 64 + 24 + 2 = 90$ (1) Octal -> Decimal

$0x5a = 5*16^1 + a*16^0 = 80 + 10 = 90$ (2) Hex -> Decimal

Table 2.6 *Examples of Octal and Hexadecimal Literals in Java*

Decimal	Octal	Hexadecimal
`8`	`010`	`0x8`
`10`	`012`	`0xa`
`16`	`020`	`0x10`
`27`	`033`	`0x1b`
`90`	`0132`	`0x5a`
`2147483647`	`017777777777`	`0x7fffffff`
`-2147483648`	`-017777777777`	`-0x7fffffff`

At (1) an octal number, expressed in base 8, is converted to its equivalent decimal value. Each digit in the octal number contributes to the final decimal value by virtue of its position, starting with position 0 (units) for the rightmost digit in the number. Since hexadecimal numbers have the base 16, this value is used as the base for converting from hexadecimal to decimal in (2).

Floating-point Literals

Floating-point data types come in two flavors: `float` or `double`.

The default type of a floating-point literal is `double`, but this can be explicitly designated by appending the suffix `D` (or `d`) to the value. A floating-point literal can also be specified to be a `float` by appending the suffix `F` (or `f`).

Floating-point literals can also be specified in scientific notation, for example `5E-1` is equivalent to $5*10^{-1}$, i.e. `0.5`, where `E` (or `e`) stands for *Exponent*.

Boolean Literals

Boolean truth-values can be denoted using the reserved literals `true` or `false`.

Character Literals

A character literal is quoted in single-quotes (`'`).

All characters are represented by 16-bit Unicode. The Unicode character set subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. In Table 2.7, note that digits (`1` to `9`), upper-case letters (`A` to `Z`) and lower-case letters (`a` to `z`) have contiguous Unicode values.

Table 2.7 *Examples of Unicode Values*

Character Literal	Unicode value (using hexadecimal digits)	Character
`' '`	`\u0020`	Space
`'0'`	`\u0030`	0
`'1'`	`\u0031`	1
`'9'`	`\u0039`	9
`'A'`	`\u0041`	A
`'B'`	`\u0042`	B
`'Z'`	`\u005a`	Z
`'a'`	`\u0061`	a
`'b'`	`\u0062`	b
`'z'`	`\u007a`	z
`‘Ñ’`	`\u0084`	Ñ
`‘å’`	`\u008c`	å
`‘ß’`	`\u00a7`	ß

Unicode Literals

Alternatively, a character literal can be defined by quoting the Unicode value, as shown in Table 2.8.

Table 2.8 *Expressing Character Literals as Unicode Values*

Character Literal	Unicode Literal	Character
`' '`	`'\u0020'`	Space
`'0'`	`'\u0030'`	0
`'A'`	`'\u0041'`	A

Escape Sequences

Certain *escape sequences* define special character values as shown in Table 2.9. These escape sequences can be single-quoted to define character literals. For example, the character literals `'\t'` and `'\u0009'` are equivalent.

Table 2.9 *Escape Sequences*

Escape Sequence	Unicode Value	Character
`\b`	`\u0008`	Backspace
`\t`	`\u0009`	Horizontal tabulation
`\n`	`\u000a`	Linefeed
`\f`	`\u000c`	Form feed
`\r`	`\u000d`	Carriage return
`\'`	`\u0027`	Apostrophe-quote
`\"`	`\u0022`	Quotation mark
`\\`	`\u005c`	Backslash

String Literals

A string literal is a sequence of characters, which must be quoted in quotation marks and which must occur on a single line.

