Language Fundamentals

2

Exam Objectives

- Identify correctly constructed `package` declarations, `import` statements, class declarations (of all forms, including inner classes), interface declarations, 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 4.6.*
  - *For class declarations, see Section 4.2.*
  - *For nested classes, see Chapter 7.*
  - *For interface declarations, see Section 6.4.*
  - *For method declarations, see Section 4.3.*
- Identify classes that correctly implement an interface where that interface is either `java.lang.Runnable` or a fully specified interface in the question.
  - *For interface implementation, see Section 6.4.*
  - *For implementation of* `java.lang.Runnable`, *see Section 9.3.*
- 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. Note: There will not be any questions regarding esoteric distinctions between keywords and manifest constants.
- 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.
  - *See also Appendix G.*

Supplementary Objectives

- State the wrapper classes for primitive data types.

2.1 Basic Language Elements

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, numbers, 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, variables, and labels.

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 `£`). However, the first character in an identifier cannot be a digit. Since Java programs are written in the Unicode character set (see p. 23), the definitions of letter and digit are interpreted according to this character set.

Identifiers in Java are case sensitive, for example, `price` and `Price` are two different identifiers.

Examples of Legal Identifiers:

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

Examples of Illegal Identifiers:

```
48chevy, all@hands, grand-sum
```

The name `48chevy` is not a legal identifier as it starts with a digit. The character `@` is not a legal character in an identifier. It is also not a legal operator so that `all@hands` cannot not be interpreted as a legal expression with two operands. The character `-` is also not a legal character in an identifier. However, it is a legal operator so `grand-sum` could be interpreted as a legal expression with two operands.

Keywords

Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. All the keywords are in lowercase, and 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: the `null` reference and the Boolean literals `true` and `false` (see Table 2.2). Keywords currently reserved, but not in use, are listed in Table 2.3. 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	default	implements	protected	throw
assert	do	import	public	throws
boolean	double	instanceof	return	transient
break	else	int	short	try
byte	extends	interface	static	void
case	final	long	strictfp	volatile
catch	finally	native	super	while
char	float	new	switch	
class	for	package	synchronized	
continue	if	private	this	

Table 2.2 *Reserved Literals in Java*

null	true	false

Table 2.3 *Reserved Keywords not Currently in Use*

const	goto

Literals

A *literal* denotes a constant value, that is, the value a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean or string values. In addition, there is the literal `null` that 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 data types are comprised of the following primitive data types: `int`, `long`, `byte`, and `short` (see Section 2.2).

The default data type of an integer literal is always `int`, but it can be specified as `long` by appending the suffix `L` (or `l`) to the integer value. Without the suffix, the `long` literals `2000L` and `0l` will be interpreted as `int` literals. There is no direct way to specify a `short` or a `byte` literal.

In addition to the decimal number system, integer literals can also be specified in octal (*base* 8) and hexadecimal (*base* 16) number systems. Octal and hexadecimal numbers are specified with `0` and `0x` (or `0X`) prefix respectively. Examples of decimal, octal and hexadecimal literals are shown in Table 2.5. Note that the leading `0` (zero) digit is not the uppercase letter `O`. The hexadecimal digits from `a` to `f` can also be specified with the corresponding uppercase forms (`A` to `F`). Negative integers (e.g. `-90`) can be specified by prefixing the minus sign (`-`) to the magnitude of the integer regardless of number system (e.g., `-0132` or `-0X5A`). Number systems and number representation are discussed in Appendix G. Java does not support literals in binary notation.

Table 2.5 *Examples of Decimal, Octal, and Hexadecimal Literals*

Decimal	Octal	Hexadecimal
8	010	0x8
10L	012L	0XaL
16	020	0x10
27	033	0x1B
90L	0132L	0x5aL
-90	-0132	-0X5A
2147483647 (i.e., 2^{31}-1)	017777777777	0x7fffffff
-2147483648 (i.e., -2^{31})	-020000000000	-0x80000000
1125899906842624L (i.e., 2^{50})	040000000000000000L	0x4000000000000L

Floating-point Literals

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

The default data type of a floating-point literal is `double`, but it 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, where `E` (or `e`) stands for *Exponent*. For example, the `double` literal `194.9E-2` in scientific notation is interpreted as `194.9*10`$^{-2}$ (i.e., `1.949`).

