Module 3

Identifiers, Keywords, and Types

Objectives

Upon completion of this module, you should be able to:

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *primitive variable* and *reference variable*
- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Describe the significance of a reference variable
- State the consequence of assigning variables of class type

This module describes some of the basic components used in Java technology programs including variables, keywords, primitive types, and class types.

Relevance

Discussion – The following questions are relevant to the material presented in this module:

- Do you know the primitive Java technology types?

- Can you describe the difference between variables holding primitive values as compared with object references?

Comments

The three permissible styles for inserting comments are:

```
// comment on one line

/* comment on one
 * or more lines
 */

/** documentation comment
 *  can also span one or more lines
 */
```

Documentation comments placed immediately before a declaration (of a variable, method, or class) indicate that the comments should be included in any documentation that is generated automatically (for example, the HTML files generated by the `javadoc` command) to serve as a description of the declared item.

Note – The format of these comments and the use of the Javadoc™ tool is described in the documentation for Java 2 SDK. Refer to the following Universal Resource Locator (URL):
`http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/`

Semicolons, Blocks, and White Space

In the Java programming language, a statement is one or more lines of code terminated with a semicolon (;).

For example,

```
totals = a + b + c + d + e + f;
```

is the same as

```
totals = a + b + c
         + d + e + f;
```

A *block*, sometimes called a compound statement, is a group of statements bound by opening and closing braces ({ }). A block has useful features that are described in Module 4, "Expressions and Flow Control." For now, just think of a block as a group of statements that are collected together.

The following are other examples of block statements or groupings:

```
// a block statement
{
  x = y + 1;
  y = x + 1;
}

// a class definition is contained in a block
public class MyDate {
  private int day;
  private int month;
  private int year;
}

// a block statement can be nested within
// another block statement
while ( i < large ) {
  a = a + i;
  // nested block
  if ( a == max ) {
    b = b + a;
    a = 0;
  }
  i = i + 1;
}
```

You can have *white space* between elements of the source code. Any amount of white space is allowed. You can use white space, including spaces, tabs, and new lines, to enhance the clarity and visual appearance of your source code. Compare:

```
{int x;x=23*54;}
```

with:

```
{
  int x;

  x = 23 * 54;
}
```

Identifiers

In the Java programming language, an *identifier* is a name given to a variable, class, or method. Identifiers start with a letter, underscore (_), or dollar sign ($). Subsequent characters can be digits. Identifiers are case-sensitive and have no maximum length.

The following are valid identifiers:

- `identifier`
- `userName`
- `user_name`
- `_sys_var1`
- `$change`

Java technology sources are in 16-bit Unicode rather than 8-bit American Standard Code for Information Interchange (ASCII) text, so a letter is a considerably wider definition than just a–z and A–Z.

While identifiers can use non-ASCII characters, be aware of the following caveats:

- Unicode can support *different* characters that look the same.
- Class names should *only* be in ASCII characters because most file systems do not support Unicode characters.

An identifier cannot be a keyword, but it can contain a keyword as part of its name. For example, `thisOne` is a valid identifier, but `this` is not because `this` is a Java technology keyword.

Note – Identifiers containing a dollar sign ($) are generally unusual, although languages such as BASIC, along with VAX/VMS systems, make extensive use of them. Because they are unfamiliar, it is probably best to avoid them unless there is a local convention or other pressing reason for including this symbol in the identifier.

Java Programming Language Keywords

Keywords have special meaning to the Java technology compiler. They identify a data type name or program construct name.

Table 3-1 lists keywords that are used in the Java programming language.

Table 3-1 Java Programming Language Keywords

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

Note – While you might think `true` and `false` are keywords, they are in fact Boolean literals, according to *The Java Language Specification* (Section 3.10.3). Similarly, `null` is in fact the `null` literal (Section 3.10.7).

The following are important notes about the keywords:

- The literals `true`, `false`, and `null` are lowercase, not uppercase as in the C++ language. Strictly speaking, these are not keywords but literals; however, the distinction is academic.
- There is no `sizeof` operator; direct memory access is not possible so the information would be valueless.
- The `goto` and `const` keywords are not used in the Java programming language.

Basic Java Programming Language Types

The Java programming language has many built in data types. These fall into two broad categories: class types and primitive types. Primitive types are simple values, are not objects. Class types are used for more complex types, including all of the types that you declare yourself. Class types are used to create objects.

