Module 4

Expressions and Flow Control

Objectives

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

- Distinguish between instance and local variables
- Describe how to initialize instance variables
- Identify and correct a `Possible reference before assignment compiler` error
- Recognize, describe, and use Java software operators
- Distinguish between legal and illegal assignments of primitive types
- Identify `boolean` expressions and their requirements in control constructs
- Recognize assignment compatibility and required casts in fundamental types
- Use `if`, `switch`, `for`, `while`, and `do` constructions and the labeled forms of `break` and `continue` as flow control structures in a program

This module describes variables, operators, and arithmetic expressions, and lays out the different control structures governing the path of execution.

Relevance

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

- What types of variables are useful to programmers?

  _____

  _____

  _____

- Can multiple classes have variables with the same name and, if so, what is their scope?

  _____

  _____

  _____

- What types of control structures are used in other languages? What methods do these languages use to control flow (such as in a loop or switch)?

  _____

  _____

  _____

Variables

This section describes variables are declared, scoped, and initialized in the Java programming language.

Variables and Scope

You have seen two ways to describe variables: variables of primitive type or variables of reference type. You have also seen two places to declare variables: inside a method or outside a method but within a *class* definition.

Variables defined inside a method are called *local* variables, but are sometimes referred to as *automatic, temporary,* or *stack* variables. You must initialize local variables explicitly before the first use. Method parameters and constructor parameters are also local variables but they are initialized by the calling code.

Variables defined outside a method are created when the object is constructed using the `new Xxx()` call. There are two possible types of these variables. The first kind is a class variable that is declared using the `static` keyword. Variables labeled `static` are created when the class is loaded and continue to exist for as long as the class is loaded. The second type is an instance variable that is declared without the `static` keyword. Instance variables continue to exist for as long as the object exists. Instance variables are sometimes referred to as member variables, because they are members of objects created from the class. The `static` variable is described later in this course in more detail. Both member variables and class variables are initialized automatically when they are created.

Method parameter variables define arguments passed in a method call. Each time the method is called, a new variable is created and it lasts only until the method is exited.

Local variables are created when execution enters the method and are destroyed when the method is exited. This is why local variables are sometimes referred to as *temporary* or *automatic*. Variables that are defined within a member function are local to that member function, so you can use the same variable name in several member functions to refer to different variables. This is illustrated in the example on Code 4-1 on page 4-3.

Code 4-1 Variable Scope Example

```
1   public class ScopeExample {
2     private int i=1;
3
4     public void firstMethod() {
5       int i=4, j=5;
6       this.i = i + j;
7       secondMethod(7);
8     }
9     public void secondMethod(int i) {
10      int j=8;
11      this.i = i + j;
12    }
13  }
```

```
1   public class TestScoping {
2     public static void main(String[] args) {
3       ScopeExample scope = new ScopeExample();
4       scope.firstMethod();
5     }
6   }
```

Note – This code serves to exemplify scoping rules *only*. Reusing names in this manner is not a good practice.

Figure 4-1 shows a visualization of the variables in Code 4-1.

