Decisions and Loops

Java Developer

StayAhead Training; January, 2023

Expressions and Operators*

Decisions and Loops

- An expression is a statement that produces a new value
- An operator is a symbol that dictates the action to be performed against one or more operands (variable or literal)

```
// x + y * z is the expression
// x, y, and z are operands
// + and * are operators
var result = x + y * z;
```

- Arithmetic operators (like every other language): `+ - * / %` (remainder)

Relational Operators

Decisions and Loops

- A relational operator is one that compares operands and produces a boolean
- == (equal to) != (not equal to) > >= < <=
- Relational operators should, for the most part, only be used with primitive variables/literals
- That is, they should not/cannot be used to compare references
- Java does **not** support operator overloading
- Relational operators are most commonly used to control decisions and loops

Logical Operators

Decisions and Loops

- A logical operator is one that combines expressions (not, not so much)
- `&&` (and) `||` (or) `!` (not)
- `expr1 && expr2`: the result is true if **both** expressions are true
- `expr1 || expr2`: the result is true if **either** expression is true
- `!expr1`: the result is the inverted

if Statements

Decisions and Loops

- An if statement provides for the execution of **one** of many blocks of code
- An if statement can comprise one or more blocks
- Each block is associated with a boolean expression
- The block that is executed is the first one whose associated boolean expression evaluates to true
- Once a block is executed the remaining blocks are skipped and control moves beyond the if statement

if Statements

Decisions and Loops

- An if statement with a **single** block takes the following form:

```
if (<boolean_expression>) {
  // instruction(s) to execute if the expression is true
}
```

- NB: if there is only one instruction to be executed then the braces *may* be omitted, though we don't recommend doing so

if Statements

Decisions and Loops

- An if statement with **two** blocks takes the following form:

```
if (<boolean_expression>) {
  // instruction(s) to execute if the expression is true
} else {
  // instruction(s) to execute if the expression is false
}
```

- NB: the else keyword may be set on a new line rather than its being adjacent to the closing brace of the first block; likewise the opening brace of each block may be set on a new line

if Statements

Decisions and Loops

- An if statement with **three** blocks takes the following form:

```
if (<boolean_expression_1>) {
  // instruction(s) to execute if expression 1 is true
} else if (<boolean_expression_2>) {
  // instruction(s) to execute if expression 2 is true
} else {
  // instructions to execute if both expressions are false
}
```

- if statements with more than three blocks require additional else if clauses

if Statements

Decisions and Loops

- Abutting if statements may appear to be related when they're not

```
// there are two if statements here
// the second one should be set on a new line
if (x == 1) {
  // instruction(s) to execute if x == 1
} if (x > 0) {
  // instruction(s) to execute if x > 0
} else {
  // instructions to execute if x <= 0
}
```

Whitespace*

Decisions and Loops

- Whitespace aids readability but is ignored by the compiler & interpreter
- This is valid…

```
if(x==1){y=1;}else{y=0;}
```

- So is this…

```
if (x == 1) y = 1;
  else y = 0;
```

- Take care to indent your code properly

switch Statements

Decisions and Loops

- A switch statement provides for the execution of **multiple** blocks of many
- A switch statement can comprise one or more blocks
- Each block is associated with an equality test
- Only primitive variables, literals, Strings, and enums can be used
- The blocks that are executed are all those that follow the first equality test that evaluates to true
- The default block alone is executed when none of the tests evaluate to true

switch Statements

Decisions and Loops

- A switch statement takes the following form:

```
switch (<operand1>) {
  case <operand2>:
    // instruction(s) to execute if operand1 == operand2
  case <operand3>:
    // instruction(s) to execute if operand1 == operand2/3
  default:
    // instruction(s) to execute if operand1 == operand2/3
    // ...or if operand1 != operand2/3
}
```

- switch statements with more than three blocks require additional case clauses

switch Statements

Decisions and Loops

- A switch statement example:

```
switch (mealDealOption) {
  case 3:
    System.out.println("Drink");
  case 2:
    System.out.println("Fries");
  case 1:
    System.out.println("Burger");
}
```

switch Statements

Decisions and Loops

- Switch statements are often forced to behave like if statements by using break:

```
switch (<operand1>) {
  case <operand2>:
    // instruction(s) to execute if operand1 == operand2
    break;
  case <operand3>:
    // instruction(s) to execute if operand1 == operand3
    break;
  default:
    // instruction(s) to execute if operand1 != operand2/3
}
```

switch Expressions (since Java 12)