Examples of `double` Literals

```
0.0      0.0d      0D
0.49     .49       .49D
49.0     49.       49D
4.9E+1   4.9E+1D   4.9e1d   4900e-2  .49E2
```

Examples of `float` Literals

```
0.0F     0f
0.49F    .49F
49.0F    49.F      49F
4.9E+1F  4900e-2f  .49E2F
```

Note that the decimal point and the exponent are optional and that at least one digit must be specified.

Boolean Literals

The primitive data type `boolean` represents the truth-values *true* or *false* that are denoted by the reserved literals `true` or `false`, respectively.

Character Literals

A character literal is quoted in single-quotes ('). All character literals have the primitive data type `char`.

Characters in Java are represented by the 16-bit Unicode character set, which subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. In Table 2.6, note that digits (`0` to `9`), upper-case letters (`A` to `Z`), and lower-case letters (`a` to `z`) have contiguous Unicode values. Any Unicode character can be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix `\u`.

Table 2.6 *Examples of Unicode Values*

Character Literal	Character Literal using Unicode value	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

Continues

Table 2.6 *Examples of Unicode Values (Continued)*

Character Literal	Character Literal using Unicode value	Character
'z'	'\u007a'	z
'Ñ'	'\u0084'	Ñ
'å'	'\u008c'	å
'ß'	'\u00a7'	ß

Escape Sequences

Certain *escape sequences* define special character values as shown in Table 2.7. These escape sequences can be single-quoted to define character literals. For example, the character literals '\t' and '\u0009' are equivalent. However, the character literals '\u000a' and '\u000d' should not be used to represent newline and carriage return in the source code. These values are interpreted as line-terminator characters by the compiler, and will cause compile time errors. One should use the escape sequences '\n' and '\r', respectively, for correct interpretation of these characters in the source code.

Table 2.7 *Escape Sequences*

Escape Sequence	Unicode Value	Character
\b	\u0008	Backspace (BS)
\t	\u0009	Horizontal tab (HT or TAB)
\n	\u000a	Linefeed (LF) a.k.a., Newline (NL)
\f	\u000c	Form feed (FF)
\r	\u000d	Carriage return (CR)
\'	\u0027	Apostrophe-quote
\"	\u0022	Quotation mark
\\	\u005c	Backslash

We can also use the escape sequence *ddd* to specify a character literal by octal value, where each digit *d* can be any octal digit (0–7), as shown in Table 2.8. The number of digits must be three or fewer, and the octal value cannot exceed \377, that is, only the first 256 characters can be specified with this notation.

Table 2.8 *Examples of Escape Sequence* `\ddd`

Escape Sequence `\ddd`	Character Literal
`'\141'`	`'a'`
`'\46'`	`'&'`
`'\60'`	`'0'`

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. All string literal are objects of the class `String` (see Section 10.5, p. 407).

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)
"Left!\nRight!"                                                (4)
```

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. In (4), we use the escape sequence `\n` to insert a newline. 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."
Left!
Right!
```

One should also use the string literals `"\n"` and `"\r"`, respectively, for correct interpretation of the characters `"\u000a"` and `"\u000d"` in the source code.

White Spaces

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

A Java program is a free-format sequence of characters that is *tokenized* by the compiler, that is, 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 as separators. 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: `// ... to the end of the line`
- A multiple-line comment: `/* ... */`
- A documentation (Javadoc) comment: `/** ... */`

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.
int age;       // From comment-start sequence to the end of the line is a comment.
```

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.
*/
```

The comment-start sequences (`//`, `/*`, `/**`) are not treated differently from other characters when occurring within comments, and are thus ignored. This means trying to nest multiple-line comments will result in compile time error:

```
/*  Formula for alchemy.
    gold = wizard.makeGold(stone);
    /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence `/*` is ignored. The last occurrence of the sequence `*/` in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A *documentation comment* is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the `javadoc` tool to generate HTML documentation for the program. Documentation comments are

usually placed in front of classes, interfaces, methods and field 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 2.0
 */
```

For details on the javadoc tool, see the documentation for the tools in the Java 2 SDK.

