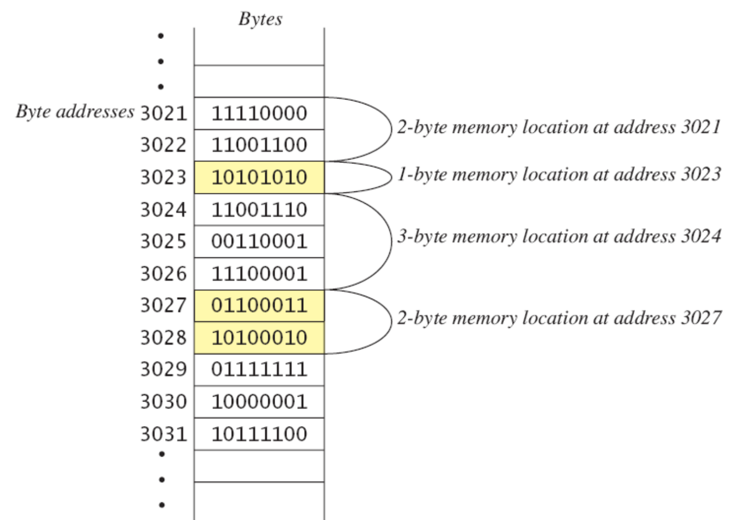
MODULE 02:
BASIC COMPUTATION IN JAVA

Outline

- Variables
  - Naming Conventions
- Data Types
  - Primitive Data Types
    - Review: int, double
    - New: boolean, char
  - The `String` Class
  - Type Conversion
- Expressions
  - Assignment
  - Mathematical
  - Boolean
- Sequential Execution

Variables

- *Variables* store data such as numbers and letters.
  - Think of them as places to store data.
  - They are implemented as memory locations.
- The data stored in a variable is called its *value.*
  - The value is stored in the memory location.
- Its value can be changed.
- In Java, we always have to declare (define) a variable before we can use it



Variables: Naming Convention

- A variable's name should suggest its use
  - e.g. **taxRate**
- Begin with lowercase, uppercase each new word
  - **int** totalWidgets;
  - Called “lower camel case”

Data Types

- A *primitive type* is used for simple, non-decomposable values such as an individual number or individual character.
  - **int**, **double**, and **char** are primitive types.
- A *class type* is used for a class of objects and has both data and methods.
  - **"Java is fun"** is a value of class type **String**

DEFINITION

- A ***data type*** is a set of values and the legal operations on those values.

Variables and Data Types

- Variables
  - Stores information your program needs
  - Each has a unique name
  - Each has a specific type

Java built-in type	what it stores	example values	operations
`int`	integer values	`42` `1234`	add, subtract, multiply, divide, remainder, compare, increment, decrement
`double`	floating-point values	`9.95` `3.0e8`	add, subtract, multiply, divide, compare
`char`	characters	`'a', 'b', '!', '2'`	compare
`String`	sequence of characters	`"Hello world!"` `"I love this!"`	concatenate
`boolean`	truth values	`true` `false`	and, or, not

Data Types: Integers

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
  - Careful! Remainder is discarded in result
  - e.g. 23/5 = 4
- Remainder: %
  - Remainder is the result
  - e.g. 23%5 = 3
- Comparision:
  - Greater then, greater than or equal:> , >=
  - Less than, less than or equal: <, <=
  - Equal: ==
  - Not equal: !=
- Increment, Decrement: ++, --

Java built-in type	what it stores	example values	operations
`int`	integer values	42 1234	add, subtract, multiply, divide, remainder, compare, increment, decrement

Data Types: Floating Point Numbers

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
  - With floating point numbers, the remainder is part of the result
  - e.g. 23.0/5.0 = 4.6
- Comparision:
  - Greater then, greater than or equal:> , >=
  - Less than, less than or equal: <, <=
  - Don't use == or != to test for equality
    - Because of the way numbers are stored, what appears to be an "equal" floating point number is not exactly the same

Java built-in type	what it stores	example values	operations
`double`	floating-point values	`9.95` `3.0e8`	add, subtract, multiply, divide, compare

