Conditionals and Looping

Decision making with if statements. Iteration with the while and for loops

Alwin Tareen

Conditional Statements

The if statement

The if statement in Java consists of 2 distinct parts: a condition and a body.

```
if (condition)
{
    body
}
```

- ► The (condition) must be enclosed by parentheses, unlike other programming languages such as Python.
- Omitting the parentheses is a common error.

Conditional Statements

The if statement

- ► The **condition** is a boolean expression that evaluates to either true or false.
- ► The **body** is the block of code that will be executed if the condition is true.
- Note that if the condition evaluates as false, then this code block will be completely skipped over.

```
if (age >= 18)
{
    System.out.println("You can drive.");
}
```

The if-else statement

- Java's if-else statement is used when you want to do one thing if a condition is true, and another thing if a condition is false.
- An if-else statement will execute either the if section or the else section, but never both.

```
if (condition)
{
    body1 // evaluated when condition is true
}
else
{
    body2 // evaluated when condition is false
}
```

An if-else Example

- ▶ If the condition is true, then the assignment statement is executed.
- ▶ If the condition is false, then the println statement is executed. Only one of the statements can be executed.

```
if (amount <= balance)
{
    balance = balance - amount;
}
else
{
    System.out.println("Insufficient balance.");
}</pre>
```

The extended if statement

Java's extended if statement can be used if we have a series of if-else statements, and only one of them can evaluate as true.

```
if (condition)
{ ... }
else (other condition)
{ ... }
else
{ ... }
```

- ► Generally, it has an else condition at the very end. This becomes the default choice for the entire structure.
- ▶ If all of the other conditions evaluate as false, then this default else condition has its code block executed.



An extended if Example

```
if (temp > 100)
   System.out.println("Stifling heat!");
else if (temp > 50)
   System.out.println("Warm environment.");
else
   System.out.println("Freezing cold!");
```

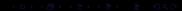
Relational Operators

- A relational operator tests the relationship between two values.
- Java has six relational operators:

Java Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

These operators are mathematical in nature:

```
boolean result = 8 > 5; // result is true
```



The Equality Operator: ==

- ► The equality operator(==) is very confusing to most Java beginners, because it is easy to mix up with the assignment operator(=).
- ► The == operator denotes equality testing.

```
if (age == 65)
{
    System.out.println("You can retire.");
}
```

- ► In the above example, the age variable is compared to 65, to see if they are equal.
- Note: you can only use == to test primitive data types, not objects such as Strings.

Logical Operators

▶ There are three logical operators in Java.

Op.	Description	Example	Result	
!	not	!a	true if a is false, and	
			false if a is true.	
&&	and	a && b	true if a and b are both	
			true, and false otherwise.	
	or	a b	true if either a or b are	
			true, and false otherwise.	

▶ Order of precedence: not, and, or.

Truth Tables

- A logical operation can be described by a truth table that lists all of the possible combinations of values for the input variables involved in an expression.
- ► The following is a two-valued truth table. It shows the outputs for the && and || operators.

		logical and	logical or
a	b	a && b	a b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

The not Operator: !

- ► The not operator gives the logical complement of a boolean value.
- It does not alter the variable upon which it acts.
- ▶ The following is the truth table for the not operator:

	logical not	
a	!a	
false	true	
true	false	

```
if (!lights)
{
    System.out.println("The room is dark.");
}
```

The and Operator: &&

► The result of a logical and(&&) operation is true if both operands are true, but false otherwise.

```
if (chips > 0 && soda > 0)
{
    System.out.println("You have snacks.");
}
```

The or Operator: ||

► The result of a logical or(||) operation is true if one or the other or both of the operands are true, but false otherwise.

```
if (money > 1000 || creditCard == true)
{
    System.out.println("You can buy an iPhone.");
}
```

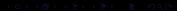
Compound Logical Conditions

- A condition can be formed by using more than one logical operator.
- ▶ This is known as, "chaining together" the operators.

```
if (month == 3 || month == 4 || month == 5)
{
    System.out.println("It is spring.");
}
```

▶ Note that you can't chain together relational operators.

```
boolean result = 5 <= 8 < 12; // error
```



DeMorgan's Laws

Negating a logical expression

- DeMorgan's laws allow us to simplify a boolean expression by distributing the negation operator.
- An interesting outcome is that all or's are converted to and's, and all and's are converted to or's.

