Operators and Flow Control

olsen
software ltd

Contents

1. Operators
2. Conditional statements
3. Loops

Demo project: **DemoOpFlow**

1. Operators

- Arithmetic operators
- Conditional operator
- Assignment operators
- Aside: working with strings
- Casting
- Aside: working with bytes
- Relational operators
- Logical operators
- Bitwise operators
- Operator precedence

Arithmetic Operators

- Basic binary operators
  - `a + b` (addition)
  - `a - b` (subtraction)
  - `a * b` (multiplication)
  - `a / b` (division)
  - `a % b` (modulo, i.e. remainder)
- Basic unary operators
  - `+a` (unary plus)
  - `-a` (unary negation)
  - `a++` (postfix increment by 1)
  - `++a` (prefix increment by 1)
  - `a--` (postfix decrement by 1)
  - `--a` (prefix increment by 1)

Conditional Operator

- The conditional operator is like an in-situ if test
  - (*condition*) ? *trueResult* : *falseResult*
- Example:

```
boolean isMale;
int age;
…
int togo = (isMale) ? (65 – age) : (60 – age);

System.out.printf("%d years to retirement\n", togo);
```

Assignment Operators (1 of 2)

- Basic assignment operator
  - a = b (assign b to a)
  - Performs widening conversion implicitly, if needed (see later)

- Primitive assignment
  - Assign LHS variable a bitwise copy of the RHS value

```
int a = 100;
int b = 200;

b = 42;
a = b;
```

- Reference assignment:
  - Assign LHS variable a reference to the RHS object

```
File file1 = new File("somefile.txt");
File file2 = new File("anotherfile.txt");

file1 = file2;
```

Assignment Operators (2 of 2)

- Compound assignment operators:
  - a += b (calculate a + b, then assign to a)
  - a –= b (calculate a – b, then assign to a)
  - a *= b (calculate a * b, then assign to a)
  - a /= b (calculate a / b, then assign to a)
  - etc.

- Example:

```
// Use *= compound operator:
x *= a + b;

// Equivalent to the following (note the precedence):
x = x * (a + b);
```

Aside: Working with Strings

- String concatenation
  - `strResult = str1 + str2`
  - `strResult = str1 + obj`
- String shortcut concatenation
  - `str1 += str2`
  - `str1 += obj`
- Examples
  - What do the following statements do?

```
String message = "Hello";
int a = 5;
int b = 6;

System.out.println(message + a + b);
System.out.println(message + (a + b));
System.out.println("" + a + b);
System.out.println(a + b);
```

Casting (1 of 2)

- Implicit conversions:
  - Java implicitly converts less-precise expns to more-precise expns
  - byte -> short -> int -> long -> float -> double
- Explicit conversions (aka casting):
  - You can explicitly cast an expression into a compatible other type
  - (*type*) *expression*
  - Might result in a loss of precision
- Explain the following example:

```
int judge1Score;
int judge2Score;
int judge3Score;
…
double averageScore = (double) (judge1Score + judge2Score + judge3Score) / 3;
```

- Questions:
  - What would happen without the above cast?
  - Can you achieve the same effect without using explicit casting?

Aside: Working with Bytes

- You can assign an integer literal value to a `byte`
  - Compiler implicitly casts integer literal value into a `byte`

```
byte age = 21;

// Equivalent to:
// byte age = (byte) 21;
```

- If you do any arithmetic with bytes, the result is an int
  - Consider the following example:

```
byte myAge = 21;
byte yourAge = 22;

int  totalAge1 = myAge + yourAge;            // This works.

byte totalAge2 = myAge + yourAge;            // This doesn't work.
byte totalAge3 = (byte)(myAge + yourAge);    // This works.
```

Relational Operators (1 of 2)

- There are 6 relational operators (all return `boolean`):
  - `==` (equality)
  - `!=` (inequality)
  - `>` (greater-than)
  - `>=` (greater-than-or-equal)
  - `<` (less-than)
  - `<=` (less-than-or-equal)