Data Types: Characters

- Comparision:
  - Greater then, greater than or equal:> , >=
  - Less than, less than or equal: <, <=
  - Equal: ==
  - Not equal: !=

Java built-in type	what it stores	example values	operations
char	characters	`'a', 'b', '!', '2'`	compare

Characters

- char data type
  - Holds a single character
  - Single apostrophe, e.g. 'a', 'z'

```
public class CharExample
{
   public static void main(String [] args)
   {
      char ch1 = 'y';
      char ch2 = 'o';
      String result = "" + ch1;

      result = result + ch2;
      result = result + ch2;
      result = result + ch2;

      System.out.println(result);
   }
}
```

Double quotes with nothing in between, an empty String

```
% java CharExample
yooo
```

Data Types: Strings (of Characters)

- Concatentation: +
  - e.g. “Hello” + “World” becomes “HelloWorld”
  - If you want a space in there, you need to put it in the string
    - Either: “Hello “ + “World”
    - Or: “Hello” + “ World”
  - Any number of strings can be concatenated using the **+** operator.

Java built-in type	what it stores	example values	operations
`String`	sequence of characters	`"Hello world!"` `"I love this!"`	concatenate

Concatenating Strings and Integers

- `String` data type
  - A sequence of characters
  - Double quote around the characters
  - Concatenation using the + operator

```
String firstName = “Michele";
String lastName  = "Van Dyne";
String fullName  = firstName + " " + lastName;
String favNumber = "42";

System.out.println(fullName +
                   "'s favorite number is " +
                   favNumber);
```

```
Michele Van Dyne's favorite number is 42
```

String Methods

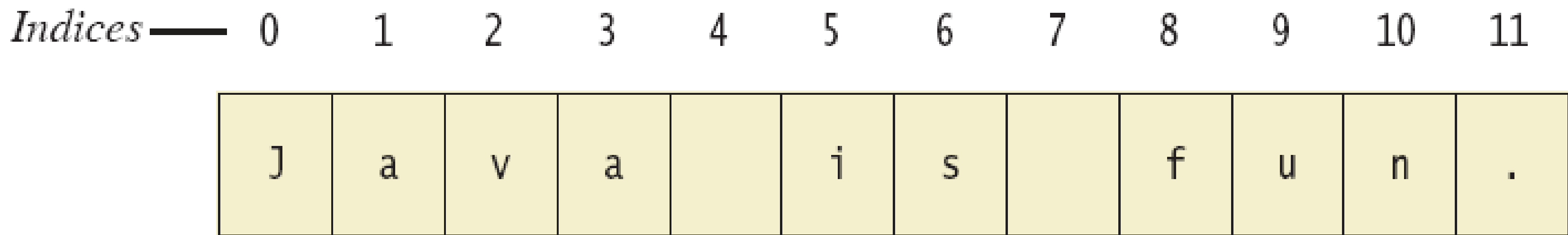
- An object of the `String` class stores data consisting of a sequence of characters.
- Objects have methods as well as data
  - Recall a data type is a set of values and the operations allowed on those values
    - For numeric data types, these are mathematical operations
    - For class data types, these operations are its methods

The Method `length()`

- The method **`length()`** returns an **`int`**.
- You can use a call to method **`length()`** anywhere an **`int`** can be used.

```
int count = command.length();
System.out.println("Length is " +
    command.length());
count = command.length() + 3;
```

`String` Indices



- Positions start with 0, not 1.
  - The 'J' in "Java is fun." is in position 0
- A position is referred to an *index.*
  - The **`'f'`** in **`"Java is fun."`** is at index 8.

Data Types: Boolean / Logical

- And: &&, &
  - If both arguments are true, result is true, otherwise false
- Or: ||, |
  - If any argument is true, result is true, otherwise false
- Not: !
  - If argument is true, result is false and vice versa
- Boolean variables should suggest a true/false value
  - Choose names such as **isPositive** or **systemsAreOk**.
  - Avoid names such as **numberSign** or **systemStatus**.

