Boolean Conditions, If-Then

The physical order of a program is the order in which the statements are *listed.* The logical order of a program is the order in which the statements are *executed*. With **conditional statements** and **control structures** that can change the order in which statements are executed.

Boolean Data Type

A Boolean expression asserts (states) that something is true or false. It is named after the mathematician George Boole.

In Java, the data type **boolean** is used to represent Boolean data. Each **boolean** constant or variable can contain one of two values: **true** or **false**.

Examples:

```
boolean b;
b = true;
b = false;
b = 10 < 12;              // true
b = 5 >= 2;               // false
```

Relational Operators

We generally use relational operators to create boolean expressions. < and >= from the previous sample are examples of relational operators. Relational operators take two operands and test for a relationship between them. The following table shows the relational operators and the Java symbols that stand for them.

Java Symbol	*Relationship*
==	Equal to (not =)
!=	Not equal to (since no key for ≠)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

More examples:

```
boolean b;
b = 10 == 12;             // false
b = 5 != 2;               // true
```

We must be careful when applying the relational operators to floating point operands, particularly equal (==) and not equal (**!=**). Integer values can be represented exactly; floating point values with fractional parts often are not exact in the low-order decimal places. Therefore, you should compare floating point values for near equality. For now, **do not compare floating point numbers for equality. == and != also does not (always) work with Strings.**

For example:

```
double d = 0.05 * 3;
boolean b = (d == 0.15);
System.out.println("Value of d: " + d);
System.out.println("Value of b: " + b);
```

This program outputs:

```
Value of d: 0.15000000000002
Value of b: false
```

Next we will see how to test for a range of values to handle this case.

Boolean Operators

A simple Boolean expression is either a Boolean variable or constant or an expression involving the relational operators that evaluates to either true or false. These simple Boolean expressions can be combined using the logical operations defined on Boolean values. There are three Boolean operators: AND, OR, and NOT. Here is a table showing the meaning of these operators and the symbols that are used to represent them in Java.

Java Symbol	*Meaning*
&&	AND is a binary Boolean operator. If both operands are true, the result is true. Otherwise, the result is false. **True** **False** **True** True False **False** False False

<table>
<tr><td>||</td><td>OR is a binary Boolean operator. If at least one of the operands is true, the result is true. Otherwise, the result is false.
<table><tr><th></th><th>True</th><th>False</th></tr><tr><td>True</td><td>True</td><td>True</td></tr><tr><td>False</td><td>True</td><td>False</td></tr></table></td></tr>
<tr><td>!</td><td>NOT is a unary Boolean operator. NOT changes the value of its operand: If the operand is true, the result is false; if the operand is false, the result is true.
<table><tr><th></th><th>Not_Value</th></tr><tr><td>True</td><td>False</td></tr><tr><td>False</td><td>True</td></tr></table></td></tr>
</table>

Some simple examples:

```
boolean b;
b = (x <= 10) || (x > 20);       // True if x <= 10 or x > 20
b = (x < 10) && (y == x);        // True if x <10 and y is x
b = (d > 0.1499) && (d < 0.1501); // True if d is about 0.150
b = (x < 10) && (x > 20);        // Always false
```

Java uses *short-circuit evaluation*. The evaluation is done in left-to-right order and halts as soon as the result is known. For example, in the expression:

```
b = (x != 0) && ((y / x) > 2);
```

If x is 0 then `(x != 0)` is false. It doesn't matter whether `((y/x)>2)` is true or false because we will && the result with false, getting false. So Java doesn't bother to continue evaluating `((y/x)>2)`. This is quite desirable in this case, because if x was 0 we could end up with a division by 0 error. We can also use short-circuit evaluation if there is an || and we find an expression to be true.

If relational operators and Boolean operators are combined in the same expression in Java, the Boolean operator NOT (**!**) has the highest precedence, the relational operators have next higher precedence, and the Boolean operators AND (**&&**) and OR (||) come last (in that order). Expressions in parentheses are always evaluated first.

For example, given the following expression (**stop** is a **bool** variable)

```
count <= 10 && sum >= limit || !stop
```

!stop is evaluated first, the expressions involving the relational operators are evaluated next, the **&&** is applied, and finally the **||** is applied.