Review Questions

2.1 Which of the following is not a legal identifier?

Select the one correct answer.

(a) a2z
(b) ödipus
(c) 52pickup
(d) _class
(e) ca$h
(f) total#

2.2 Which statement is true?

Select the one correct 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 correct 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.

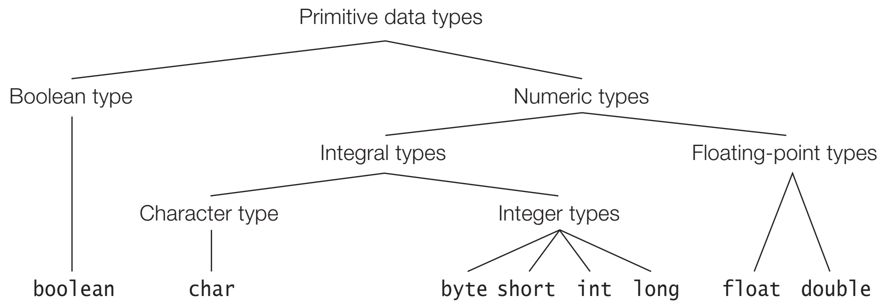
2.2 Primitive Data Types

Figure 2.1 gives an overview of the primitive data types in Java.

Primitive data types in Java can be divided into three main categories:

- *Integral types*—represent signed integers (`byte`, `short`, `int`, `long`) and unsigned character values (`char`)
- *Floating-point types* (`float`, `double`)—represent fractional signed numbers
- *Boolean type* (`boolean`)—represent logical values

Figure 2.1 *Primitive Data Types in Java*



Primitive data values are not objects. Each primitive data type defines the range of values in the data type, and operations on these values are defined by special operators in the language (see Chapter 3).

Each primitive data type also 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.

Integer Types

Table 2.9 *Range of Integer Values*

Data Type	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 data types are `byte`, `short`, `int`, and `long` (see Table 2.9). Their values are signed integers represented by 2's complement (see Section G.4, p. 598).

Character Type

Table 2.10 *Range of Character Values*

Data Type	Width (bits)	Minimum Unicode value	Maximum Unicode value
char	16	0x0 (\u0000)	0xffff (\uffff)

Characters are represented by the data type `char` (see Table 2.10). Their values are unsigned integers that denote all the 65536 (2^{16}) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters.

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.

Floating-point Types

Table 2.11 *Range of Floating-point Values*

Data Type	Width (bits)	Minimum Positive Value MIN_VALUE	Maximum Positive Value MAX_VALUE
float	32	1.401298464324817E-45f	3.402823476638528860e+38f
double	64	4.94065645841246544e-324	1.79769313486231570e+308

Floating-point numbers are represented by the `float` and `double` data types.

Floating-point numbers conform to the IEEE 754-1985 binary floating-point standard. Table 2.11 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. For example, the value of the expression `(1.0/3.0)` is represented as an approximation due to the finite number of bits used.

Boolean Type

Table 2.12 *Boolean Values*

Data Type	Width	True Value Literal	False Value Literal
`boolean`	not applicable	`true`	`false`

The data type `boolean` represents the two logical values denoted by the literals `true` and `false` (see Table 2.12).

Boolean values are produced by all *relational* (see Section 3.9), *conditional* (see Section 3.12) and *boolean logical operators* (see Section 3.11), and are primarily used to govern the flow of control during program execution.

Table 2.13 summarizes the pertinent facts about the primitive data types: their width or size, which indicates the number of the bits required to store a primitive value; their range (of legal values), which is specified by the minimum and the maximum values permissible; and the name of the corresponding wrapper class.

Table 2.13 *Summary of Primitive Data Types*

Data Type	Width (bits)	Minimum Value, Maximum Value	Wrapper Class
`boolean`	not applicable	`true`, `false` (no ordering implied)	`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-45f`, `±3.402823476638528860e+38f`	`Float`
`double`	64	`±4.94065645841246544e-324`, `±1.79769313486231570e+308`	`Double`

Review Questions

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

Select the two correct answers.