Primitive Types

The Java programming language defines eight *primitive* data types, which can be considered in four categories:

- Logical – `boolean`
- Textual – `char`
- Integral – `byte`, `short`, `int`, and `long`
- Floating point – `double` and `float`

Logical – `boolean`

Logical values are represented using the `boolean` type, which takes one of two values: true or false. These values can be used to represent any two states, such as on and off, or yes and no. The `boolean` type has two literal values: `true` and `false`. The following code is an example of the declaration and initialization of a `boolean` type variable:

```
// declares the variable truth as boolean and
// assigns it the value true
boolean truth = true;
```

Note – There are no casts between integer types and the `boolean` type. Some languages, most notably C and C++, allow numeric values to be interpreted as logical values. This is not permitted in the Java programming language; when a `boolean` type is required, you can use only `boolean` values.

Textual – `char`

Single characters are represented by using the `char` type. A `char` represents a 16-bit, unsigned Unicode character. You must enclose a `char` literal in single quotes (' '). For example:

`‘a’`	The letter a
`‘\t’`	A tab
`‘\u????’`	A specific Unicode character, `????`, is replaced with exactly four hexadecimal digits. For example, `’\u03A6’` is the Greek letter phi [Φ].

Textual – `String`

You use the `String` type, which is not a primitive but a class, to represent sequences of characters. The characters themselves are Unicode, and the backslash notation shown previously for the `char` type also works in a `String`. Unlike C and C++, strings do not end with `\0`.

A `String` literal is enclosed in double quote marks:

```
"The quick brown fox jumps over the lazy dog."
```

Some examples of the declarations and initialization of `char` and `String` type variables are:

```
// declares and initializes a char variable
char ch = ’A’;

// declares two char variables
char ch1,ch2;

// declares two String variables and initializes them
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";

// declares two String variables
String str1,str2;
```

Integral – byte, short, int, and long

There are four integral types in the Java programming language. Each type is declared using one of the keywords `byte`, `short`, `int`, or `long`. You can represent literals of integral type using decimal, octal, or hexadecimal forms as follows:

`2`	This is the decimal form for the integer `2`.
`077`	The leading `0` indicates an octal value.
`0xBAAC`	The leading `0x` indicates a hexadecimal value.

All numeric types in the Java programming language represent signed numbers.

Integral literals are of type `int` unless followed explicitly by the letter `L` that indicates a `long` value. In the Java programming language, you can use either an uppercase or lowercase `L`. A lowercase `l` is not recommended because it is hard to distinguish from the digit `1`.

Long versions of the literals shown previously are:

`2L`	The `L` indicates that the decimal value `2` is represented as a long value.
`077L`	The leading `0` indicates an octal value.
`0xBAACL`	The `0x` prefix indicates a hexadecimal value.

The size and range for the four integral types are shown in Table 3-2. The range representation is defined by the Java programming language specification as a 2's complement and is platform-independent.

Table 3-2 Integral Data Types – Size and Range

Integer Length	**Name or Type**	**Range**
8 bits	`byte`	From -2^{7} to $2^{7}-1$
16 bits	`short`	From -2^{15} to $2^{15}-1$
32 bits	`int`	From -2^{31} to $2^{31}-1$
64 bits	`long`	From -2^{63} to $2^{63}-1$

Floating Point – `float` and `double`

You can declare a floating-point variable using the keywords `float` or `double`. The following list contains examples of floating-point numbers. A numeric literal is a floating point if it includes either a decimal point or an exponent part (the letter `E` or `e`), or is followed by the letter `F` or `f` (float) or the letter `D` or `d` (double). Some examples of floating-point literals include:

`3.14`	A simple floating-point value (a double)
`6.02E23`	A large floating-point value
`2.718F`	A simple `float` size value
`123.4E+306D`	A large `double` value with redundant `D`

Note – The `23` after the `E` in the second example is implicitly positive. That example is equivalent to `6.02E+23`.

Floating point literals are `double` by default. You can declare a literal of type `float` by appending `F` or `f` to the value.

The format of a floating point number is defined by *The Java Language Specification* to be Institute of Electrical and Electronics Engineers (IEEE) 754, using the sizes shown in Table 3-3. This format is platform independent.

Table 3-3 Floating Point Data Type Size

Float Length	**Name or Type**
32 bits	`float`
64 bits	`double`

Note – Floating point literals are `double` unless declared explicitly as `float`.