It is a good idea to use parenthesis to make your expressions more readable, e.g:

```
(((count <= 10) && (sum >= limit))  ||  (!stop))
```

This also helps avoid difficult-to-find errors if the programmer forgets the precedence rules.

The following table summarizes the precedence of some of the common Java operators:.

Operator	Type	Order of Evaluation
( ) [ ] .	Parentheses Array subscript Member access	left to right
++ --	Prefix increment, decrement	right to left
++ -- -	Postfix increment, decrement Unary minus	right to left
* / %	Multiplicative	left to right
+ -	Additive	left to right
< > <= >=	Relational	left to right
== !=	Equality	left to right
&&	And	left to right
||	Or	left to right
? :	Conditional	right to left
= += -= *= /= %=	Assignment	right to left

If-Then and If-Then-Else Statements

The If statement allows the programmer to change the logical order of a program; that is, make the order in which the statements are executed differ from the order in which they are listed in the program. The If-Then statement uses a Boolean expression to determine whether to execute a statement or to skip it. The format is as follows:

```
if (boolean_expression)
     statement;
```

The statement will be executed if the Boolean expression is true.

If you wish to execute multiple statements, which is called a *block*, use curly braces:

```
if (boolean_expression)
{
      statement1;
      statement2;
      …
      statement99;
}
```

Although the curly braces are not needed when only a single statement is executed, some programmers always use curly braces to help avoid errors such as:

```
if (boolean_expression)
      statement1;
      statement2;
```

This is really the same as:

```
if (boolean_expression)
      statement1;
statement2;
```

Such a condition commonly arises when initially only a single statement is desired, and then a programmer goes back and adds additional statements, forgetting to add curly braces.

We can also add an optional **else** or **else if** clause to an if statement. The else statement will be executed if all above statements are false. Use else if to test for multiple conditions:

```
if (boolean_expression1)
      statement1;             // Expr1 true
else if (boolean_expression2)
      statement2;             // Expr1 false, Expr2 true
else if (boolean_expression3)
      statement3;             // Expr1, Expr2 false, Expr3 true
…
else
      statement_all_above_failed;   // Expr1, Expr2, Expr3 false
```

Here are some examples of if statements:

```
System.out.println("Today is a ");
if (temperature <= 32)
{
    System.out.println("Cold day.");
    System.out.println("Sitting by the fire is appropriate.");
}
else
{
    System.out.println ("nice day.  How about taking a walk?");
}
```

There is a point of Java syntax that you should note: There is never a semicolon after the right brace of a block (compound statement).

This example outputs if a number is even or odd:

```
if (num % 0 == 0)
{
     System.out.println(num + " is even");
}
else
{
     System.out.println(num + " is odd");
}
```

This example computes how much tax is owed based on the rule of nothing on the first 15000, 5% on income from $15000-$25000, and 10% on income over $25000.

```
if (income <= 15000)
{
     tax = 0;
}
else if (income <= 25000)
{
     tax = 0.05 * (income - 15000);
}
else
{
     tax = 0.05 * (income - (25000 - 15000));
     tax += 0.10 * (income - 25000);
}
```

Finally here is an example using else-if, also referred to as a **nested if statement**:

```
if (y==false)
if (z < 50)
{
}
else
{
  ….
}
```

There may be confusion as to what the final else statement goes to. Does it match up with z<50? or with y==false? The rule is that the else is paired with the most recent if statement that does not have an else. In this case, the final else statement is paired with (z < 50). The above is equivalent to:

```
if (y==false)
{
     if (z < 50)
     {
        …
     }
     else
     {
      ….
     }
}
```

If we wanted the else to match up with y==false, we should change the braces accordingly:

```
if (y==false)
{
     if (z<5)
     {
          …
     }
}
else
{
     …
}
```

In nested If statements, there may be confusion as to which **if** an **else** belongs. In the absence of braces, the compiler pairs an **else** with the most recent **if** that doesn't have an **else**. You can override this pairing by enclosing the preceding **if** in braces to make the **then** clause of the outer If statement complete.

