Primitive Data Types
Variables

*variable*: a piece of computer memory that holds a data value

Two parts to every variable:

1. *identifier*: the name by which we refer to the variable
2. *data type*: the type of data the variable holds (e.g., string, number, boolean)
Types of Data Type

Two categories: *primitive type* and *class type*

**Primitives**
represents basic data types
examples:
- `char`  // holds a single character
- `int`   // holds integer values
- `double` // holds decimal values
- `boolean` // holds true/false values

**Classes**
represents more complex data
examples:
- `String`  // ** holds textual data
- `Scanner` // reads input
- `Date`    // represents day/month/year
- `Math`    // complex mathematical ops
<table>
<thead>
<tr>
<th>Data</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>“Carpe Diem”</td>
<td>42</td>
<td>true</td>
</tr>
<tr>
<td>text</td>
<td>3.14159</td>
<td>false</td>
</tr>
<tr>
<td>numbers</td>
<td>logical values</td>
<td></td>
</tr>
</tbody>
</table>
Data

“Carpe Diem” 42 3.14159 true

text numbers logical values
## Primitive Data Types in Java

### Integer Numeric Types (can only be whole numbers)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>-128 through 127</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32678 through 32677</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2147483648 through 2147483647</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>-9223372036854775808 through 9223372036854775807</td>
<td></td>
</tr>
</tbody>
</table>

### Decimal Numeric Types (can be whole or decimal numbers)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>7 decimal digits of accuracy</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>15 decimal digits of accuracy</td>
</tr>
</tbody>
</table>

### Character Type

- **char**: 2 bytes, any keyboard character

### Logical Type

- **boolean**: 1 byte, true or false
Declaration & Initialization of Primitive Variables

declare a single variable

```c
int age;
```

initialize a primitive variable

```c
age = 29;
```

declare & initialize a single primitive variable

```c
int age = 29;
```

declare & initialize multiple primitive variables of the same type

```c
int age = 29, weight, temp = -10;
```
Declaring & Initializing Numeric Data Types

integer numeric types

```java
int age = 29;
int temp = -4;
```

decimal numeric types

```java
double height = 5.33;
double length = 5.0;  // note the use of the decimal!
double width = 3;    // note the lack of a decimal!
double outdoorTemp = -4.25;
double mole = 6.022E23;
double verySmallNumber = 5.6E-15;
```
# Numerical Operators in Java (int)

## Unary Prefix Operator
- negation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>negation</td>
<td>-6</td>
</tr>
</tbody>
</table>

## Binary Infix Operators
- addition
- subtraction
- multiplication
- division (quotient)
- modulus, mod (remainder)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>6 + 4 (= 10)</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>6 - 4 (= 2)</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>6 * 4 (= 24)</td>
</tr>
<tr>
<td>/</td>
<td>division (quotient)</td>
<td>6 / 4 (= 1)</td>
</tr>
<tr>
<td>%</td>
<td>modulus, mod (remainder)</td>
<td>6 % 4 (= 2)</td>
</tr>
</tbody>
</table>

## Unary Prefix/Postfix Operators
- increment by 1
- decrement by 1

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>increment by 1</td>
</tr>
<tr>
<td>--</td>
<td>decrement by 1</td>
</tr>
</tbody>
</table>
Division & Modulus (Mod) for int

Division of two integers results in two values: the quotient and remainder

*quotient* describes how many times the divisor goes into the dividend
*remainder* describes the amount “left over” from the division

<table>
<thead>
<tr>
<th>traditional math</th>
<th>int math</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 / 4 = 4.75</td>
<td>19 / 4 = 4</td>
</tr>
<tr>
<td>= 4 3/4</td>
<td>19 % 4 = 3</td>
</tr>
<tr>
<td></td>
<td>// 3/4</td>
</tr>
</tbody>
</table>
Operator Precedence

Will work the same way you’re familiar with from math
work from left to right across a mathematical statement, starting with highest precedence
mod has the same level of precedence as multiply and divide

\[ 2 + \frac{19}{(4 + 1)} - 5 \mod 3 \]
\[ 2 + \frac{19}{5} - 5 \mod 3 \]
\[ 2 + 3 - 5 \mod 3 \]
\[ 2 + 3 - 2 \]
\[ 5 - 2 \]
\[ 3 \]
# Numerical Operators in Java (double)