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

2.5 Which of the following primitive data types are not integer types?

Select the three correct answers.

(a) Type `boolean`
(b) Type `byte`
(c) Type `float`
(d) Type `short`
(e) Type `double`

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

Select the one correct answer.

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

2.3 Variable Declarations

A *variable* stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can only store values of primitive data types and references to objects. Variables that store references to objects are called *reference variables*.

Declaring and Initializing Variables

Variable declarations are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. We show some examples of declaring variables that can store primitive values:

```
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.
```

The first declaration above is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also include initialization code to specify an appropriate 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.
```

Object Reference Variables

An *object reference* is a value that denotes an object in Java. Such reference values can be stored in variables and used to manipulate the object denoted by the reference value.

A variable declaration that specifies a *reference type* (i.e., a class, an array, or an interface name) declares an object reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration only specifies the name and the reference type. The declaration determines what objects a reference variable can denote. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

```
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 any objects of class `Pizza` or `Hamburger`. The declarations only create variables that can store references to objects of these classes.

A declaration can also include an initializer to create an object whose reference 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.

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

Reference variables for arrays are discussed in Section 4.1.

Lifetime of Variables

Lifetime of a variable, that is, the time a variable is accessible during execution, is determined by the context in which it is declared. We distinguish between lifetime of variables in three contexts:

- *Instance variables*—members of a class and created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the *state* of the object. Instance variables exist as long as the object they belong to exists.

- *Static variables*—also members of a class, but not created for any object of the class and, therefore, belong only to the class (see Section 4.10, p. 144). They are created when the class is loaded at runtime, and exist as long as the class exists.
- *Local variables* (also called *method automatic variables*)—declared in methods and in blocks and created for each execution of the method or block. After the execution of the method or block completes, local (non-`final`) variables are no longer accessible.

2.4 Initial Values for Variables

Default Values for Fields

Default values for fields of primitive data types and reference types are listed in Table 2.14. The value assigned depends on the type of the field.

Table 2.14 *Default Values*

Data Type	Default Value
`boolean`	`false`
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0L` for `long`, `0` for others
Floating-point (`float`, `double`)	`0.0F` or `0.0D`
Reference types	`null`

If no initialization is provided for a static variable either in the declaration or in a static initializer block (see Section 8.2, p. 336), it is initialized with the default value of its type when the class is loaded.

Similarly, if no initialization is provided for an instance variable either in the declaration or in an instance initializer block (see Section 8.2, p. 338), it is initialized with the default value of its type when the class is instantiated.

The fields of reference types are always initialized with the `null` reference value, if no initialization is provided.

Example 2.1 illustrates default initialization of fields. 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`.

Example 2.1 *Default Values for Fields*

```
public 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 variable counter: "     + Light.counter);
        System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
        System.out.println("Instance variable indicator: " + bulb.indicator);
        System.out.println("Instance variable location: "  + bulb.location);
        return;
    }
}
```

Output from the program:

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

Initializing Local Variables of Primitive Data Types

Local variables are *not* initialized when they are created at method invocation, that is, when the execution of a method is started. They must be explicitly initialized before being used. The compiler will report attempts to use uninitialized local variables.

Example 2.2 *Flagging Uninitialized Local Variables of Primitive Data Types*

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

        if (weight <  10) thePrice = 1000;
        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` used in the `println` statement at (1) may not be initialized. However, it can be seen that at runtime the local variable `thePrice` will get 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. It only compiles the body of a conditional statement if it can deduce the condition to be `true`. The program will compile correctly if the variable is initialized in the declaration, or if an unconditional assignment is made to the variable. Replacing the declaration of the local variables in Example 2.2 with the following declaration solves the problem:

```
int weight = 10, thePrice = 0;        // Both local variables initialized.
```

Initializing Local Reference Variables

Local reference variables are bound by the same initialization rules as local variables of primitive data types.