Common Bug #1: Confusing = and ==

The assignment operator (=) and the equality test operator (==) can easily be miskeyed one for the other. What happens if this occurs? Fortunately, the program will not compile. Look at the following statements.

```
int i=0;
i == i + 1;
System.out.println(i);
```

This code fragment generates an error during compilation. i==i+1 will be flagged as an improper instruction.

Look at the next statement going the other direction:

```
int i=0;
if (i=1)
{
     System.out.println("Value is 1");
}
else
{
     System.out.println("Value is 0");
}
```

This code will also be flagged as an error by the compiler. i=1 does not return a Boolean, and we must make a Boolean comparison in the if-statement.

Fortunately, these common problems are discovered by the Java compiler. However, if you start to program in C or C++, these statements will **not** be flagged as errors by the compiler because they are valid statements. Unfortunately, they are probably not statements you wish to make and will likely result in a program that does not function correctly.

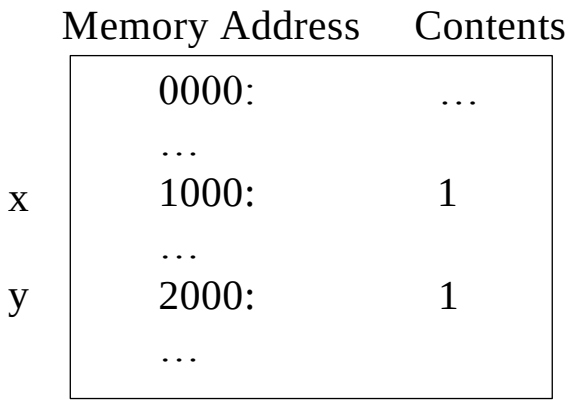
Common Bug #2: Using == with Strings and Objects

Fortunately, the compiler catches the previous bugs (unless the data types happen to be Boolean). Not so for the next common bug, using == with Strings. Although == correctly tests two values for primitive data types like numbers and chars, it has a different meaning when applied to objects.

First, let's see how == works correctly on primitive data types. All variables are stored at some memory address. For variables of primitive data types, the value is stored directly in memory at that address. For example, say that we create two integers:

```
int x=1,y=1;
```

Let's say that the compiler decides to place variable x at address 1000 and variable y at address 2000. A snapshot of memory looks something like this:



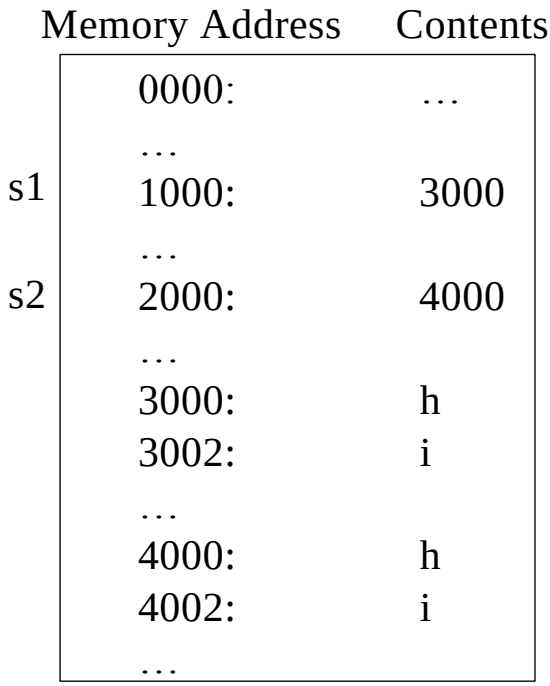
When Java executes x == y, the == operator checks to see if the contents of memory corresponding to variables x and y are the same. In this case, we compare 1 from address 1000 with 1 from address 2000, they are identical, and Java correctly returns the boolean value true.

Objects such as strings are stored differently, resulting in a different behavior when we use ==. An object really occupies a number of bytes of memory. These bytes store data associated with the object (e.g., a string of characters). The variable that represents the object is really storing the **memory address** where the data is stored.