## Unary Prefix Operator
- negation

## Binary Infix Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>6.2 + 4.1 (= 10.3)</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>6.2 - 4.1 (= 2.1)</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>6.2 * 4.1 (= 25.42)</td>
</tr>
<tr>
<td>/</td>
<td>division (quotient)</td>
<td>6.2 / 4.1 (= 1.51...)</td>
</tr>
<tr>
<td>%</td>
<td>modulus, mod (remainder)</td>
<td>6.2 % 4.1 (= 2.10...)</td>
</tr>
</tbody>
</table>

## Unary Prefix/Postfix Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>increment by 1</td>
</tr>
<tr>
<td>--</td>
<td>decrement by 1</td>
</tr>
</tbody>
</table>

*NOTE:* you will rarely (if ever) use this with doubles!
Prefix/Postfix Increment/Decrement (int & double)

Frequently want to increase/decrease an int/double variable by 1

We can use the increment/decrement operators as shorthand to do this

Two forms: prefix and postfix

- prefix has the operator *before* the variable
- postfix has the operator *after* the variable

Always use it by itself!

```c
int age = 29;
age = age + 1;
age = age - 1;

++age; //age = 30 after this line
age++;  //age = 31 after this line

--age; //age = 30 after this line
age--;  //age = 29 after this line

age = age--; //never do this!
age = ++age;  //never do this!
```
# Arithmetic Shortcut Operators (int & double)

```c
int x = 5;
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Equivalent To</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>x += 2;</code></td>
<td><code>x = x + 2;</code></td>
<td><code>x = 7</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>x -= 2;</code></td>
<td><code>x = x - 2;</code></td>
<td><code>x = 3</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>x *= 2;</code></td>
<td><code>x = x * 2;</code></td>
<td><code>x = 10</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>x /= 2;</code></td>
<td><code>x = x / 2;</code></td>
<td><code>x = 2</code></td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>x %= 2;</code></td>
<td><code>x = x % 2;</code></td>
<td><code>x = 1</code></td>
</tr>
</tbody>
</table>
More Complex Operations

What if we want to...

- take the square root of a number?
- display a number in a particular format (e.g., currency)?
- generate a random number?

We can use *classes*, which represent/manipulate more complex data
Math Class

Provides a range of methods for advanced mathematical operations

- square root/powers
- logarithms
- trigonometric functions
- constant values ($e$, $\pi$)
Math Class

returns the result of calculating $<base>^{<exponent>}$ (e.g., $2^3$)

```javascript
Math.pow(<base>, <exponent>);
```

returns the result of calculating $\sqrt{<expression>}$ (e.g., $\sqrt{9}$)

```javascript
Math.sqrt(<expression>);
```

returns the absolute value of $<value>$

```javascript
Math.abs(<value>);
```
Math Class

returns the smaller value between <num1> and <num2>

```java
Math.min(<num1>, <num2>);
```

returns the larger value between <num1> and <num2>