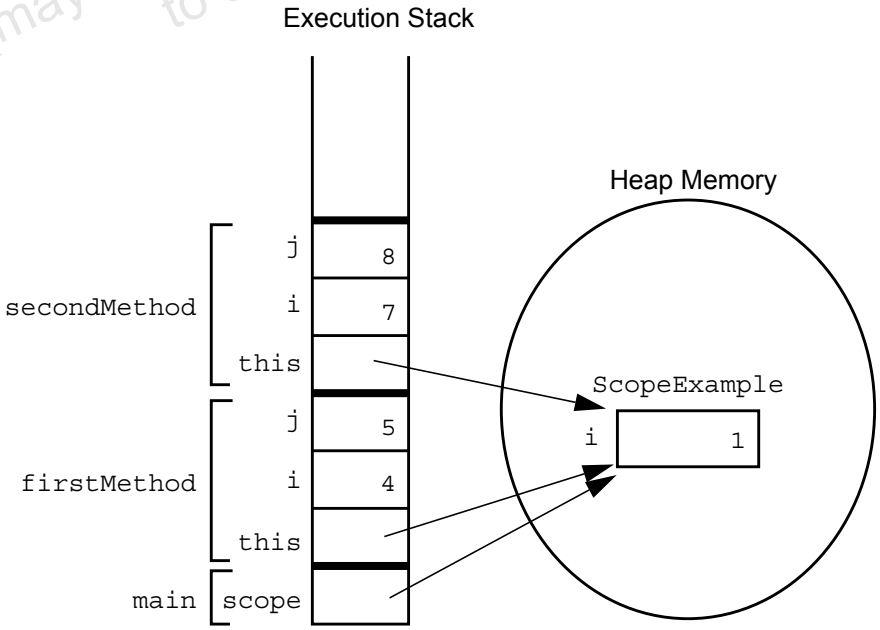

Figure 4-1 Diagram of Variable Scoping for Code 4-1

Variable Initialization

You must initialize a variable in a Java program before you can use it. For stack variables it is easy for the compiler to check if the code assigns a value to such a variable before it attempts to use that value: The `this` reference and the method parameters have assigned values when the method executions starts. Local variables that are defined within the method code can be checked by looking at the method code only. So the compiler enforces that the method code cannot read a variable value before we have assigned one.

For object attributes on the heap, such a check is not easily possible: How can the compiler know whether a client of our class calls a setter method on an object before maybe some other client wants to call a getter method on it? So the only reasonable way to handle this problem is to enforce assigning an initial value already in the constructor. If the programmer does not provide a default value, the compiler will assign the default value. Table 4-1 lists the default values for primitive and reference instance variables.

Table 4-1 Default Values of Primitive Types

Variable	Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0D`
`char`	`'\u0000'`
`boolean`	`false`
All reference types	`null`

Note – A reference that has the `null` value does not refer to an object. An attempt to use the object it refers to causes an exception. Exceptions are errors that occur at runtime and are described in a later module.

Initialization Before Use Principle

While variables defined outside of a method are initialized automatically, local variables *must* be initialized manually before use. The compiler flags an error if it can determine a condition where a variable can be used before being initialized.

```
public void doComputation() {
  int x = (int)(Math.random() * 100);
  int y;
  int z;
  if (x > 50) {
    y = 9;
  }
  z = y + x;  // Possible use before initialization
}
```

Operators

This section describes the operators in the Java programming language.

Operator Precedence

The Java programming language operators are similar in style and function to those of C and C++. Table 4-2 lists the operators in order of precedence (L to R means left-to-right associative; R to L means right-to-left associative).

Table 4-2 Operators and Precedence

Associative	Operators
R to L	`++ -- + - ~ ! (<data_type>)`
L to R	`* / %`
L to R	`+ -`
L to R	`<< >> >>>`
L to R	`< > <= >= instanceof`
L to R	`== !=`
L to R	`&`
L to R	`^`
L to R	`|`
L to R	`&&`
L to R	`||`
R to L	`<boolean_expr> ? <expr1> : <expr2>`
R to L	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Note – The `instanceof` operator is unique to the Java programming language and is described in Module 6.

Logical Operators

Most Java technology operators are taken from other languages and behave as expected.

Relational and logical operators return a `boolean` result. The value `0` is not interpreted automatically as false and non-zero values are not interpreted automatically as true.

```
int i = 1;
if (i)      // generates a compile error
if (i != 0) // Correct
```

The `boolean` operators supported are `!`, `&`, `^`, and `|` for the algebraic Boolean operations NOT, AND, XOR, and OR, respectively. Each of these returns a `boolean` result. The operators `&&` and `||` are the short-circuit equivalents of the operators `&` and `|`.

Short-Circuit Logical Operators

The operators `&&` (defined as `AND`) and `||` (defined as `OR`) perform *short-circuit* logical expressions. Consider this example:

```
MyDate d = reservation.getDepartureDate();
if ((d != null) && (d.day > 31)) {
  // do something with d
}
```

The `boolean` expression that forms the argument to the `if ()` statement is legal and entirely safe. This is because the second subexpression is skipped when the first subexpression is `false`, because the entire expression is always false when the first subexpression is false, regardless of how the second subexpression evaluates. Similarly, if the `||` operator is used and the first expression returns `true`, the second expression is not evaluated because the whole expression is already known to be true.

