Control Flow

Expressions Statements And Blocks

Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

```
int cadence = 0;
```

The expression `cadence = 0` returns an `int`

Statements

- Statements are roughly equivalent to sentences in natural languages.
- A *statement* forms a complete unit of execution.
- The following types of expressions can be made into a statement by terminating the expression with a semicolon (`;`).
- Ex :

```
// assignment statement
aValue = 8933.234;

// increment statement
aValue++;

// method invocation statement
System.out.println("Hello World!");

// object creation statement
Bicycle myBike = new Bicycle();
```

Blocks

A *block* is a group of zero or more statements between balanced braces.

Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building ___, which compute values.
2. Expressions are the core components of ___.
3. Statements may be grouped into ___.
4. The following code snippet is an example of a ___ expression.
   ```
   1 * 2 * 3
   ```
5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ___.
6. A block is a group of zero or more statements between balanced ___ and can be used anywhere a single statement is allowed.

Exercises

Identify the following kinds of expression statements:

- `aValue = 8933.234;`
- `aValue++;`
- `System.out.println("Hello World!");`
- `Bicycle myBike = new Bicycle();`

Control Flow Statements

The `if-then` Statement

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`.

```
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

The `if-then-else` Statement

The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`.

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The switch Statement

- The `switch` statement can have a number of possible execution paths.
- A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in Enum Types), the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer`.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

Break

Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, statements in `switch` blocks *fall through*: All statements after the matching `case` label are executed in sequence.

The while and do-while Statements

The `while` statement continually executes a block of statements while a particular condition is `true`. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The Java programming language also provides a `do-while` statement, which can be expressed as follows: Expression is checked after executing the do block.

```
do {
    statement(s)
} while (expression);
```

The for Statement

The `for` statement provides a compact way to iterate over a range of values.

The general form of the `for` statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the `for` statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to `false`, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

Enhanced For

The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers =
             {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
     }
```

Branching Statement

The break statement;

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop

```
class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts =
            { 32, 87, 3, 589,
              12, 1076, 2000,
              8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

The continue statement;

The `continue` statement skips the current iteration of a `for`, `while` , or `do-while` loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the `boolean` expression that controls the loop.

The `return` Statement;

- The last of the branching statements is the `return` statement.
- The `return` statement exits from the current method, and control flow returns to where the method was invoked.
- The `return` statement has two forms: one that returns a value, and one that doesn't.

Questions and Exercises: Control Flow Statements

Questions

1. The most basic control flow statement supported by the Java programming language is the ___ statement.
2. The ___ statement allows for any number of possible execution paths.
3. The ___ statement is similar to the `while` statement, but evaluates its expression at the ___ of the loop.
4. How do you write an infinite loop using the `for` statement?
5. How do you write an infinite loop using the `while` statement?

Exercises

1. What output do you think the code will produce if `aNumber` is 3?
   Consider the following code snippet.

```
if (aNumber >= 0)
    if (aNumber == 0)
       System.out.println("first string");
    else System.out.println("second string");
       System.out.println("third string");
```

2. Write a test program containing the previous code snippet; make `aNumber` 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is; in other words, what is the control flow for the code snippet?
3. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
4. Use braces, { and }, to further clarify the code.