

# 3.1

## Boolean Expressions

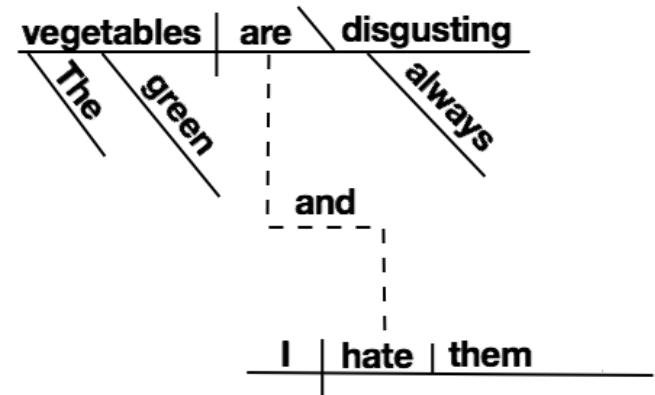
# Programming Language Grammar

Before we talk about boolean expressions, let's talk about grammar, which is what expressions are part of.

Expressions are part of Java's grammar. Programming languages have grammar just like natural languages... but they're made up of different elements than nouns, verbs, adjectives, etc.

Three of the elements in Java's grammar are:

- Expressions
- Statements
- 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.

Examples:

`false`

`2+2`

`1.0/2.0`

`"Hello, world"`

`player.getLocation()`

`System.out.println("Hello, world")`

`x + y == 4`

`x = 2`

# Expressions

An expression evaluates to a single value, and that value has a type.

Examples:

**false** – boolean

**2+2** – int

**1.0/2.0** – double

**“Hello, world”** – String

**player.getLocation()** – Room

**System.out.println(“Hello, world”)** – void

**x + y == 4** – boolean

**x = 2** – int, if x is type int

# Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution.

The body of a method is a series of statements.

Examples:

```
System.out.println("Hello, world!");
```

```
return Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
```

```
player.setLocation(player.getLocation()).getNorth();
```

```
x = 3;
```

# Expression Statements

A lot of the statements we've used are expressions with a semicolon at the end.

These are called Expression Statements.

Examples:

```
System.out.println("Hello, world!");
```

```
return Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
```

```
player.setLocation(player.getLocation()).getNorth());
```

```
x = 3;
```

(The return is not an Expression Statement but still has an expression in it.)

# Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

```
{  
    statement1  
    statement2  
    ...  
    statementN  
}
```

# Blocks

This means you could write

```
public class Main {  
    public static void main(String args[]) {  
        {  
            {  
                {  
                    System.out.println("Hello, world");  
                }  
            }  
        }  
    }  
}
```

You wouldn't, but you could!



# Syntactically (in)significant whitespace

Java statements are terminated by semicolons. Java blocks are delimited by {}.

Whitespace (\n, \t, space) in Java is **syntactically insignificant**. (Not Python!)

These programs are equivalent:

```
public class Hello {  
    public static void main(String args[]) {  
        System.out.println("Hello, world!");  
    }  
}
```

```
public class Hello{public static void main(String args[]){System.out.println("Hello, world!");}}
```

We indent Java code nicely to make it readable for programmers... but Java is indifferent. Code can even be deliberately obfuscated.

# Heritage of ; and {}

Back in the 90s, AP Comp Sci was taught in Pascal...

Note "begin" and "end" became "{" and "}" in C/C++/Java

Statements end in semicolons, and the whole program ends with a period, like an English sentence.

```
program Printing;

var i : integer;

procedure PrintAnInteger(j : integer);
begin
    ...
end;

function triple(const x: integer): integer;
begin
    triple := x * 3
end;

begin { main program }
    ...
    PrintAnInteger(i);
    PrintAnInteger(triple(i))
end.
```