Bitwise Logical Operators

Bit manipulation operations, including logical and shift operations, perform low-level operations directly on the binary representations used in integers. These operations are not often used in enterprise-type systems but might be critical in graphical, scientific, or control systems. The ability to operate directly on binary might save large amounts of memory, might enable certain computations to be performed very efficiently, and can greatly simplify operations on collections of bits, such as data read from or written to parallel I/O ports.

The Java programming language supports bitwise operations on integral data types. These are represented as the operators ~, &, ^, and | for the bitwise operations of NOT (bitwise complement), bitwise AND, bitwise XOR, and bitwise OR, respectively.

Figure 4-2 shows examples of the bitwise operators on byte-sized binary numbers.

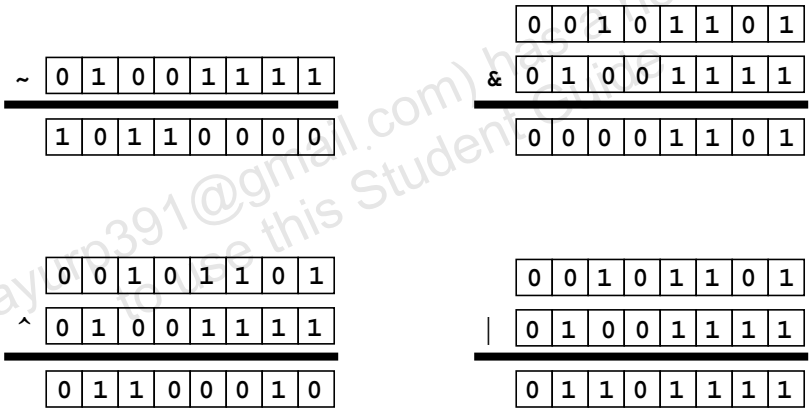

Figure 4-2 Examples of the Bitwise Operators

Right-Shift Operators >> and >>>

The Java programming language provides two right-shift operators. The operator >> performs an *arithmetic* or *signed* right shift. The result of this shift is that the first operand is divided by 2 raised to the number of times specified by the second operand. For example:

```
128 >> 1 returns 128/2^1 = 64
256 >> 4 returns 256/2^4 = 16
-256 >> 4 returns -256/2^4 = -16
```

The >> operator results in the sign bit being copied during the shift.

The *logical* or *unsigned* right shift operator >>> works on the bit pattern rather than the arithmetic meaning of a value and always places 0s in the most significant bits; for example:

```
1010 ... >> 2 gives 111010 ...
1010 ... >>> 2 gives 001010 ...
```

Left-Shift Operator <<

The operator << performs a left shift. The result of this shift is that the first operand is multiplied by two raised to the number specified by the second operand; for example:

```
128 << 1 returns 128*2^1 = 256
16 << 2 returns 16*2^2 = 64
```

All three shift operators reduce their right-hand operand modulo 32 for an `int` type left-hand operand and modulo 64 for a `long` type left-hand operand. Therefore, for any `int x`, `x >>> 32` results in `x` being unchanged, not `0` as you might expect.

The shift operators are permitted only on integral types. The unsigned right shift >>> is effective only on `int` or `long` values. If you use it on a `short` or `byte` value, the value is promoted, with sign extension, to an `int` before >>> is applied. By this point, the unsigned shift usually has become a signed shift.

Shift Operator Examples

Figure 4-3 show the bit patterns of a positive and a negative number and the bit patterns resulting from the three shift operators: >>, >>>, and <<.