Escape sequences as well as Unicode values can appear in string literals:

```
"Here comes a tab.\t And here comes another one\u0009!"   // (1)
"What's on the menu?"                                     // (2)
"\"String literals are double-quoted.\""                  // (3)
```

In (1), the tab character is specified using the escape sequence and the Unicode value respectively. In (2), the single apostrophe need not be escaped in strings, but it would be if specified as a character literal(`'\''`). In (3), the double apostrophes in the string must be escaped. Printing these strings would give the following result:

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
```

White Spaces

A white space is a sequence of spaces, tabs, form feeds and line terminator characters. Line terminators can be newline, carriage return or carriage return-newline sequence in a Java source file.

A Java program is a free-format sequence of characters which is *tokenized* by the compiler, i.e. broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes white space has to be inserted explicitly. For example, the identifier `classRoom` will be interpreted as a single token, unless white space is inserted to distinguish the keyword `class` from the identifier `Room`.

White space aids not only in separating tokens, but also in formatting the program so that it is easy for humans to read. The compiler ignores the white spaces once the tokens are identified.

Comments

A program can be documented by inserting comments at relevant places. These comments are for documentation purposes and are ignored by the compiler.

Java provides three types of comments to document a program:

- A single-line comment
- A multiple-line comment
- A documentation (or Javadoc) comment

Regardless of the type of comment, they cannot be nested. The comment-start sequences (`//`, `/*`, `/**`) are not treated differently from other characters when occurring within comments.

Single-line Comment

All characters after the comment-start sequence `//` through to the end of the line constitute a single-line comment.

```
// This comment ends at the end of this line.
```

Multiple-line Comment

A multiple-line comment, as the name suggests, can span several lines. Such a comment starts with `/*` and ends with `*/`.

```
/*  A comment
    on several
    lines.
*/
```

Documentation Comment

A documentation comment is a special-purpose comment which when placed at appropriate places in the program can be extracted and used by the `javadoc` utility to generate HTML documentation for the program. Documentation comments are usually placed in front of class, interface, method and variable definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with `/**` and ends with `*/`:

```
/**
 *  This class implements a gizmo
 *  @author K.A.M.
 *  @version 1.0
 */
```

For a detailed discussion of the `javadoc` utility, see Chapter 19.

Review questions

2.1 Which of the following is not a legal identifier?

Select all valid answers.

(a) `a2z`
(b) `ödipus`
(c) `52pickup`
(d) `_class`
(e) `ca$h`

2.2 Which one of these statements is correct?

Select the one right answer.

(a) `new` and `delete` are keywords in the Java language.
(b) `try`, `catch` and `thrown` are keywords in the Java language.
(c) `static`, `unsigned` and `long` are keywords in the Java language.
(d) `exit`, `class` and `while` are keywords in the Java language.
(e) `return`, `goto` and `default` are keywords in the Java language.
(f) `for`, `while` and `next` are keywords in the Java language.

2.3 Is this a complete and legal comment?

```
/* // */
```

Select the one right answer.

(a) No, the block comment (`/* ... */`) is not ended since the single-line comment (`// ...`) comments out the closing part.
(b) It is a completely valid comment. The `//` part is ignored by the compiler.
(c) This combination of comments is illegal and the compiler will reject it.

2.2 Primitive Datatypes

Figure 2.1 gives an overview of the primitive datatypes in Java.

Primitive datatypes in Java can be divided into three main categories:

- Integral types consisting of integers and characters:
  Integer datatypes are `byte`, `short`, `int` and `long`. They represent signed integers.

  The character datatype is represented by the `char` type. It represents the symbols in the Unicode character set, like letters, digits and special characters.
- Floating-point types:
  This category includes `float` and `double` datatypes. They represent fractional signed numbers.

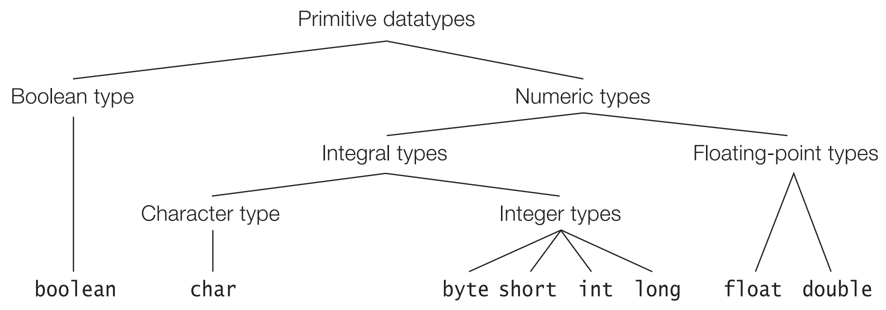

Figure 2.1 *Primitive Datatypes in Java*

- Boolean type:
  The datatype `boolean` represents truth-values `true` and `false`.

Primitive data values are *atomic* and are not objects. Each primitive datatype defines the range of values in the datatype, and operations on these values are defined by special operators in the language.

Each primitive datatype has a corresponding *wrapper* class that can be used to represent a primitive value as an object. Wrapper classes are discussed in Section 10.3.

2.3 Variable Declarations

Declaring, Initializing and Using Variables

Variables in Java come in three flavors:

- *Instance variables* that are members of a class and are instantiated for each object of the class. In other words, all instances, i.e. objects, of the class will have their own instances of these variables, which are local to the object. The values of these variables at any given time constitute the *state* of the object.
- *Static variables* that are also members of a class, but these are not instantiated for any object of the class and therefore belong only to the class (Section 4.10, p. 121).
- *Local variables* (also called *method automatic variables*), which are declared in methods and in blocks, are instantiated for each invocation of the method or block. In Java, local variables must be declared before they can be used (Section 4.8, p. 113).

A *variable* stores values of datatypes. A variable has a name, a type, a particular size and a value associated with it.

A variable declaration, in its simplest form, can be used to specify the name and the type of variables. This implicitly determines their size and the values that can be stored in them.