# International Obsfuscated C Code Contest (IOCCC)

```

#include<stdlib.h>
#include /*haha*/
#include <stdio.h>
#define Bwahaha
#define Haha/**/
/*Don*/ /*Yang*/
char*ha =/*Bwaha hahahaha */"\40archive"/*
Bwahaha */"\x20" "from..."; ;typedef unsigned Foo;
#define Ha/!*/ printf ( " " /*yadaya dayada */
/*^_^*/ /*@_@*/ ;void main( void) { int i= 0;
Foo foo =~(Foo) time(+ NULL); srand( foo);! Ha
Bwahaha Haha/*! */"%s" "\n\n" ,ha);; for(i= 0;
i<19;i= i+1){for (foo=( Foo)/* haha*/ rand() %
70;foo> 0;foo = foo-1) Ha);Ha "Don!" "\r\n"
);}Ha/* */"\40" "\x20" "\x20" "\x20"
"\t \t" "Da\40" "Man!" "\n"); for(i= 67;i>0 /*/
*/;i--) Ha);/* ^_^ */ printf (" \40" "(%d/" "%d/"
"%-2d)", 0x3,+4 , 'b'); Ha);Ha );Ha); Ha);Ha );Ha);;
Ha);Ha); Ha);Ha); Ha);; Ha);; Ha);Ha );Ha); Ha);;
exit(EXIT_SUCCESS) ;i+=-i++; /*(c) DON YANG,1998 */}
```

# Boolean Expressions

A boolean expression is any expression that evaluates to type boolean. Boolean expressions are important for **control flow**.

The simplest boolean expressions are simply

true

false

... but most boolean expressions use boolean operators

# Equality Operators

<code>x == y</code>	true if x is equal to y, false otherwise
<code>x != y</code>	true if x is not equal to y, false otherwise

Keep in mind that you must use "==" , not "=", when testing if two values are equal.

= is the assignment operator and will assign the value of y to x!

# Equality Operators

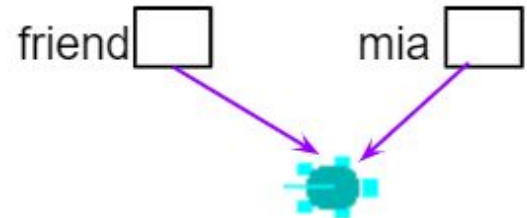
For object types, `==` and `!=` compare whether the two sides are references to the same object... not whether anything else about the objects are equal, such as the characters in two Strings.

For Strings, remember to use `equals()` and not the `==` or `!=` operators.

```
Turtle juan = new Turtle(world);  
Turtle mia = new Turtle(world);
```

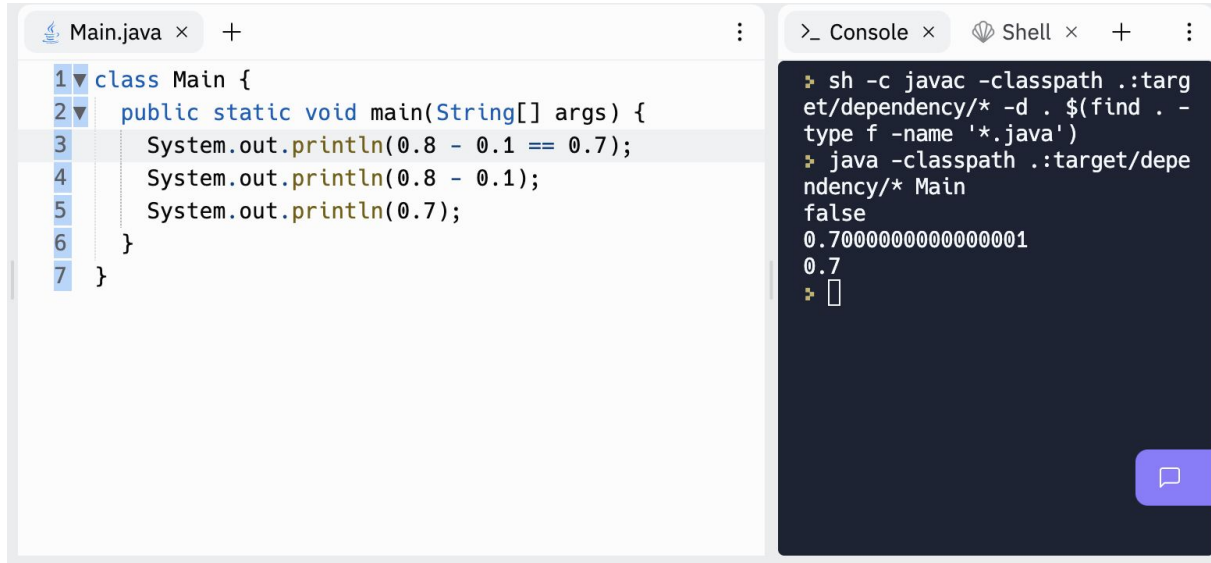


```
Turtle friend = mia;
```



# Equality Operators

Even primitive types can surprise you...