Variables, Declarations, and Assignments

The following program illustrates how to declare and assign values to int, float, boolean, char, and String type variables:

```
public class Assign {
  public static void main (String args []) {
    // declare integer variables
    int x, y;
    // declare and assign floating point
    float z = 3.414f;
    // declare and assign double
    double w = 3.1415;
    // declare and assign boolean
    boolean truth = true;
    // declare character variable
    char c;
    // declare String variable
    String str;
    // declare and assign String variable
    String str1 = "bye";
    // assign value to char variable
    c = 'A';
    // assign value to String variable
    str = "Hi out there!";
    // assign values to int variables
    x = 6;
    y = 1000;
  }
}
```

The following are examples of illegal assignments:

```
y = 3.1415926;
// 3.1415926 is not an int
// It requires casting and decimal will be truncated.

w = 175,000;
// The comma symbol (,) cannot appear;

truth = 1;
// this is a common mistake made by ex-C / C++ programmers

z = 3.14156;
// Can't fit double into a float. This requires casting.
```

Java Reference Types

As you have seen, there are eight primitive Java types: boolean, char, byte, short, int, long, float, and double. All other types refer to objects rather than primitives. Variables that refer to objects are *reference variables*. For example, you can define a class MyDate:

```
public class MyDate {
  private int day = 1;
  private int month = 1;
  private int year = 2000;
  public MyDate(int day, int month, int year) { ... }
  public String toString() { ... }
}
```

The following is an example of using MyDate:

```
public class TestMyDate {
  public static void main(String[] args) {
    MyDate today = new MyDate(22, 7, 1964);
  }
}
```

The variable today is a reference variable holding one MyDate object.

Constructing and Initializing Objects

You have seen how you must execute a call to `new Xyz()` to allocate space for a new object. You will see that sometimes you can place arguments in the parentheses, for example `new MyDate(22, 7, 1964)`.

Using the keyword `new` causes the following:

1. First, the space for the new object is allocated and initialized to the form of `0` or null. In the Java programming language, this phase is indivisible to ensure that you cannot have an object with random values in it.
2. Second, any explicit initialization is performed.
3. Third, a *constructor*, which is a special method, is executed. Arguments passed in the parentheses to `new` are passed to the constructor (`22`, `7`, `1964`).
4. Finally, the return value from the new operation is a reference to the new object in heap memory. This reference is stored in the reference variable.

Memory Allocation and Layout

In a method body, the following declaration allocates storage only for the reference shown in Figure 3-1:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

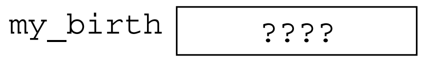

Figure 3-1 Declaration of the Object Reference

The keyword new in the following example implies allocation and initialization of storage, as shown in Figure 3-2.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

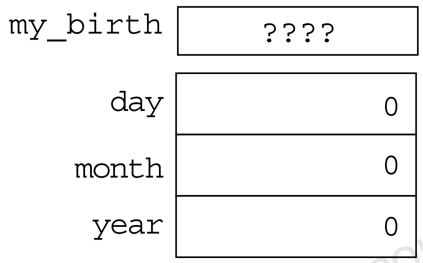

Figure 3-2 Using the new Keyword

Explicit Attribute Initialization

If you place simple assignment expressions in your member declarations, you can initialize members explicitly during construction of your object.

In the MyDate class in this example, initializing all three attributes explicitly is declared, as shown in Figure 3-3:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

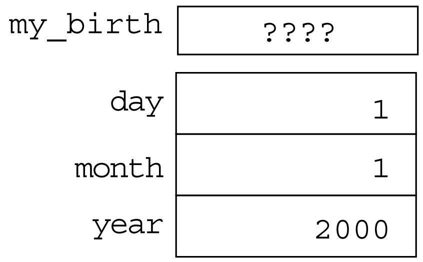

Figure 3-3 Explicit Initialization

Executing the Constructor

The final stage of initializing a new object is to call the constructor. The constructor enables you to override the default initialization. You can perform computations. You can also pass arguments into the construction process so that the code that requests the construction of the new object can control the object it creates.