For example, say that we create a String object s1:

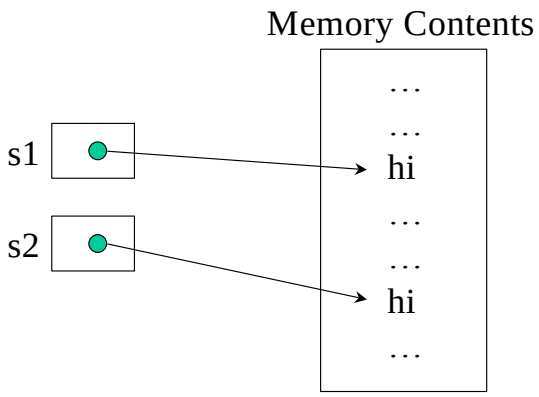
```
String s1 = "hi", s2 = "hi";
```

The variable s1 is also stored somewhere in memory, let's say at address 1000. The variable s2 is also stored somewhere, let's say at address 2000. The contents of the two object variables contains the memory address of the place where we are putting the object. Let's say we have "hi" for s1 stored at address 3000, and "hi" for s2 stored at address 4000:



What happens now if we execute: s1 == s2 ? Java will do the same thing it did before: it compares the contents of s1 with the contents of s2. In this case, it is comparing 3000 with 4000. These are actually memory addresses, and they are different, so Java will return back the value false. In this case, the == operator isn't smart enough to know that we actually want to go look at the data stored at the memory addresses and compare them instead.

Whenever we create variables for objects that actually store memory addresses, these variables are called **references** or **pointers**. Graphically, we can depict them a bit more conveniently using arrows and boxes:



Here is a program that illustrates the above problem (if you actually run the above, Java may store the "hi" in the same place and == may actually work):

```
boolean b;
String s1,s2;
Scanner keyboard = new Scanner(System.in);

s1 = keyboard.nextLine();
s2 = keyboard.nextLine();

System.out.println("S1 = " + s1);
System.out.println("S2 = " + s2);
if (s1 == s2)
     System.out.println("Equal");
else
     System.out.println("Not equal");
```

Try running this program with the same input. Java will say they are not equal.

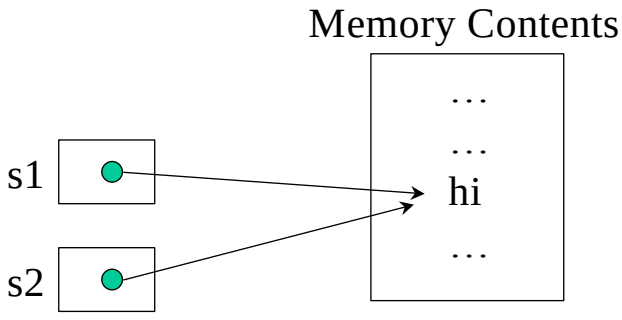
As one further example, consider the following change:

```
boolean b;
String s1,s2;
Scanner keyboard = new Scanner(System.in);

s1 = keyboard.nextLine();
s2 = s1;

System.out.println("S1 = " + s1);
System.out.println("S2 = " + s2);
if (s1 == s2)
     System.out.println("Equal");
else
     System.out.println("Not equal");
```

Using our hypothetical values from earlier, we now have a picture that looks like:



Both variables are now referring to the same location due to the assignment, so the program now prints out "Equal". You might wonder why such behavior might be

useful. This will become much more useful later on when we discuss linked data structures.

For now then, how do we compare strings to see if they are the same? We need to compare every character of each string to see if they match. Fortunately, there is already a string method defined that does this for us, called equals():

```
s1 = keyboard.readLine();
s2 = keyboard.readLine();

System.out.println("S1 = " + s1);
System.out.println("S2 = " + s2);
if (s1.equals(s2))
      System.out.println("Equal");
else
      System.out.println("Not equal");
```

There is also a method called equalsIgnoreCase() that checks for equality but treats upper and lower case the same.

There is also a method called compareTo() that compares two strings and returns a number based on lexicographic ordering (i.e. if one is alphabetically greater than or less than the other). This will be useful later when we do things like sort lists of names.