The screenshot shows an IDE with two panels. The left panel displays a Java file named `Main.java` with the following code:

```
1 class Main {  
2     public static void main(String[] args) {  
3         System.out.println(0.8 - 0.1 == 0.7);  
4         System.out.println(0.8 - 0.1);  
5         System.out.println(0.7);  
6     }  
7 }
```

The right panel shows the console output of the program:

```
>_ Console x Shell x +  
✦ sh -c javac -classpath .:target/dependency/* -d . $(find . -  
type f -name '*.java')  
✦ java -classpath .:target/dependency/* Main  
false  
0.7000000000000001  
0.7  
✦ []
```

0.1 is terminating in decimal, but repeats forever in binary: 0.000110011...

Financial applications use "binary coded decimals" instead of float / double types to avoid this "round off error".

# Equality Operators: operand types

The equality operators may be used to compare two operands that are convertible ([§5.1.8](#)) to numeric type, or two operands of type boolean or Boolean, or two operands that are each of either reference type or the null type. All other cases result in a compile-time error.

```
String string1="x", string2="y";  
double double1=1.0, double2=2.0;  
int int1=1, int2=2;  
boolean boolean1=false, boolean2=true;
```

```
System.out.println(string1 == string2); // Allowed but usually not what you want  
System.out.println(double1 == int1); // Allowed, both are numeric  
System.out.println(boolean1 == int1); // Compile time error  
System.out.println(string1 == int1); // Compile time error
```



# Relational Operators

$x > y$	true if x is greater than y, false otherwise
$x < y$	true if x is less than y, false otherwise
$x \geq y$	true if x is greater than or equal to y, false otherwise
$x \leq y$	true if x is less than or equal to y, false otherwise

Tip: To remember  $\geq$  and  $\leq$ , think of it as the order in which you say it. Greater than ( $>$ ) or equal to ( $=$ )