```java
Math.max(<num1>, <num2>);
```

returns the value of π as a double

```java
Math.PI;
```
DecimalFormat Class

Allows us to format numeric values in particular way

  currency
  specific number of decimal places

Uses a pattern String to indicate formatting

  0: displays a digit
  #: displays a digit, unless a leading zero (then omitted)
  .: displays a decimal
  ,: displays a comma
Example: DecimalFormat

double x = 0.329523;
DecimalFormat df1 = new DecimalFormat("0.0");
DecimalFormat df2 = new DecimalFormat("0.00");
DecimalFormat df3 = new DecimalFormat("00.00");
DecimalFormat df4 = new DecimalFormat("#0.00");

System.out.println("X = " + df1.format(x));
System.out.println("X = " + df2.format(x));
System.out.println("X = " + df3.format(x));
System.out.println("X = " + df4.format(x));

X = 0.3
X = 0.33
X = 00.33
X = 0.33
Example: DecimalFormat

double wage, hours;
double pay;

// Ask user for their 'wage' and 'hours' worked
// Calculate their pay for the week

pay = hours \* wage;
System.out.print("Total pay for " + hours + " hours of work ");
System.out.print("is $" + pay);

Enter Wage : 20.00
Enter Hours: 51.0
Total pay for 51.0 hours of work is $1020.0
double wage, hours;
double pay;
DecimalFormat df = new DecimalFormat("$###,##0.00");

// Ask user for their 'wage' and 'hours' worked
// Calculate their pay for the week

pay = hours * wage;
System.out.print("Total pay for " + hours + " hours of work ");
System.out.print("is " + df.format(pay));

Enter Wage : 20.00
Enter Hours: 51.0
Total pay for 51.0 hours of work is $1,020.00
Mixing int & double Values

Sometimes, we might want to mix int & double values.

Consider the following equation; what does it evaluate to?

```
double x = 2.5 + 9 / 2;
```

2.5 + 4;

???

N.B.: uses int division! Java assumes numbers without a decimal (e.g., 3 vs 3.0) are ints when not stored in a variable.

Java requires both inputs of an operator to be of the same data type. Achieves this through the process of coercion.
Coercion

*coercion*: automatically changing a value’s type to enable an operation
always coerced to the widest type necessary

There is a strict ordering on types

<table>
<thead>
<tr>
<th>narrower types</th>
<th>wider types</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 1 byte</td>
<td>-128 through 127</td>
</tr>
<tr>
<td>short 2 bytes</td>
<td>-32678 through 32677</td>
</tr>
<tr>
<td>int 4 bytes</td>
<td>-2147483648 through 2147483647</td>
</tr>
<tr>
<td>long 8 bytes</td>
<td>-9223372036854775808 through 9223372036854775807</td>
</tr>
<tr>
<td>float 4 bytes</td>
<td>7 decimal digits of accuracy</td>
</tr>
<tr>
<td>double 8 bytes</td>
<td>15 decimal digits of accuracy</td>
</tr>
</tbody>
</table>
Mixing int & double Values

double x = 2.5 + 9 / 2;

What if we want to force double division here?
Mixing int & double Values

double x = 2.5 + 9 / 2.0;
    ^    ^
   double int double
    ^    ^
   double 4.5 ;
    ^    ^
   double double
    ^    ^
   double 7.0 ;
Casting

casting: explicitly changing the data type of a value

can cast to a narrower or wider type

always initiated by the programmer

\[(\text{<dataTypeToCastTo>}) \text{<expression>}\];

```c
int num;
num = (int) 5.33;  // results in num = 5
double perc;
perc = 93 / (double) 100;  // results in perc = 0.93
perc = 93 / ((double) 100);  // results in perc = 0.93
perc = (double) num / 100;  // results in perc = 0.05
```
Data

“Carpe Diem”  42  3.14159  true

text  numbers  logical values
The char Data Type

Similar to String, but contains exactly one character

uses single quotes (') instead of double quotes ("")

Has a few operations, but we’re only concerned with assignment for now

Will primarily use it with String methods

```java
String exampleStr = "Hello, home!";
int index = exampleStr.indexOf('h'); //index = 7
char charPos = exampleStr.charAt(5); //charPos = ',',
```

N.B.: char values
Declaring & Initialzing the char Data Type

character type

```java
char letterA = 'a';
char space = ' ';  
char bang = '!' ;
```

can be an escape sequence too