```
char a, b, c;             // a, b and c are character variables.
double area;              // area is a floating-point variable.
boolean flag;             // flag is a boolean variable.
```

A declaration can also include initialization code to specify an initial value for the variable:

```
int i = 10,               // i is an int variable with initial value 10.
    j = 101;              // j is an int variable with initial value 101.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

In Java, variables can only store values of primitive datatypes and references to objects.

Initializers for initializing member variables in objects, classes and interfaces are discussed in Section 8.2.

Object Reference Variables

An *object reference* provides a handle for an object. References can be stored in variables.

In Java, reference variables must be declared and initialized before they can be used. A reference variable has a name and a type or class associated with it. A reference variable declaration, in its simplest form, can be used to specify the name and the type. This determines what objects a reference variable can denote.

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class Hamburger,
          smallOne; // and so can variable smallOne.
```

It is important to note that the declarations above do not create objects of class `Pizza` or `Hamburger`. They only create variables which can store references to objects of these classes.

A declaration can also include an initializer to create an object that can be assigned to the reference variable:

```
Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration with initializer.
```

The reference variable `yummyPizza` can reference objects of class `Pizza`. The keyword `new`, together with the *constructor call* `Pizza("Hot&Spicy")`, creates an object of class `Pizza`. The reference to this object is assigned to the variable `yummyPizza`. The newly created object of class `Pizza` can now be manipulated through the reference stored in this variable.

2.4 Integers

Table 2.10 *Range of Integer Values*

Datatype	Width (bits)	Minimum value MIN_VALUE	Maximum value MAX_VALUE
byte	8	-2^{7} (-128)	$2^{7}-1$ (+127)
short	16	-2^{15} (-32768)	$2^{15}-1$ (+32767)
int	32	-2^{31} (-2147483648)	$2^{31}-1$ (+2147483647)
long	64	-2^{63} (-9223372036854775808L)	$2^{63}-1$ (+9223372036854775807L)

Integer values are represented as signed with 2's complement (Section 3.12, p. 64).

```
int i      = -215;                 // int literal
int max    = 0x7fffffff;           // 2147483647 as hex int literal
int min    = 0x80000000;           // -2147483648 as hex int literal
long isbn  = 05402202647L;         // octal long literal
long phone = 55584152L;            // long literal
```

2.5 Characters

Table 2.11 *Range of Character Values*

Datatype	Width (bits)	Minimum Unicode value	Maximum Unicode value
char	16	0x0	0xffff

The char datatype encompasses all the 65536 (2^{16}) characters in the Unicode character set as 16-bit values. The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set. See Section 18.4 on page 570 for a discussion on *character encodings*.

2.6 Floating-point Numbers

Table 2.12 *Range of Floating-point Values*

Datatype	Width (bits)	Minimum value MIN_VALUE	Maximum value MAX_VALUE
float	32	1.40129846432481707e-45	3.40282346638528860e+38
double	64	4.94065645841246544e-324	1.79769313486231570e+308

Floating-point numbers conform to the IEEE 754-1985 standard. Table 2.12 shows the range of values for positive floating-point numbers, but these apply equally to negative floating-point numbers with the `'-'` sign as prefix. Zero can be either `0.0` or `-0.0`.

Since the size for representation is finite, certain floating-point numbers can only be represented as approximations.