Decisions and Loops

- A switch expression returns from each case so that break is redundant, e.g.

```
switch (<operand1>) {
  case <operand2> ->
    // instruction(s) to execute if operand1 == operand2
  case <operand3> ->
    // instruction(s) to execute if operand1 == operand3
  default ->
    // instruction(s) to execute if operand1 != operand2/3
}
```

- Note the arrow (->) operator in place of the colon (:)

switch Expressions (since Java 12)

Decisions and Loops

- A switch expression may produce a result (each case returns), e.g.

```
var result = switch (x) {
  case 1 -> "x is 1";
  case 2 -> "x is 2";
  default -> "x is neither 1 nor 2";
}
```

switch Expressions (since Java 12)

Decisions and Loops

- Where each case comprises more than one statement, the yield keyword may be used to specify the return value, e.g.

```
var result = switch (x) {
  case 1 ->
    System.out.println("Processing case 1...");
    yield "x is 1";
  case 2 ->
    System.out.println("Processing case 2...");
    yield "x is 2";
  ...
}
```

switch Expressions (since Java 12)

Decisions and Loops

- The yield keyword may even be used in a traditional switch statement where each case produces a value, e.g.

```
var result = switch (x) {
  case 1: yield "x is 1";
  case 2: yield "x is 2";
  default: yield "x is neither 1 nor 2";
}
```

The Ternary Operator

Decisions and Loops

- The ternary operator provides for the assigning of one of two values
- It take the following form:

```
var x = (<expr>) ? <value_if_true> : <value_if_false>;
```

- The expression must produce true or false (it must be a boolean expression)

while Loops

Decisions and Loops

- A while loop provides for the execution of a block of code **repeatedly**
- The block is associated with a boolean expression
- The block will be executed repeatedly while the expression evaluates to true
- The expression may be evaluated before the block (while) or after (do while)
- A while loop will execute the block zero or more times
- A do while loop will execute the bock one or more times

while Loops

Decisions and Loops

- A while loop takes the following form:

```
while (<boolean_expression>) {
  // instruction(s) to execute while the expr. is true
}
```

- The instruction(s) may **never** be executed

while Loops

Decisions and Loops

- A do while loop takes the following form:

```
do {
  // instruction(s) to execute while the expr. is true
} while (<boolean_expression>);
```

- The instruction(s) will be executed at least once

for Loops

Decisions and Loops

- A for loop *also* provides for the execution of a block of code **repeatedly**
- The block is associated with a boolean expression
- The block will be executed repeatedly while the expression evaluates to true
- The expression is always evaluated before the block
- for loops come in two varieties: traditional and enhanced
- for loops are most commonly used it iterate over the elements of a collection

for Loops

Decisions and Loops

- The traditional for loop takes the following form:

```
for (<init_stmt>; <bool_expr>; <post_block_stmt>) {
  // instruction(s) to execute while the expr. is true
}
```

- The initial statement is executed only once before the block is executed
- The boolean expression is evaluated before each iteration
- The post block statement is executed after each iteration

for Loops

Decisions and Loops

- A traditional for loop example:

```
for (var x = 1; x <= 5; x = x + 1) {
  System.out.println(x);
}
```

- Note that `x = x + 1` might be refactored as `x += 1` or `x++`

for Loops

Decisions and Loops

- The enhanced for loop takes the following form:

```
for (var <element_name> : <collection>) {
  // instruction(s) to exe. while there are more elements
}
```

- <element_name> is a variable name of your choosing; it will reference each subsequent element in the collection in each iteration
- <collection> is a variable that references an array/other type of collection

for Loops

Decisions and Loops

- An enhanced for loop example:

```
var names = new String[] {"Tom", "Dick", "Harry"};
for (var name : names) {
  System.out.println(name);
}
```

- The block will be executed three times (the array size is three)
- During the first iteration the variable, name, will reference the first element of the array - "Tom", during the second iteration name will reference "Dick" etc.

Branching Statements

Decisions and Loops

- break forces the early termination of a loop/control to move beyond the loop
- continue forces control to move immediately to the next iteration, ignoring all remaining instructions in the block
- return may be used with or without a return value inside a decision or loop statement - it forces the termination of the method and control to pass back to the caller

Nesting*

Decisions and Loops

- Nesting is the existence of one thing (decision/loop statement) inside another
- Nesting is very common, e.g.

```
for (var name : names) {
  if (name.startsWith("S")) {
    System.out.println(name);
  }
}
```

- Take care to indent your code properly

Infinite Loops*

Decisions and Loops

- Infinite loops can make for more readable code when combined with break

```
int number = 0;
while (true) {
  System.out.print("Enter a number between 1 and 10: ");
  number = scanner.nextInt();
  if (number >= 1 && number <= 10) {
    break;
  } else {
    System.out.println("Try again!");
}
```