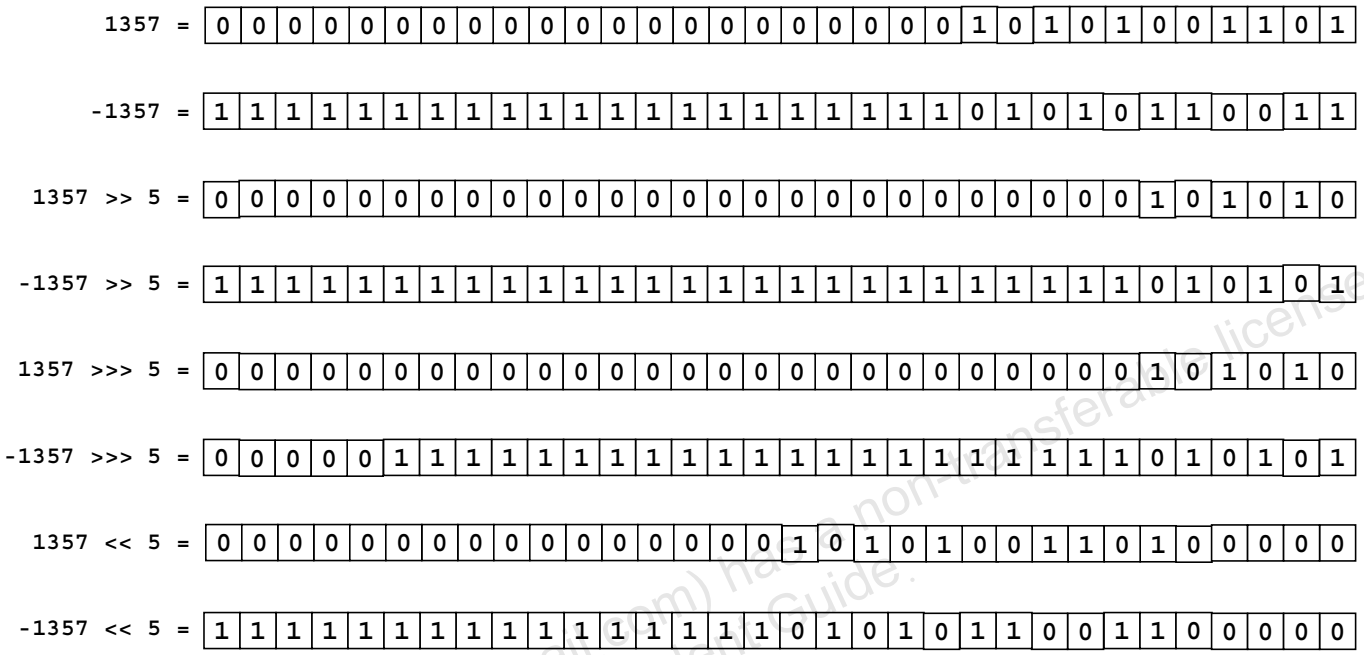

Figure 4-3 Examples of the Shift Operators

Note – The code that generated these examples (including printing out the complete bit pattern) can be found in the file `examples/mod04_stmts/TestShift.java`.

String Concatenation With +

The + operator performs a concatenation of `String` objects, producing a new `String`.

```
String salutation = "Dr. ";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

The result of the last line is:

```
Dr. Pete Seymour
```

If either argument of the + operator is a `String` object, then the other argument is converted to a `String` object. All objects can be converted to a `String` object automatically, although the result might be rather cryptic. The object that is not a `String` object is converted to a string equivalent using the `toString()` member function.

Casting

Casting means assigning a value of one type to a variable of another type. If the two types are compatible, the Java software performs the conversion automatically. For example, an `int` value can always be assigned to a `long` variable.

Where information might be lost in an assignment, the compiler requires that you confirm the assignment with a cast. For example, *squeeze* a `long` value into an `int` variable like this:

```
long bigValue = 99L;
int squashed = bigValue;       // Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L;            // Wrong, needs a cast
int squashed = (int) 99L;      // OK, but...
int squashed = 99;             // default integer literal
```

The desired target type is placed in parentheses and used as a prefix to the expression that must be modified. Although it might not be necessary, it is advisable to enclose the entire expression to be cast in parentheses. Otherwise, the precedence of the cast operation can cause problems.

Note – Reference type variables can also be cast; see Module 6.

Promotion and Casting of Expressions

Variables can be promoted automatically to a longer form (such as `int` to `long`) when there would be no loss of information.