```
float pi = 3.14159F;
double p = 314.159e-2;
double fraction = 1.0/3.0;
```

2.7 Booleans

Table 2.13 *Boolean Values*

Datatype	Width	True value/literal	False value/literal
`boolean`	not applicable	`true`	`false`

The `boolean` datatype is used to represent logical values that can be either the literal `true` or the literal `false`.

Boolean values are returned by all *relational* (Section 3.8), *conditional* (Section 3.11) and boolean logical operators (Section 3.10), and are primarily used to govern the flow of control during program execution.

Note that boolean values cannot be converted to other primitive data values, and vice versa.

2.8 Wrapper Classes

The wrapper classes for primitive datatypes are found in the `java.lang` package, and are summarized in Table 2.14. For each primitive datatype there is a corresponding wrapper class to represent the values of the primitive datatype as an object. These wrapper classes also define useful methods for manipulating both primitive data values and objects. Wrapper classes are discussed in detail in Section 10.3.

The wrapper classes for integers (`Byte`, `Short`, `Integer`, and `Long`) are subclasses of the `java.lang.Number` class, as are the wrapper classes for floating-point numbers (`Float`, `Double`).

Examples of Primitive Values as Objects:

```
Integer intObj     = new Integer(2010);
Long longObj       = new Long(2030L);

Float floatObj     = new Float(3.14F);
Double doubleObj   = new Double(3.14D);

Character charObj = new Character('\t');
Boolean boolObj    = new Boolean(true);
```

Table 2.14 *Summary of Primitive Datatypes*

Datatype	Width (bits)	Minimum value, Maximum value	Wrapper Class
`boolean`	not applicable	`true, false (no ordering)`	`Boolean`
`byte`	8	-2^{7}, $2^{7}-1$	`Byte`
`short`	16	-2^{15}, $2^{15}-1$	`Short`
`char`	16	`0x0, 0xffff`	`Character`
`int`	32	-2^{31}, $2^{31}-1$	`Integer`
`long`	64	-2^{63}, $2^{63}-1$	`Long`
`float`	32	`±1.40129846432481707e-45, ±3.40282346638528860e+38`	`Float`
`double`	64	`±4.94065645841246544e-324, ±1.79769313486231570e+308`	`Double`

Review questions

2.4 Which of the following does not denote a primitive data value in Java?

Select all valid answers.

(a) `"t"`
(b) `'k'`
(c) `50.5F`
(d) `"hello"`
(e) `false`

2.5 Which of the following lines are valid declarations?

Select all valid answers.

(a) `char a = '\u0061';`
(b) `char \u0061 = 'a';`
(c) `ch\u0061r a = 'a';`

2.6 Which integral type in Java has the exact range from `-2147483648` (-2^{31}) to `2147483647` ($2^{31}-1$), inclusive?

Select the one right answer.

(a) Type `byte`
(b) Type `short`
(c) Type `int`
(d) Type `long`
(e) Type `char`

2.9 Initial Values for Variables

Default Values for Member Variables

Default values for primitive datatypes are listed in Table 2.15.

Table 2.15 *Default Values*

Datatype	Default value
boolean	false
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0`
Floating-point (`float`, `double`)	`+0.0F or +0.0D`
Object reference	`null`

Static variables in a class are initialized to default values when the class is loaded, if they are not explicitly initialized.

Instance variables are also initialized to default values when the class is instantiated, if they are not explicitly initialized.

Note that a reference variable is initialized with the value `null`.

Example 2.1 *Default Values for Member Variables*

```
class Light {
    // Static variable
    static int counter;      // Default value 0 when class is loaded.