Java built-in type	what it stores	example values	operations
`boolean`	truth values	`true` `false`	and, or, not

Booleans

- `boolean` data type

logical AND	logical OR	logical NOT
&&	||	!

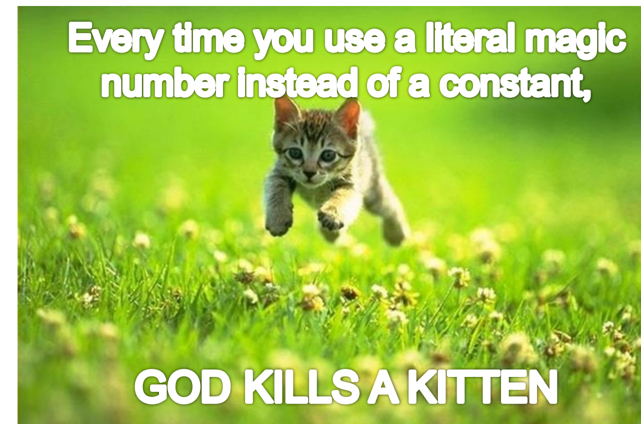
`!a` → "Is a set to `false`?"
`a && b` → "Are both a *and* b set to `true`?"
`a || b` → "Is either a *or* b (or both) set to `true`?"

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

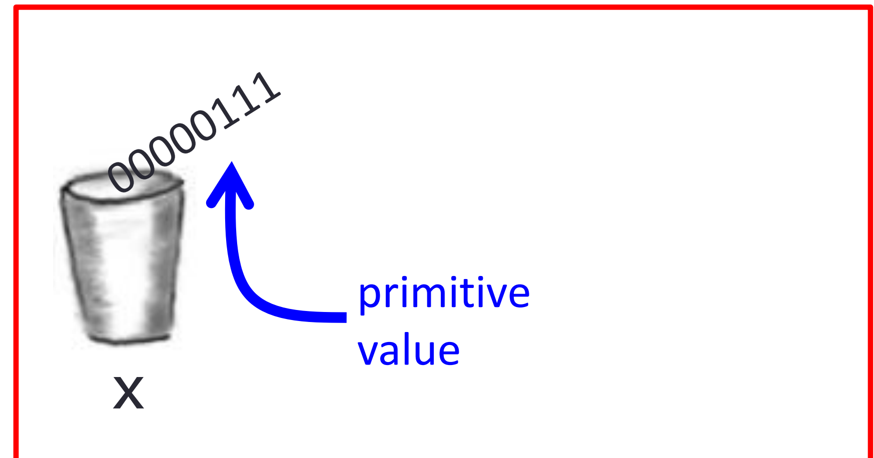
Data Types: Constants

- Literal expressions such as `42, 3.7`, or `'y'` are called *constants*.
- Integer constants can be preceded by a **+** or **–** sign, but cannot contain commas.
- Floating-point constants can be written
  - With digits after a decimal point or
  - Using *e notation.*
- Constants
  - All upper case, use _ between words
  - **double** SPEED_LIGHT = 3.0e8;