```
long bigval = 6;      // 6 is an int type, OK
int smallval = 99L;   // 99L is a long, illegal

double z = 12.414F;   // 12.414F is float, OK
float z1 = 12.414;    // 12.414 is double, illegal
```

In general, you can think of an expression as being *assignment-compatible* if the variable type is at least as large (maximum value) as the expression type.

For binary operators, such as the + operator, when the two operands are of primitive numeric types, the result type is determined as the largest type of either operand, or `int`. Therefore, all binary operations on numeric types result in at least an `int` result, and possibly a larger one if `long`, `float`, or `double` operands are in the expression. This might result in overflow or loss of precision when the result is assigned to a variable.

For example, the following code fragment:

```
short a, b, c;
a = 1;
b = 2;
c = a + b;
```

causes an error because it raises each `short` to an `int` before operating on it. However, if `c` is declared as an `int`, or a cast is done as:

```
c = (short)(a + b);
```

then the code works.

Branching Statements

Conditional statements enable the selective execution of portions of the program according to the value of some expressions. The Java programming language supports the `if` and `switch` statements for two-way and multiple-way branching, respectively.

Simple `if`, `else` Statements

The basic syntax for an `if` statement is:

```
if ( <boolean_expression> )
  <statement_or_block>
```

For example:

```
if ( x < 10 )
  System.out.println("Are you finished yet?");
```

However, it is recommended that you place all then statements into a block. For example:

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
}
```

Complex `if`, `else` Statements

If you require an `else` clause, then you must use the `if-else` statement:

```
if ( <boolean_expression> )
  <statement_or_block>
else
  <statement_or_block>
```

For example:

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
} else {
  System.out.println("Keep working...");
}
```

If you require a series of conditional checks, then you can chain a sequence of `if-else-if` statements:

```
if ( <boolean_expression> )
  <statement_or_block>
else if ( <boolean_expression> )
  <statement_or_block>
```

For example:

```
int count = getCount(); // a method defined in the class
if (count < 0) {
  System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
  System.out.println("Error: count value is too big.");
} else {
  System.out.println("There will be " + count +
                     " people for lunch today.");
}
```

The Java programming language differs from C and C++ because an `if()` statement takes a `boolean` expression, not a numeric value. You cannot convert or cast `boolean` types and numeric types. If you have:

```
if (x) // x is int
```

use:

```
if (x != 0)
```

The entire `else` part of the statement is optional, and you can omit it if no action is to be taken when the tested condition is false.

The switch Statement

The switch statement syntax is:

```
switch ( <expression> ) {
  case <constant1>:
    <statement_or_block>*
    [break;]
  case <constant2>:
    <statement_or_block>*
    [break;]
  default:
    <statement_or_block>*
    [break;]
}
```

In the switch (<expression>) statement, <expression> must be expression-compatible with an int type. Promotion occurs with byte, short, or char types. Floating point, long expressions, or object references (including Strings) are not permitted.

Note – An enumerated type value may also be used in the <expression> and <constantN> elements. See Enumerated Types in Module 7.

The optional default label specifies the code segment to be executed when the value of the variable or expression cannot match any of the case values. If there is no break statement as the last statement in the code segment for a certain case, the execution continues into the code segment for the next case without checking the case expression's value.

A return statement can be used instead of a break statement. Moreover, if the switch is in a loop, then a continue statement would also cause execution to exit out of the switch construct.

Note – Nine out of ten switch statements required breaks in each case block. Forgetting the break statement causes the most programming errors when using switch statements.

An example switch statement is shown in Code 4-2 on page 4-17.

Code 4-2 The `switch` Statement Example 1

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
    addRadio();
    addWheels();
    addEngine();
    break;
  case STANDARD:
    addRadio();
    addWheels();
    addEngine();
    break;
  default:
    addWheels();
    addEngine();
}
```

Code 4-2 configures a car object based on the `carModel`. If `carModel` is the integer constant `DELUXE`, then air conditioning is added to the car, as is a radio, and, of course, wheels and an engine. However, if the `carModel` is only a `STANDARD`, then only a radio, wheels, and an engine are added. Finally, by default, any car model will have wheels and an engine added.

A second example `switch` statement is shown in Code 4-3.