    // Instance variables
    int noOfWatts = 100;     // Explicitly set to 100.
    boolean indicator;       // Implicitly set to default value false.
    String location;         // Implicitly set to default value null.
    public static void main(String args[]) {
        Light bulb = new Light();
        System.out.println("Static member counter: " + Light.counter);
        System.out.println("Instance member noOfWatts: " + bulb.noOfWatts);
        System.out.println("Instance member indicator: " + bulb.indicator);
        System.out.println("Instance member location: " + bulb.location);
    }
}
```

Output from the program:

```
Static member counter: 0
Instance member noOfWatts: 100
Instance member indicator: false
Instance member location: null
```

Example 2.1 illustrates default initialization of member variables. Note that static variables are initialized when the class is loaded the first time, and instance variables are initialized accordingly in *every* object created from the class `Light`.

Initializing Local Variables of Primitive Datatypes

Local variables are *not* initialized when they are instantiated at method invocation. The compiler `javac` reports use of uninitialized local variables.

Example 2.2 *Flagging Uninitialized Local Variables of Primitive Datatypes*

```
public class TooSmartClass {
    public static void main(String args[]) {
        int weight = 10, thePrice;                       // local variables

        if (weight < 10) thePrice = 100;
        if (weight > 50) thePrice = 5000;
        if (weight >= 10) thePrice = weight*10;          // Always executed.

        System.out.println("The price is: " + thePrice);   // (1)
    }
}
```

In Example 2.2, the compiler complains that the local variable `thePrice` in the `println` statement at (1) may not be initialized. However, from the program it can be seen that the local variable `thePrice` gets the value `100` in the last `if`-statement before it is used in the `println` statement. The compiler does not perform a rigorous analysis of the program in this regard. The program will compile correctly if the variable was initialized in the declaration, or if an unconditional assignment is made to the variable in the method.

Initializing Local Reference Variables

Note that the same initialization rules that apply to local variables of primitive datatypes also apply to local reference variables.

Example 2.3 *Flagging Uninitialized Local Reference Variables*

```
public class VerySmartClass {
    public static void main(String args[]) {
        String oneLongString;       // local reference variable

        System.out.println("The string length is: " + oneLongString.length());
    }
}
```

In Example 2.3, the compiler complains that the local variable `oneLongString` in the `println` statement may not be initialized. Objects should be created and their state initialized appropriately (for example, in a constructor) before use. If the variable `oneLongString` is set to the value `null`, the program will compile. However, at runtime, a `NullPointerException` will be thrown since the variable `oneLongString` will not reference any object. The golden rule is to ensure that a reference variable denotes an object before invoking methods via the reference, i.e. it is not `null`.

Arrays and their default values are discussed in Section 4.1 on page 88.

Review questions

2.7 Given the following code, which statement is true?

```
int a, b;
b = 5;
```

Select the one right answer.

(a) Variable `a` is not declared.
(b) Variable `b` is not declared.
(c) Variable `a` is declared but not initialized.
(d) Variable `b` is declared but not initialized.
(e) Variable `b` is initialized but not declared.

2.8 In which of these variable declarations will the variable remain uninitialized unless explicitly initialized?

Select all valid answers.

(a) Declaration of an instance variable of type `int`.
(b) Declaration of a static class variable of type `float`.
(c) Declaration of a local variable of type `float`.
(d) Declaration of a static class variable of type `Object`.
(e) Declaration of an instance variable of type `int[]`.

2.10 Java Source File Structure

A Java source file has the following elements, specified in the following order.

1. An optional `package` definition to specify a package name. The classes and interfaces defined in the file will belong to this package. If omitted, the definitions will belong to the *default package*. Packages are discussed in Section 4.5.
2. Zero or more `import` statements. The `import` statement is discussed in Section 4.5 on page 107.
3. Any number of class and interface definitions. Technically a source file need not have any such definitions, but that is hardly useful. The classes and interfaces can be defined in any order. Note that JDK imposes the restriction that only one `public` class definition per source file can be defined, and it requires that the file name match this `public` class. If the `public` class name is `NewApp` then the file name must be `NewApp.java`. Classes are discussed in Section 4.2, and interfaces are discussed in Section 6.4.

The above structure is depicted by a skeletal source file in Figure 2.2.