Creating and Initializing a Primitive Variable

```
byte x = 7;
```

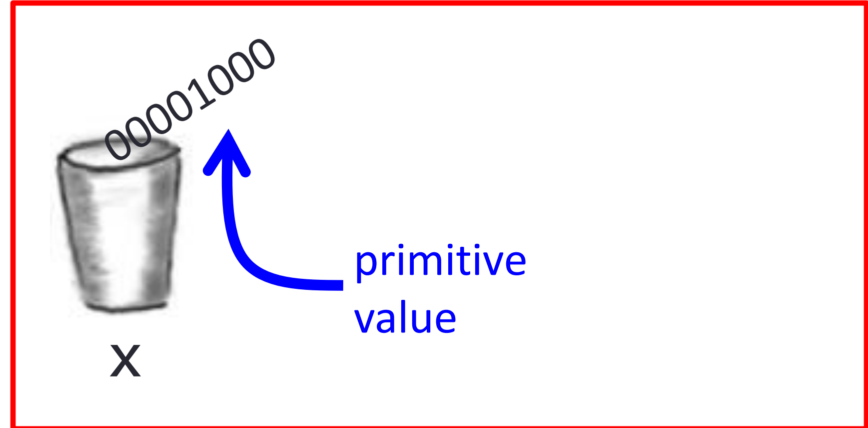


x

01101010101010010101010111010101010101010101011101010 1
00010101010101111101010101010101010101001001101010101
01001010101011000000111010101010101110101010001010101 0
1010111101010101010101010101010010011010101010101001010101
0111010101010101010101010111010101000101010101010111010
1010101010101010100100110101010101001010101011101010 10
1010101010101110101010001010101010101111010101010101 01
0101010010011010101010101001010101011101010101010101 01

Changing the Value of a Primitive Variable

```
byte x = 7;
x = x + 1;
```



x

01101010101010010101010111010101010101010101011101010 1
00010101010101011111010101010101010101010010011010101 01
01001010101011000001000010101010101110101010001010101 0
10101111010101010101010101010100100110101010101001010101
01110101010101010101010101110101010001010101010101111010
10101010101010101001001101010101010100101010101110101010
10101010101011101010100010101010101011110101010101010101
01010100100110101010101001010101011101010101010101010101

Mathematical Expressions: Parentheses and Precedence

- Parentheses can communicate the order in which arithmetic operations are performed
- examples:

```
(cost + tax) * discount
(cost + (tax * discount)
```

- Without parentheses, an expressions is evaluated according to the *rules of precedence.*

Precedence Rules

- When binary operators have equal precedence, the operator on the left acts before the operator(s) on the right.

Highest Precedence

First: the unary operators +, -, !, ++, and --

Second: the binary arithmetic operators *, /, and %

Third: the binary arithmetic operators + and –

Lowest Precedence

Boolean Expressions: Comparisons

- Given two numbers → return a `boolean`

operator	meaning	true example	false example
`==`	equal	`7 == 7`	`7 == 8`
`!=`	not equal	`7 != 8`	`7 != 7`
`<`	less than	`7 < 8`	`8 < 7`
`<=`	less than or equal	`7 <= 7`	`8 <= 7`
`>`	greater than	`8 > 7`	`7 > 8`
`>=`	greater than or equal	`8 >= 2`	`8 >= 10`

Is the sum of a, b and c equal to 0?
Is grade in the B range?
Is sumItems an even number?

```
(a + b + c) == 0
(grade >= 80.0) && (grade < 90.0)
(sumItems % 2) == 0
```

Leap Year Example

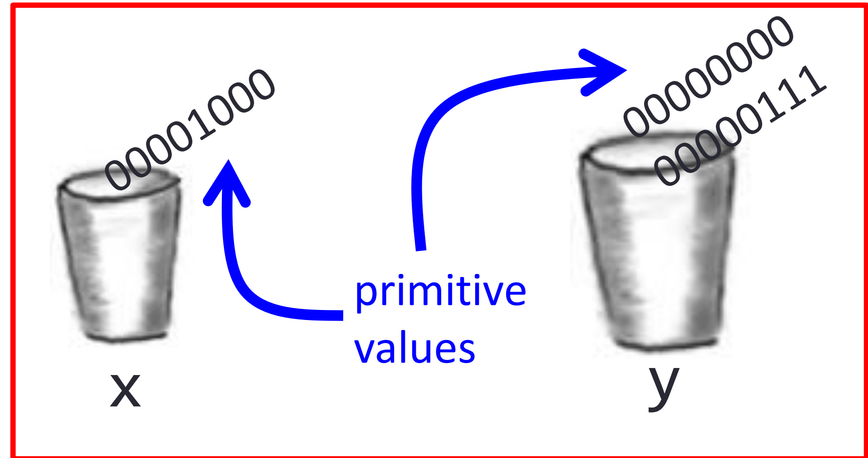
- Years divisible by 4 but not by 100 → leap year
- Years divisible by 400 → leap year

```
public class LeapYear
{
   public static void main(String [] args)
   {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;

      // Leap year if divisible by 4 but not by 100
      isLeapYear = (year % 4 == 0) && (year % 100 != 0);

      // But also leap year if divisible by 400
      isLeapYear = isLeapYear || (year % 400 == 0);
      System.out.println(isLeapYear);
   }
}
```

```
% java LeapYear 2000
true
```