```java
char singleQuote = '\''; 
char tab = '\t';  
char lineBreak = '\n' ;
```
Strings

Strings are a collection of `char` values concatenated together

"This is a string."

<p>| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |</p>
<table>
<thead>
<tr>
<th>Data</th>
<th>“Carpe Diem”</th>
<th>42</th>
<th>3.14159</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>text</td>
<td></td>
<td>numbers</td>
<td>logical values</td>
<td></td>
</tr>
</tbody>
</table>
Logical Data

Can express exactly one of two values: true or false

in programming, we also think of these as 1 (true) and 0 (false)

Operators are used to express logical ideas that can be evaluated

&& (and)

|| (or)

! (not)
&& (and)

Can express whether or not two statements are true

   it is raining and it is cold

   I attend UWL and I am a science major

If one or both of the statements are false, then the entire expression is false

Evaluation:

   0 && 0 is 0
   0 && 1 is 0
   1 && 0 is 0
   1 && 1 is 1
Truth Table: a table where each row corresponds to one combination of inputs, columns for statements give the input values, and subsequent columns give the truth value for the results of individual operators.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P &amp;&amp; Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
&& (and)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Q</td>
<td>P &amp;&amp; Q</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0 = false
1 = true
|| (or)

Can express whether one or both of two statements are true

it is raining or it is cold

I attend UWL or I am a science major

If one or both of the statements are true, then the entire expression is true
Nuances of **∥**

In English, we use “or” to present two mutually exclusive possibilities

   e.g., “Did you have pizza or spaghetti for dinner?”
   possible answers: pizza, spaghetti, neither, both (maybe?)

Logically, the answer could be “yes” or “no”

   no: you had neither
   yes: you had spaghetti, or pizza, or both

Spectrum of possible answers does not work with our logical value system

   we instead work with true (yes) or false (no)
\[ \| (\text{or}) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Q</td>
<td>P | Q</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0 = false
1 = true
!(not)

Can express the opposite value of a single statement

  it is not raining
  I am not a science major

If the statement is true, the expression is false, and vice versa
<table>
<thead>
<tr>
<th>P</th>
<th>!P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = false
1 = true
Expressing More Complex Ideas

Often want to express more complex ideas

“Show up to lab or don’t show up to lab and submit exercise three”

Want to know the outcome of every possible scenario (set of inputs)

Can combine statements into larger expressions

\[ \text{goToLab} \lor (\neg\text{goToLab} \land \text{submitEx3}) \]

How to evaluate possible outcomes?

use truth tables

one statement at a time
Example: Truth Table for Complex Expressions

\[ \text{goToLab} \lor (\neg \text{goToLab} \land \text{submitEx3}) \]

<table>
<thead>
<tr>
<th>goToLab</th>
<th>submitEx3</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Create one column per variable
| Create one column per variable list in alphabetical order

For \( N \) variables, you will have \( 2^N \) additional rows
| For \( N \) variables, you will have \( 2^N \) additional rows in this case, \( 2^2 = 4 \)

Fill rows with every combination of 0s and 1s
| Fill rows with every combination of 0s and 1s easiest way? count in binary

i.e., count using only 0s and 1s
## Counting

<table>
<thead>
<tr>
<th>Decimal</th>
<th></th>
<th></th>
<th>Binary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>17</td>
<td>1</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>18</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>19</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>20</td>
<td>100</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>21</td>
<td>101</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>22</td>
<td>110</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>23</td>
<td>111</td>
<td>1111</td>
</tr>
</tbody>
</table>
# Counting

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>
Example: Truth Table for Complex Expressions

$$\text{goToLab} \ || \ (!\text{goToLab} \ & \ & \ \text{submitEx3})$$

<table>
<thead>
<tr>
<th>goToLab</th>
<th>submitEx3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Create one column per variable

list in alphabetical order

For N variables, you will have $$2^N$$ additional rows

in this case, $$2^2 = 4$$

Fill rows with every combination of 0s and 1s
easiest way? count in binary
i.e., count using only 0s and 1s
## Precedence for Logical Operators

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precedence</td>
<td>()</td>
</tr>
<tr>
<td>negation</td>
<td>!</td>
</tr>
<tr>
<td>logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
</tr>
</tbody>
</table>

### Truth Tables

|   | P | Q | + P || Q |   | P | Q | * P && Q |
|---|---|---|------|---|---|---|---------|
| 0 | 0 | 0 | 0    | 0 | 0 | 0 | 0       |
| 0 | 1 | 1 | 1    | 0 | 1 | 0 | 0       |
| 1 | 0 | 1 | 1    | 1 | 0 | 0 | 0       |
| 1 | 1 | 1 | 1    | 1 | 1 | 1 | 1       |

### Example

It matters!

Work out \( P \&\& Q || R \) two ways: performing \( || \) first and performing \( \&\& \) first.
Example: Truth Table for Complex Expressions

\begin{align*}
\text{goToLab} \lor (\neg \text{goToLab} \land \text{submitEx3})
\end{align*}

\begin{tabular}{|c|c|c|c|}
\hline
\text{goToLab} & \text{submitEx3} & \neg \text{goToLab} & \neg \text{goToLab} \land \text{submitEx3} \\
\hline
0 & 0 & 1 & 0 \\
\hline
0 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|}
\hline
\text{goToLab} & \neg \text{goToLab} \land \text{submitEx3} & \text{goToLab} \lor (\neg \text{goToLab} \land \text{submitEx3}) \\
\hline
0 & 0 & 0 \\
\hline
0 & 1 & 0 \\
\hline
1 & 0 & 0 \\
\hline
1 & 1 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\text{P} & \text{Q} & \text{P} \lor \text{Q} \\
\hline
0 & 0 & 0 \\
\hline
0 & 1 & 1 \\
\hline
1 & 0 & 1 \\
\hline
1 & 1 & 1 \\
\hline
\end{tabular}
The boolean Data Type

Can only contain one of two values: true or false

Declaration/initialization/assignment work the same as int, double, char

Uses the logical operators (i.e., !, ||, &&)