```
// Filename: NewApp.java

// PART 1: (OPTIONAL)
// Package name
package com.company.project.fragilePackage;

// PART 2: (ZERO OR MORE)
// Packages used
import java.util.*;
import java.io.*;

// PART 3: (ZERO OR MORE)
// Definitions of classes and interfaces (in any order)
public class NewApp { }
class C1 { }
interface I1 { }
// ...
class Cn { }
interface Im { }
// end of file
```

Figure 2.2 *Java Source File Structure*

Review questions

2.9 What will be the result of attempting to compile this class?

```
import java.util.*;

package com.acme.toolkit;

public class AClass {
    public Other anInstance;
}

class Other {
    int value;
}
```

Select the one right answer.

(a) The class will fail to compile, since the class `Other` has not yet been declared when referenced in class `AClass`.
(b) The class will fail to compile, since `import` statements must never be at the very top of a file.
(c) The class will fail to compile, since the `package` declaration can never occur after an `import` statement.
(d) The class will fail to compile, since the class `Other` must be defined in a file called `Other.java`.
(e) The class will fail to compile, since the class `Other` must be declared `public`.
(f) The class will compile without errors.

2.10 Is an empty file a valid source file?

Answer yes or no.

2.11 The `main()` Method

The Java interpreter executes a method called `main` in the class specified on the command line. This is the standard way in which a standalone application is invoked. The `main()` method has the following signature:

```
public static void main(String args[])
```

The command

```
java TooSmartClass
```

results in a call to the `TooSmartClass.main()` method. Note that any class can have a `main()` method. Only the `main()` method of the class specified to the Java interpreter is executed.

The `main()` Method Modifiers

The `main()` method always has `public` accessibility so that the interpreter can call it (Section 4.9, p. 115). It is a `static` method belonging to the class (Section 4.10, p. 121). It does not return a value, i.e. it is declared `void` (Section 5.4, p. 148). It always has an array of `String` objects as its only formal parameter. This array contains any arguments passed to the program on the command line (Section 3.23, p. 82). All this adds up to the following definition of the `main()` method:

```
...
public static void main(String args[]) {
    // ...
}
...
```

The requirements above do not exclude specification of additional modifiers (Section 4.10, p. 121).

Review questions

2.11 Which of these are valid declarations of the `main()` method?

Select all valid answers.

(a) `static void main(String args[]) { /* ... */ }`
(b) `public static int main(String args[]) { /* ... */ }`
(c) `public static void main(String args) { /* ... */ }`
(d) `final static public void main(String[] arguments) { /* ... */ }`
(e) `public int main(Strings args[], int argc) { /* ... */ }`
(f) `public void main(String args[]) { /* ... */ }`

Chapter summary

The following information was included in this chapter:

- Explanation of identifiers, keywords, literals, white spaces, and comments.
- Explanation of all the primitive datatypes in Java.
- Declaration, initialization and usage of variables, including reference variables.
- Usage of default values for member variables.
- Structure of a Java source file.
- Declaration of `main()` method.

Programming exercises

2.1 The following program has several errors. Modify it so that it will compile and run without errors.

```
import java.util.*;

package com.acme;

public class Exercise1 {
    int counter;

    void main(String args[]) {
        Exercise1 instance = new Exercise1();
        instance.go();
    }

    public void go() {
        int sum;
        int i = 0;
        while (i<100) {
            if (i == 0) sum = 100;
            sum = sum + i;
            i++;
        }
        System.out.println(sum);
    }
}
```

2.2 The following program has several errors. Modify it so that it will compile and run without errors.

```
// Filename: Temperature.java
PUBLIC CLASS temperature {
    PUBLIC void main(string args) {
        double fahrenheit = 62.5;
        */ Convert /*
        double celsius = f2c(fahrenheit);
        System.out.println(fahrenheit + 'F = ' + celsius + 'C');
    }

    double f2c(float fahr) {
        RETURN (fahr - 32) * 5 / 9;
    }
}
```