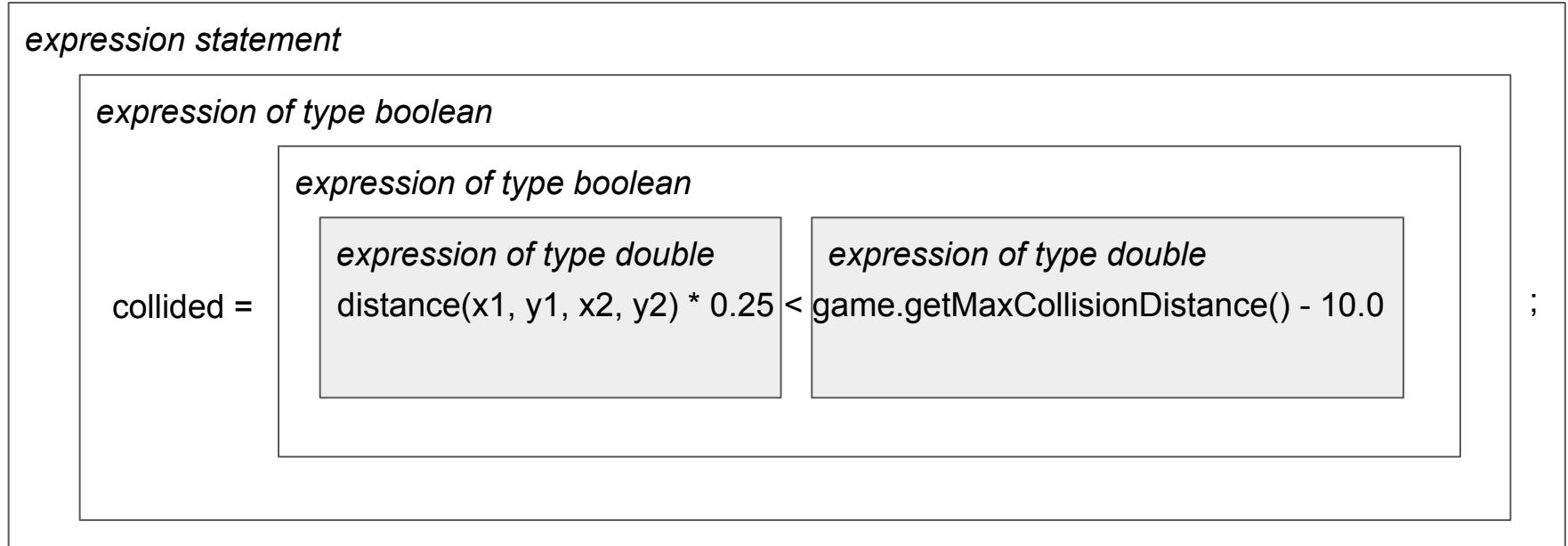
# Relational Operators

$x > y$	true if x is greater than y, false otherwise
$x < y$	true if x is less than y, false otherwise
$x \geq y$	true if x is greater than or equal to y, false otherwise
$x \leq y$	true if x is less than or equal to y, false otherwise

These don't work on objects, such as Strings! This is where you use the `String.compareTo` method.  
(`compareTo` isn't part of `Object` like `equals` - it comes from the `Comparable` interface.)

# Nesting of Expressions

Operators in Java like == and < take operands. The operands are themselves expressions. Expressions thus can nest inside other expressions.



# Operator Precedence

Java has an order in which it evaluates operators, just like PEMDAS tells you to multiply/divide before adding/subtracting when doing math.

So  $x + 2 < y + 3$  and  $(x+2) < (y+3)$  are equivalent because  $+$  has higher precedence than  $<$ .

When in doubt, use parentheses, and sometimes it's best to add parentheses to make code more readable.

Operator Precedence	
Operators	Precedence
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^=  = <<= >>= >>>=

# Modulo Operator (%)

<code>x % y</code>	returns the remainder of dividing x by y
--------------------	--

This is NOT a boolean operator, but it's a handy one frequently used with operators like `==`. `%` works on integers but also on floats/doubles in Java.

One use is checking whether a number is even or odd.

`x % 2 == 0` means x is even

`x % 2 == 1` means x is odd

You can use it to check if a number is an even multiple of another:

`x % y == 0` means x is a multiple of y

(can be divided by y with a remainder of 0)

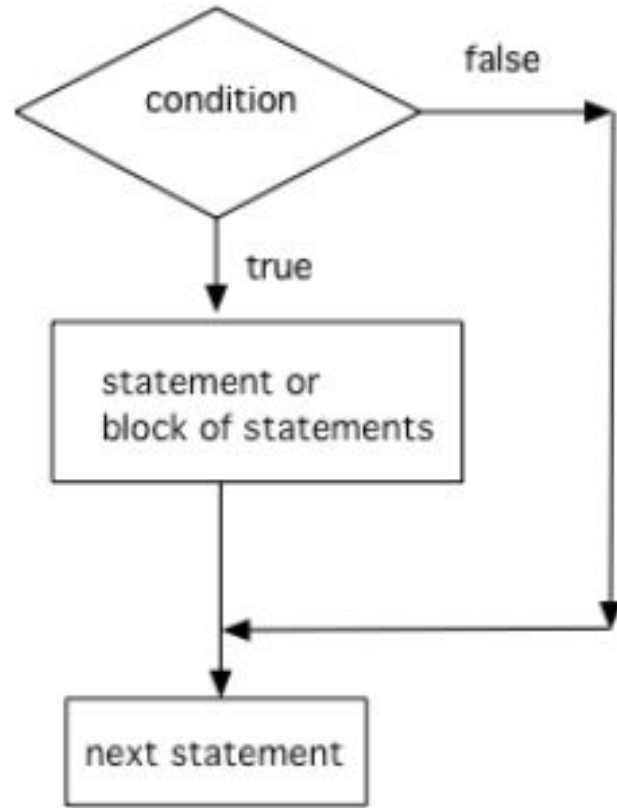
# 3.2

if statement

Let's put those booleans to work!

```
// Condition
if (boolean expression) {
    // Statement or block of statements
    Do statement;
}
```

```
// Next statement
Do statement;
```