Creating and Initializing a Primitive Variable

```java
byte x = 7;
x = x + 1;

short y = 7;
```



x

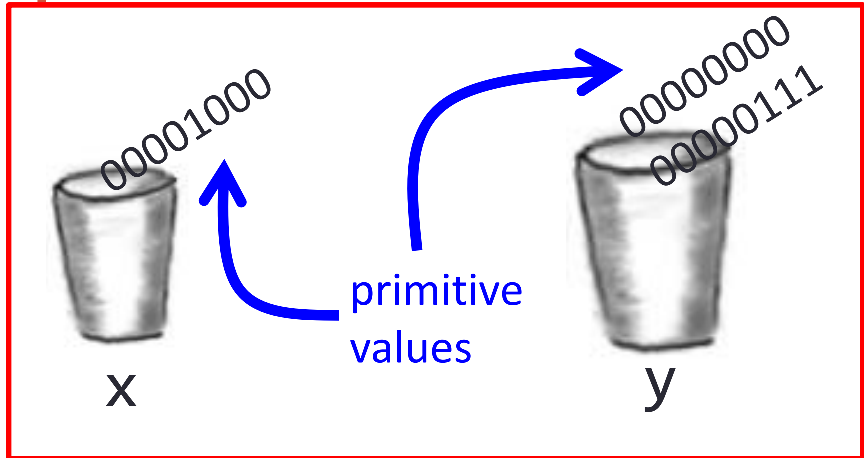
y

0110101010101001010101011101010101010101010101110101 01
0001010101010101111010101010101010101010010011010101 01
0100101010101011000001000010101010101110101010001010101 0
1010111101010101010101010101001001101010101010100101010 1
0111010000000000000111011101010100010101010101011101 0
1010101010101010100100110101010101001010101011101010 10
1010101010101110101010001010101010101111010101010101 01
0101010010011010101010100101010101110101010101010101 01

You can't put a big cup into a small one

You may know 7 can fit in a byte, but compiler doesn't!

```
byte x = 7;
x = x + 1;

short y = 7;

x = y;
```



x

y

```
0110101010101001010101011101010101010101010111010101
0001010101010101111010101010101010101001001101010101
0100101010101110000010000101010101011101010100010101010
101011110101010101010101010101001001101010101010010101 01
011101000000000000001110111010101000101010101010111010
101010101010101010010011010101010100101010101110101010
1010101010101110101010001010101010111010101010101010 1
010101001001101010101010010101010111010101010101010101
```

Assignment Compatibilities

- Java is said to be *strongly typed.*
  - You can't, for example, assign a floating point value to a variable declared to store an integer.
- Sometimes conversions between numbers are possible.

  `doubleVariable = 7;`

  is possible even if **`doubleVariable`** is of type **`double`**, for example.

Assignment Compatibilities

- A value of one type can be assigned to a variable of any type further to the right

  `byte --> short --> int --> long`

  `--> float --> double`

  - But not to a variable of any type further to the left.
  - This is called "automatic type conversion"
- You can assign a value of type char to a variable of type **`int`**.

Type Casting

- A *type cast* temporarily changes the value of a variable from the declared type to some other type.
- For example,

```
double distance;
distance = 9.0;
int points;
points = (int)distance;
```

  - Illegal without **`(int)`**
  - The value of **`(int)distance`** is **`9`**,
  - The value of **`distance`**, both before and after the cast, is **`9.0`**.
- Any nonzero value to the right of the decimal point is ***truncated*** rather than *rounded.*

Type Conversion

- Java is strongly typed
  - Helps protect you from mistakes (aka "bugs")

```
public class TypeExample0
{
   public static void main(String [] args)
   {
      int orderTotal = 0;
      double costItem = 29.95;

      orderTotal = costItem * 1.06;
      System.out.println("total=" + orderTotal);
   }
}
```

```
% javac TypeExample0.java
TypeExample0.java:7: possible loss of precision
found    : double
required: int
      orderTotal = costItem * 1.06;
                              ^
```