Relational Operators (2 of 2)

- If you use == or != on primitive types:
  - You are comparing numeric values
  - i.e. do they contain the same value
- If you use == or != on reference types:
  - You are comparing object references
  - i.e. do they point to the same object
- To compare the *values* of objects:
  - Use the `equals()` method, e.g. `str1.equals(str2)`
  - The `String` class also has `equalsIgnoreCase()`

Logical Operators

- Short-circuit logical operators:
  - && (logical AND)
  - || (logical OR)
- Non-short-circuit logical operators:
  - & (logical AND)
  - | (logical OR)
  - ^ (logical XOR)
- Logical inverse operator:
  - ! (logical NOT)
- Note:
  - All these operators require boolean operands

Bitwise Operators

- Bitwise AND and OR binary operators
  - & (bitwise AND)
  - ^ (bitwise exclusive OR)
  - | (bitwise inclusive OR)
- Bitwise NOT unary operator
  - ~ (bitwise NOT)
- Bitwise shift operators
  - << (shift bits left)
  - >> (signed right-shift, i.e. shift bits right and preserve top bit)
  - >>> (unsigned right-shift, i.e. shift bits right and set top bit to 0)

Order of Precedence

- Operators have the following precedence (use parens to override this precedence):
  - Unary: `++ -- + - ! ~ (`*type*`)`
  - Multiplicative: `* / %`
  - Additive: `+ -`
  - Bitwise shift: `<< >> >>>`
  - Relational: `> >= < <= instanceof`
  - Equality: `== !=`
  - Bitwise AND: `&`
  - Bitwise exclusive OR: `^`
  - Bitwise inclusive OR: `|`
  - Logical AND: `&& &`
  - Logical OR: `|| |`
  - Ternary: `?:`
  - Assignment: `= += -=` etc.

2. Conditional Statements

- Using if tests
- Quiz
- Nesting if tests
- Using switch tests
- Using strings in switch tests

Using if Tests

- Basic if tests

```
if (booleanTest) {
  body
}
```

Executes *body* if *booleanTest* is `true`

- if-else tests

```
if (booleanTest) {
  body1
} else {
  body2
}
```

Executes *body1* if *booleanTest* is `true`

Otherwise, executes *body2*

- if-else-if tests

```
if (booleanTest1) {
  body1
}
else if (booleanTest2) {
  body2
}
else if (test3) {
  body3
}
…
else {
  lastBody
}
```

Executes *body1* if *booleanTest1* is `true`

Or executes *body2* if *booleanTest2* is `true`

Or executes *body3* if *booleanTest3* is `true`

If all else fails, executes (optional) *lastBody*

- Notes:
  - Test conditions must be `boolean` (or `Boolean`)
  - {} are optional if you want a 1-line statement

Quiz

- Explain the following examples
  - … and spot the deliberate gotchas ☺

```
int i = … ;
int j = … ;

if (i == j) {
 System.out.println("Equal.");
}

if (i == j)
  System.out.println("Equal.");

if (i == j)
  System.out.println("Equal.");
  System.out.println("Goodbye.");

if (i == j);
  System.out.println("Equal.");

if (i = j)
  System.out.println("Equal.");

if (i)
 System.out.println("i is non-zero.");
```

```
boolean b = … ;

if (b == true) …

if (b == methodThatReturnsBoolean()) …

if (b = methodThatReturnsBoolean()) …
```

```
boolean b1 = … ;
boolean b2 = … ;

if (b1)
if (b2)
System.out.println("Yes");
else System.out.println("No");
```

Nesting if Tests

- You can nest if tests inside each other
  - Use {} to ensure correct logic, as needed
  - Use indentation for readability

```
int age = … ;
String gender = … ;

if (age < 18) {

  if (gender.equals("Male")) {
    System.out.println("boy");
  } else {
    System.out.println("girl");
  }

} else {

   if (age >= 100) {
      System.out.print("centurion ");
   }

  if (gender.equals("Male")) {
    System.out.println("man");
  } else {
    System.out.println("woman");
  }
}
```