Pro-tip: Always use { } in your if statements

**Not advised:**

if (boolean expression)

Do statement;

**Better:**

if (boolean expression) {

Do statement;

}

**Note:** This is something we call "style". A programming **style** is a set of programming conventions and best practices. **Consistently** following a basic Java style will help you avoid errors and mistakes.



# Practice!

## Replit: fizzbuzz

Main.java x +

```
1 class Main {
2     //
3     // Fizz buzz is a group word game for children to teach them about division.
4     // Players take turns to count incrementally, replacing any number divisible
5     // by three with the word "fizz", and any number divisible by five with the word "buzz".
6     // (https://en.wikipedia.org/wiki/Fizz\_buzz)
7     //
8     // This is also a common programming interview question!
9     //
10    // Exercises:
11    // 1. In FizzBuzz.java, implement the calcFizzBuzz method.
12    // 2. In TestSuite.java, implement the missing methods.
13    //
14
15    private static void printFizzBuzz(int i, int n) {
16        System.out.println("FizzBuzz("+i+") = " + FizzBuzz.calcFizzBuzz(i));
17        if (i < n) {
18            // Looping using recursion since we haven't officially learned loop statements yet!
19            printFizzBuzz(i+1, n);
20        }
21    }
22
23    public static void main(String[] args) {
24        printFizzBuzz(1, 50);
25
26        TestSuite testSuite = new TestSuite();
27        testSuite.runTests();
28    }
29 }
```

# TDD (Test Driven Development)

```
TestSuite.java x +
23
24 private void assertFizzBuzzEquals(int i, String expected) {
25     // TODO
26     // Your implementation should call FizzBuzz.calcFizzBuzz(i)
27     // and check it against the expected return value "expected".
28     //
29     // If it matches, add 1 to successfulTests.
30     // If it does not match, add 1 to failedTests and print out
31     // the value of i, the expected value, and the actual value.
32     //
33     // In both cases, add 1 to numTests to track the total number
34     // of tests performed.
35 }
36
37 private void testFizzBuzz() {
38     assertFizzBuzzEquals(1, "1");
39     assertFizzBuzzEquals(2, "2");
40     assertFizzBuzzEquals(3, "Fizz");
41     assertFizzBuzzEquals(4, "4");
42     assertFizzBuzzEquals(5, "Buzz");
43     assertFizzBuzzEquals(6, "Fizz");
44     assertFizzBuzzEquals(7, "7");
45     assertFizzBuzzEquals(8, "8");
46     assertFizzBuzzEquals(9, "Fizz");
47     assertFizzBuzzEquals(10, "Buzz");
48     assertFizzBuzzEquals(11, "11");
49     assertFizzBuzzEquals(12, "Fizz");
50     assertFizzBuzzEquals(13, "13");
51     assertFizzBuzzEquals(14, "14");
52     assertFizzBuzzEquals(15, "FizzBuzz");
53     assertFizzBuzzEquals(16, "16");
54     assertFizzBuzzEquals(17, "17");
55     assertFizzBuzzEquals(18, "Fizz");
56     assertFizzBuzzEquals(19, "19");
57     assertFizzBuzzEquals(20, "Buzz");
58     assertFizzBuzzEquals(21, "Fizz");
59     assertFizzBuzzEquals(22, "22");
60     assertFizzBuzzEquals(23, "23");
61     assertFizzBuzzEquals(24, "Fizz");
62     assertFizzBuzzEquals(25, "Buzz");
63     assertFizzBuzzEquals(26, "26");
64     assertFizzBuzzEquals(27, "Fizz");
65     assertFizzBuzzEquals(28, "28");
66     assertFizzBuzzEquals(29, "29");
67     assertFizzBuzzEquals(30, "FizzBuzz");
68 }
69 }
```