Code 4-3 The `switch` Statement Example 2

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
  case STANDARD:
    addRadio();
  default:
    addWheels();
    addEngine();
}
```

Code 4-3 solves the redundant method calls in the previous example by permitting the flow of control to descend through multiple case blocks. For example, if the `carModel` is `DELUXE`, then the `addAirConditioning` method is called, and then the flow of control falls through the next case statement and calls the `addRadio` method, and finally the flow of control falls through the `default` statement and calls the `addWheels` and `addEngine` methods.

Looping Statements

Looping statements enable you to execute blocks of statements repeatedly. The Java programming language supports three types of loop constructs: `for`, `while`, and `do` loops. The `for` and `while` loops test the loop condition before executing the loop body; the `do` loops check the loop condition after executing the loop body. This implies that the `for` and `while` loops might not execute the loop body even once, whereas `do` loops execute the loop body at least once.

The `for` Loops

The `for` loop syntax is:

```
for ( <init_expr>; <test_expr>; <alter_expr> )
  <statement_or_block>
```

For example:

```
for ( int i = 0; i < 10; i++ )
  System.out.println(i + " squared is " + (i*i));
```

However, it is recommended that you place all loop-clause statements into a block. For example:

```
for ( int i = 0; i < 10; i++ ) {
  System.out.println(i + " squared is " + (i*i));
}
```

In the previous example, `int i` is declared and defined within the `for` block. The variable `i` is accessible only within the scope of this particular `for` block.

Note – The Java programming language permits the comma separator in a `for()` loop structure. For example,
`for (i = 0, j = 0; j < 10; i++, j++) { }` is legal, and it initializes both `i` and `j` to `0`, and increments both `i` and `j` after executing the loop body.

The `while` Loop

The `while` loop syntax is:

```
while ( <test_expr> )
  <statement_or_block>
```

For example:

```
int i = 0;
while ( i < 10 ) {
  System.out.println(i + " squared is " + (i*i));
  i++;
}
```

Ensure that the loop-control variable is initialized appropriately before the loop body begins execution. You must update the control variable appropriately to prevent an infinite loop.

The `do/while` Loop

The syntax for the `do/while` loop is:

```
do
  <statement_or_block>
while ( <test_expr> );
```

For example:

```
int i = 0;
do {
  System.out.println(i + " squared is " + (i*i));
  i++;
} while ( i < 10 );
```

As with the `while` loops, ensure that the loop-control variable is initialized appropriately, updated in the body of the loop, and tested properly.

Use the `for` loop in cases where the loop is to be executed a predetermined number of times. Use the `while` and `do` loops in cases where this is not determined beforehand.

Special Loop Flow Control

You can use the following statements to further control loop statements:

- `break [<label>];`

Use the `break` statement to prematurely exit from `switch` statements, loop statements, and labeled blocks.

- `continue [<label>];`

  Use the `continue` statement to skip over and jump to the end of the loop body, and then return control to the loop-control statement.

- *`<label>`* `:` *`<statement>`*

  The `label` statement identifies any valid statement to which control must be transferred. With regard to a labeled `break` statement, the label can identify any statement. With regard to a labeled `continue` statement, the label must identify a loop construct.

The `break` Statement

Here is an example loop with an unlabeled `break` statement:

```
1   do {
2     statement;
3     if ( condition ) {
4       break;
5     }
6     statement;
7   } while ( test_expr );
```

The `continue` Statement

Here is an example loop with an unlabeled `continue` statement:

```
1   do {
2     statement;
3     if ( condition ) {
4       continue;
5     }
6     statement;
7   } while ( test_expr );
```

Using `break` Statements with Labels

Here is an example loop with a labelled `break` statement:

```
1   outer:
2     do {
3       statement1;
4       do {
5         statement2;
6         if ( condition ) {
7           break outer;
8         }
9         statement3;
10      } while ( test_expr );
11      statement4;
12    } while ( test_expr );
```

Using `continue` Statements with Labels

Here is an example loop with a labelled `continue` statement:

```
1   test:
2     do {
3       statement1;
4       do {
5         statement2;
6         if ( condition ) {
7           continue test;
8         }
9         statement3;
10      } while ( test_expr );
11      statement4;
12    } while ( test_expr );
```