```java
boolean entree = true;
boolean salad = false;
boolean soup = true;

boolean validOrder = entree && (salad || soup);
```

What is validOrder set to?

true
boolean Operators

Uses the logical operators (i.e., !, ||, &&)
Also uses **relational** and **equality operators**

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precedence</td>
<td>()</td>
</tr>
<tr>
<td>negation</td>
<td>!</td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
</tr>
</tbody>
</table>
Relational and Equality Operators

< (less than)
  8 < 3 (false), 3 < 8 (true)

> (greater than)
  8 > 3 (true), 3 > 8 (false)

<= (less than or equal to)
  6 <= 6 (true), 6 <= 7 (true)
  7 <= 6 (false)

>= (greater than or equal to)
  6 >= 6 (true), 6 >= 7 (true)
  7 >= 6 (false)

== (equality)
  6 == 6 (true), 8 == 3 (false)

!= (inequality)
  6 != 6 (false), 8 != 3 (true)
Example: boolean Expressions

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator(s)</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precedence</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>negation</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td></td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td>logical AND</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

boolean x = 2.5 > 3 || !(4 != 5.1);

double x = 2.5 > 3 || !(true);

false || false ;

false ;
## Operator Precedence

We can mix types, operators in a single expression

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precedence</td>
<td>()</td>
</tr>
<tr>
<td>prefix</td>
<td>!, –</td>
</tr>
<tr>
<td>multiplicative</td>
<td>*, /, %</td>
</tr>
<tr>
<td>additive</td>
<td>+, –</td>
</tr>
<tr>
<td>relational</td>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
boolean x = 2.5 + 4 > 3 || !(4 % 2 != 5.1);
2.5 + 4 > 3 || !(0 != 5.1);
2.5 + 4 > 3 || !(true);
2.5 + 4 > 3 || false;
6.5 > 3 || false;
true || false;
true;
```
Short-Circuit Evaluation

Two situations where evaluation of && and || will be terminated early

false && …
true || …

Java will always compute the lefthand side of an operator first

2.5 + 4 > 3 || !(4 % 2 != 5.1)
6.5 > 3 || !(4 % 2 != 5.1)
true || !(4 % 2 != 5.1)
true
Short-Circuit Evaluation

Two situations where evaluation of && and || will be terminated early

false && ...
true || ...

Java will always compute the lefthand side of an operator first

2.5 + 4 > 3 || !(4 % 2 != 5.1)
6.5 > 3 || !(4 % 2 != 5.1)
true || !(4 % 2 != 5.1)
true

int num = ...; //user input
boolean divByNum;
divByNum = 2 >= 1 / num;
divByNum = num != 0 && 2 >= 1 / num;

entire expression evaluates to false if num != 0 is false (i.e., if num is 0)