Using switch Tests

- The `switch` statement is useful if you want to test a single expression against a finite set of expected values
- General syntax:

```
switch (expression) {

case constant1:
  branch1Statements;
  break;

case constant2:
  branch2Statements;
  break;

…

default:
  defaultBranchStatements;
  break;
}
```

- Expression must be:
  - char, byte, short, int (or wrappers)
  - enum (Java 6 onwards)
  - string (Java 7 onwards)
- Cases:
  - Must be (different) constants
  - Are evaluated in order, top-down
- If you omit `break`:
  - Fall-through occurs
- The `default` branch:
  - Is optional
  - Doesn't have to be at the end!

Using Strings in Switch Statements

- Java SE 7 lets you use string literals in `switch` statements
  - Equivalent to calling `equals()`
  - i.e. case-sensitive

```
String country;
int diallingCode;
…

switch (country.toLowerCase()) {

  case "uk":
    diallingCode = 44;
    break;

  case "usa":
  case "canada":
    diallingCode = 1;
    break;
  …
}
```

DemoStringsInSwitch.java

3. Loops

- Using while loops
- Using do-while loops
- Using for loops
- Unconditional jumps

Using while Loops

- The `while` loop is the most straightforward loop construct
  - Boolean test is evaluated
  - If `true`, loop body is executed
  - Boolean test is re-evaluated
  - Etc…

```
while (booleanTest) {
   loopBody
}
```

- Note:
  - Loop body will not be executed if test is `false` initially

- How would you write a `while` loop…
  - To display 1 – 5?
  - To display the first 5 odd numbers?
  - To read 5 strings from the console, and output in uppercase?

Using do-while Loops

- The `do-while` loop has its test at the end of the loop
  - Loop body is always evaluated at least once
  - Handy for input validation
  - Note the trailing semicolon!

```
do {
   loopBody
} while (booleanTest);
```

- How would you write a `do-while` loop…
  - To keep reading strings from the console, until the user enters "Oslo", "Bergen", or "Trondheim" (in any case)?

Using for Loops

- The `for` loop is the most explicit loop construct
  - Initialization part can declare/initialize variable(s)
  - Test part can incorporate any number of tests
  - Update part can do anything, e.g. update loop variable(s)
  - You can omit any (or all!) parts of the for-loop syntax
  - Note: Also see for-each later

```
for (init; booleanTest; update) {
  loopBody
}
```

Note:
If you declare variables in the initialization section (or in the loop body, of course), they are scoped to the for-loop

- How would you write a `for` loop…
  - To display the first 5 odd numbers?
  - To display 100 – 50, in downward steps of 10?
  - To loop indefinitely?

Unconditional Jumps (1 of 2)

- Sometimes it can be convenient to use unconditional jump statements within a loop
  - `break`
    - Terminates innermost loop
  - `continue`
    - Terminates current iteration of innermost loop, and starts next iteration
    - If used in a `for` loop, transfers control to the update part
  - `return`
    - Terminates entire method
- Discuss:

```
for (initialization; test; update) {
  …
  if (someCondition)
    break;
  …
  if (someOtherCondition)
    continue;
  …
}
```

Unconditional Jumps (2 of 2)

- You can use `break` and `continue` in nested loops
  - By default, they relate to the inner loop
  - To relate to the outer loop, use labels
- Discuss:

```
myOuterLabel:

// Outer Loop.
for (init; booleanTest; update) {
  …

  // Inner Loop.
  for (init; booleanTest; update) {
    if (someCondition)
      break myOuterLabel;
    …
    if (someOtherCondition)
      continue myOuterLabel;
    …
  }
}
```

Any Questions?