Example 2.3 *Flagging Uninitialized Local Reference Variables*

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

        System.out.println("The message length is: " + importantMessage.length());
    }
}
```

In Example 2.3, the compiler complains that the local variable `importantMessage` used in the `println` statement may not be initialized. If the variable `importantMessage` is set to the value `null`, the program will compile. However, at runtime, a `NullPointerException` will be thrown since the variable `importantMessage` will not denote any object. The golden rule is to ensure that a reference variable, whether local or not, is assigned a reference to an object before it is used, that is, ensure that it does not have the value `null`. The program compiles and runs if we replace the declaration with the following declaration, which creates a string literal and assigns its reference to the local reference variable `importantMessage`:

```
String importantMessage = "Initialize before use!";
```

Arrays and their default values are discussed in Section 4.1 on page 101.

Review Questions

2.7 Which of the following lines are valid declarations?

Select the three correct answers.

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

2.8 Given the following code within a method, which statement is true?

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

Select the one correct answer.

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

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

Select the one correct answer.

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

2.5 Java Source File Structure

The structure of a skeletal Java source file is depicted in Figure 2.2. A Java source file can have the following elements that, if present, must be specified in the following order:

1. An optional `package` declaration to specify a package name. Packages are discussed in Section 4.6.

2. Zero or more `import` declarations. Since `import` declarations introduce class and interface names in the source code, they must be placed before any type declarations. The `import` statement is discussed in Section 4.6.
3. Any number of *top-level* class and interface declarations. Since these declarations belong to the same package, they are said to be defined at the *top level*, which is the package level.

   The classes and interfaces can be defined in any order. Class and interface declarations are collectively known as *type declarations*. Technically, a source file need not have any such definitions, but that is hardly useful.

   The Java 2 SDK imposes the restriction that at the most one `public` class definition per source file can be defined. If a `public` class is defined, the file name must 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.

Note that except for the `package` and the `import` statements, all code is encapsulated in classes and interfaces. No such restriction applies to comments and white space.

Figure 2.2 *Java Source File Structure*

```
// Filename: NewApp.java
```

```
// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilePackage;
```

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

```
// PART 3: (ZERO OR MORE) top-level class and interface declarations
public class NewApp { }

class AClass { }

interface IOne { }

class BClass { }

interface ITwo { }
// ...
// end of file
```

Review Questions

2.10 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 correct 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.

2.11 Is an empty file a valid source file?

Answer true or false.

2.6 The `main()` Method

The mechanics of compiling and running Java applications using the Java 2 SDK are outlined in Section 1.10. The Java interpreter executes a method called `main` in the class specified on the command line. Any class can have a `main()` method, but only the `main()` method of the class specified to the Java interpreter is executed to start a Java application.

The `main()` method must have `public` accessibility so that the interpreter can call it (see Section 4.9, p. 138). It is a static method belonging to the class, so that no object of the class is required to start the execution (see Section 4.10, p. 144). It does not return a value, that is, it is declared `void` (see Section 5.4, p. 176). 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 (see Section 3.23, p. 95). All this adds up to the following definition of the `main()` method:

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

The above requirements do not exclude specification of additional modifiers (see Section 4.10, p. 144) or any `throws` clause (see Section 5.9, p. 201). The `main()` method can also be overloaded like any other method (see p. 116). The Java interpreter ensures that the `main()` method, that complies with the above definition is the starting point of the program execution.

Review Questions

2.12 Which of these are valid declarations of the `main()` method in order to start the execution of a Java application?

Select the two correct answers.

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

2.13 Which of the following are reserved keywords?

Select the three correct answers.

(a) `public`
(b) `static`
(c) `void`
(d) `main`
(e) `String`
(f) `args`

Chapter Summary

The following topics were discussed in this chapter:

- basic language elements: identifiers, keywords, literals, white spaces, and comments
- primitive data types: integral, floating-point, and Boolean
- converting numbers between decimal, octal, and hexadecimal systems
- lifetime of fields and local variables
- declaration and initialization of variables, including reference variables
- usage of default values for fields

- structure of a valid Java source file
- declaration of the `main()` method whose execution starts the application

Programming Exercises

2.1 The following program has several errors. Modify it so that it will compile and run without errors. (See Section 4.6 on page 129 for compiling and running code from packages.)

```
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;
    }
}
```