The following example calls the constructor, as shown in Figure 3-4:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

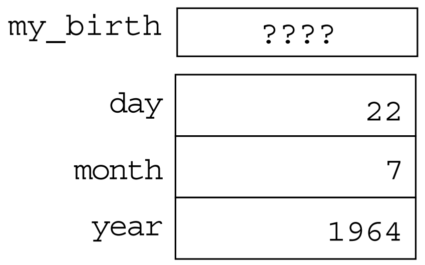

Figure 3-4 Constructor Call

Assigning a Variable

The variable assignment then sets up the reference variable `my_birth` in this example, so that it refers properly to the newly created object as shown in Figure 3-5.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

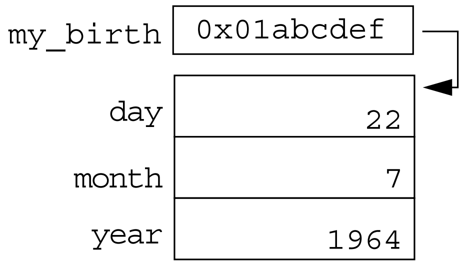

Figure 3-5 Reference Variable Assignment

This Is Not the Whole Story

It turns out that constructing and initializing objects is more complex than is described here. This topic is revisited in Module 6.

Assigning References

In the Java programming language, a variable declared with a type of class is referred to as a reference type because it refers to a non-primitive type. This has consequences for the meaning of the assignment. Consider this code fragment:

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```

Four variables are created: two primitives of type `int` and two references of type `MyDate`. The value of `x` is 7, and this value is copied into `y`. Both `x` and `y` are independent variables, and further changes to either do not affect the other.

With the variables `s` and `t`, only one `MyDate` object exists, and it contains the date 22 July 1964. Both `s` and `t` refer to that single object, as shown in Figure 3-6.

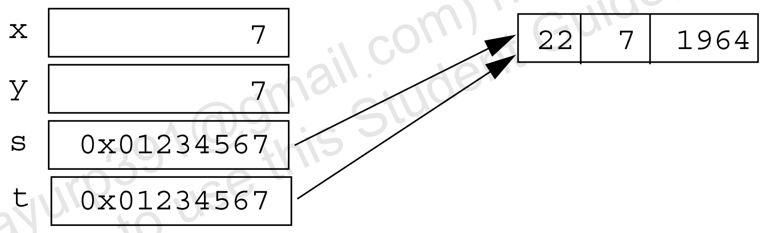

Figure 3-6 Two Variables Refer to the Same Reference Object

With a reassignment of the variable `t`, the new `MyDate` object (for 22 December 1964) is created and `t` refers to this object, as shown in Figure 3-7. This scenario is depicted as:

```
t = new MyDate(22, 12, 1964); // reassign the variable
```

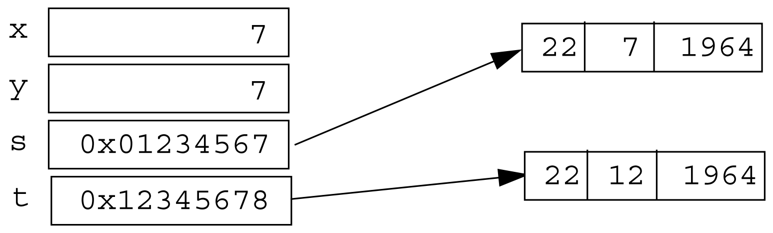

Figure 3-7 Variable `t` Refers to a New Object

Pass-by-Value

The Java programming language passes arguments only *by value*, that is, you cannot change the argument value in the calling method from within the called method.

However, when an object instance is passed as an argument to a method, the value of the argument is not the object itself but a reference to the object. You can change the *contents* of the object in the called method but not the object reference.

To many people, this looks like pass-by-reference, and behaviorally, it has much in common with pass-by-reference. However, there are two reasons this is inaccurate. First, the ability to change the *thing* passed into a method only applies to objects, not primitive values. Second, the actual value associated with a variable of object type is the reference to the object, and not the object itself. This is an important distinction in other ways, and if clearly understood, is entirely supporting of the point that the Java programming language passes arguments by value.