For example, the statement, "I don't like chocolate **or** vanilla." is exactly the same as, "I do not like chocolate **and** I do not like vanilla."

Short-Circuit Evaluation

Logical efficiency

- ► The && and || operators are **short-circuited**.
- This means that if the left-hand operand in an boolean expression can decide the entire expression's outcome, then the right-hand side is not evaluated.

Consider the following example:

```
boolean a = false;
boolean result = a && (b || c && (p && q) || m);
```

result will always evaluate as false, regardless of the other boolean values.

Short-Circuit Evaluation

Short-circuit evaluation with and: &&

If the left-hand operand is false, then the result of the entire boolean expression will be false, no matter what the right-hand operand is.

```
boolean a = false;
boolean result = a && (p && q); // a is false
```

Short-circuit evaluation with or: ||

If the left-hand operand is true, then the result of the entire boolean expression will be true, no matter what the right-hand operand is.

```
boolean b = true;
boolean result = b || (p && q); // b is true
```

The while Loop

The indefinite loop

- Recall that a boolean condition is a statement that evaluates to either true or false.
- A while loop repeates looping as long as its boolean condition is true. It is also known as an indefinite loop.
- Java's while loop has the following structure:

```
while (boolean condition)
{
    code block of statements;
}
```

The Counter-controlled Loop

Looping a given number of times

- ► A counter-controlled loop is one that repeats a predetermined number of times.
- The condition in this loop is controlled by a counter variable.
- ► The counter variable keeps track of the number of times that a loop is executed.

```
int count = 0;
while (count < 5)
{
    System.out.println(count);
    count++;
}</pre>
```

The Infinite Loop

Beware the endless loop

- ► A common mistake is when a programmer forgets to increment the counter variable within the body of the while loop.
- If this case occurs, then the boolean condition will always evaluate as true. It will never become false.
- Therefore, the code block of statements within the while loop will execute indefinitely.

```
int count = 0;
while (count < 100)
{
    System.out.println("Hi"); // indefinite loop
}</pre>
```

Summing a Sequence of Integers with while

This Java program uses a while statement to sum the following sequence of integers:

$$1+2+3+4+5+6+7+8+9+10$$

```
int count = 1;
int sum = 0;
while (count <= 10)
{
    sum += count;
    count++;
}
System.out.println(sum);</pre>
```

Incrementing by a Different Amount

- ▶ A counter variable can be incremented by a value other than one.
- ► For example, the following counter is incremented by 10, each time through the loop.

```
int count = 0;
while (count < 100)
{
    System.out.println(count);
    count += 10;
}</pre>
```

The for Loop

The definite loop

- Counter-controlled loops are so frequently used, that programming languages have developed a special structure for them.
- The for statement combines counter initialization, condition testing, and counter updating into a single expression.
- It is also known as a definite loop.

```
for (initialize counter; test counter; update counter)
{
   code block of statements;
}
```

while and for Loop Equivalence

Consider the following while loop:

```
int count = 0;
while (count < 5)
{
    System.out.println("Hello");
    count++;
}</pre>
```

This while loop can be equivalently expressed as the following for loop:

```
for (int count = 0; count < 5; count++)
{
    System.out.println("Hello");
}</pre>
```

Counting Through a Sequence of Integers

- ▶ The following for loop counts from 0 to 9.
- ► The counter variable i is declared as part of the for loop, therefore it only exists in that code block.
- Attempting to use i outside the for loop would result in an error.

```
for (int i = 0; i < 10; i++)
{
    System.out.println(i);
}</pre>
```

Summing a Sequence of Integers with for

This Java program uses a for statement to sum the following sequence of integers:

$$1+2+3+4+5+6+7+8+9+10$$

```
int sum = 0;
for (int i = 1; i <= 10; i++)
{
    sum += i;
}
System.out.println(sum);</pre>
```

Incrementing by a Different Amount

- ▶ The counter variable can be incremented by a value other than one.
- ► For example, the following counter is incremented by 10, each time through the loop.

```
for (int i = 0; i < 100; i += 10)
{
    System.out.println(i);
}</pre>
```

Flowchart representation of a for Loop