The following code example illustrates this point:

```
public class PassTest {

  // Methods to change the current values
  public static void changeInt(int value) {
    value = 55;
  }
  public static void changeObjectRef(MyDate ref) {
    ref = new MyDate(1, 1, 2000);
  }
  public static void changeObjectAttr(MyDate ref) {
    ref.setDay(4);
  }

  public static void main(String args[]) {
    MyDate date;
    int val;

    // Assign the int
    val = 11;
    // Try to change it
```

```
      changeInt(val);
      // What is the current value?
      System.out.println("Int value is: " + val);

      // Assign the date
      date = new MyDate(22, 7, 1964);
      // Try to change it
      changeObjectRef(date);
      // What is the current value?
      System.out.println("MyDate: " + date);

      // Now change the day attribute
      // through the object reference
      changeObjectAttr(date);
      // What is the current value?
      System.out.println("MyDate: " + date);
    }
  }
```

This code outputs the following:

```
java PassTest

Int value is: 11
MyDate: 22-7-1964
MyDate: 4-7-1964
```

The `MyDate` object is not changed by the `changeObjectRef` method; however, the `changeObjectAttr` method changes the day attribute of the `MyDate` object.

The `this` Reference

Two uses of the `this` keyword are:

- To resolve ambiguity between instance variables and parameters
- To pass the current object as a parameter to another method

Code 3-1 provides a class definition that demonstrates these uses. The `MyDate` class declares instance variables, Lines 2–4. One of the parameters to one of the constructors (Lines 6–10) is also called `day`, so in that context, the keyword `this` resolves the ambiguity (Line 7). The `addDays` method creates a new date object (Line 18). In this constructor call, the method uses the `this` keyword as an argument to refer to the current object.

Code 3-1 The Use of the `this` Keyword

```
1   public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
5
6     public MyDate(int day, int month, int year) {
7       this.day   = day;
8       this.month = month;
9       this.year  = year;
10    }
11    public MyDate(MyDate date) {
12      this.day   = date.day;
13      this.month = date.month;
14      this.year  = date.year;
15    }
16
17    public MyDate addDays(int moreDays) {
18      MyDate newDate = new MyDate(this);
19      newDate.day = newDate.day + moreDays;
20      // Not Yet Implemented: wrap around code...
21      return newDate;
22    }
23    public String toString() {
24      return "" + day + "-" + month + "-" + year;
25    }
26  }
```

Java Programming Language Coding Conventions

The following are the coding conventions of the Java programming language:

- *Packages* – Package names should be nouns in lowercase.

  ```
  package shipping.objects
  ```

- *Classes* – Class names should be nouns, in mixed case, with the first letter of each word capitalized.

  ```
  class AccountBook
  ```

- *Interfaces* – Interface names should be capitalized like class names.

  ```
  interface Account
  ```

- *Methods* – Method names should be verbs, in mixed case, with the first letter in lowercase. Within each method name, capital letters separate words. Limit the use of underscores.

  ```
  balanceAccount()
  ```

- *Variables* – All variables should be in mixed case with a lowercase first letter. Words are separated by capital letters. Limit the use of underscores, and avoid using the dollar sign ($) because this character has special meaning to inner classes.

  ```
  currentCustomer
  ```

  Variables should be meaningful and indicate to the casual reader the intent of their use. Avoid single character names except for temporary *throwaway* variables (for example, `i`, `j`, and `k`, used as loop control variables).

- *Constants* – Primitive constants should be all uppercase with the words separated by underscores. Object constants can use mixed-case letters.

  ```
  HEAD_COUNT
  MAXIMUM_SIZE
  ```

- *Control structures* – Use braces (`{ }`) around all statements, even single statements, when they are part of a control structure, such as an `if-else` or `for` statement.

  ```
  if ( condition ) {
    statement1;
  } else {
    statement2;
  }
  ```

- *Spacing* – Place only a single statement on any line, and use two-space or four-space indentations to make your code readable. The number of spaces can vary depending on what code standards you use.
- *Comments* – Use comments to explain code segments that are not obvious. Use the // comment delimiter for normal commenting; you can comment large sections of code using the `/* . . . */` delimiters. Use the `/** . . . */` documenting comment to provide input to `javadoc` for generating HTML documentation for the code.

```
// A comment that takes up only one line.

/* Comments that continue past one line and take up
   space on multiple lines. */

/** A comment for documentation purposes.
 * @see Another class for more information
 */
```

Note – The `@see` tag is a special `javadoc` tag giving the effect of a *see also* link that references a class or method. For more information about `javadoc`, refer to the complete definition of the documentation system in *The Design of Distributed Hyperlinked Programming Documentation*, a paper by Lisa Friendly. It is available at the following URL:
`http://java.sun.com/docs/javadoc-paper.html`

Also, for more information on Sun's Java technology coding conventions, refer to the following Web page:
`http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`