Type Conversion

- Converting from one type to another:
  - Manually → using a cast
    - A cast is accomplished by putting a type inside ()'s
  - Casting to `int` drops fractional part
    - Does not round!

```
public class TypeExample1
{
   public static void main(String [] args)
   {
      int orderTotal = 0;
      double costItem = 29.95;

      orderTotal = (int) (costItem * 1.06);

      System.out.println("total=" + orderTotal);
   }
}
```

```
% java TypeExample1
total=31
```

Type Conversion

- Automatic conversion
  - Numeric types:
    - If no loss of precision → automatic promotion

```
public class TypeExample2
{
   public static void main(String [] args)
   {
      double orderTotal = 0.0;
      int costItem = 30;

      orderTotal = costItem * 1.06;

      System.out.println("total=" + orderTotal);
   }
}
```

```
% java TypeExample2
total=31.8
```

Type Conversion

- Automatic conversion
  - `String` concatenation using the + operator converts numeric types to also be a `String`

```
public class TypeExample3
{
   public static void main(String [] args)
   {
      double costItem = 29.95;

      String message = "The widget costs ";
      message = message + costItem;
      message = message + "!";

      System.out.println(message);
   }
}
```

```
% java TypeExample3
The widget costs 29.95!
```

Static Methods

- Java has lots of helper methods
  - Things that take value(s) and return a result
    - e.g. Math functions
    - e.g. Type conversion: `String` → `int`
      `String` → `double`
    - e.g. Random number generation
- For now, we'll stick to `static` methods
  - Live in some particular Java class
    - e.g. `Math`, `Integer` or `Double`
  - Call using class name followed by dot

Type Conversion Quiz

- Automatic: no loss of precision
  - `int` will convert to a `double` if need be
  - `double` cannot automatically convert to `int`
- Manual: cast or using a static method

expression	resulting type	resulting value
`(int) 3.14159`		
`Math.round(3.6)`		
`2 * 3.0`		
`2 * (int) 3.0`		
`(int) 2 * 3.0`		

Type Conversion Quiz

- Automatic: no loss of precision
  - `int` will convert to a `double` if need be
  - `double` cannot automatically convert to `int`
- Manual: cast or using a method

expression	resulting type	resulting value
`(int) 3.14159`	`int`	`3`
`Math.round(3.6)`	`long`	`4`
`2 * 3.0`	`double`	`6.0`
`2 * (int) 3.0`	`int`	`6`
`(int) 2 * 3.0`	`double`	`6.0`

String Conversion Quiz

- `String` conversion, using:
  - `Integer.parseInt()`
  - `Double.parseDouble()`

expression	resulting type	resulting value
`Integer.parseInt("30")`		
`Double.parseDouble("30")`		
`Integer.parseInt("30.1")`		
`Double.parseDouble("30.1")`		
`Integer.parseInt("$30")`		
`Double.parseDouble(3.14)`		

String Conversion Quiz

- `String` conversion, using:
  - `Integer.parseInt()`
  - `Double.parseDouble()`

expression	resulting type	resulting value
`Integer.parseInt("30")`	`int`	`30`
`Double.parseDouble("30")`	`double`	`30.0`
`Integer.parseInt("30.1")`	`(runtime error, can't parse as int)`	
`Double.parseDouble("30.1")`	`double`	`30.1`
`Integer.parseInt("$30")`	`(runtime error, can't parse as int)`	
`Double.parseDouble(3.14)`	`(compile error, 3.14 not a String)`	

String Concatenation Quiz

- + is addition for numeric types
- + is concatenation for `String` type
- numeric types convert to `String` if needed
  - Strings never (automatically) go back to number

expression	resulting type	resulting value
`"testing " + 1 + 2 + 3`		
`"3.1" + 4159`		
`"2" + " + " + "3"`		
`1 + 2 + 3 + "66"`		

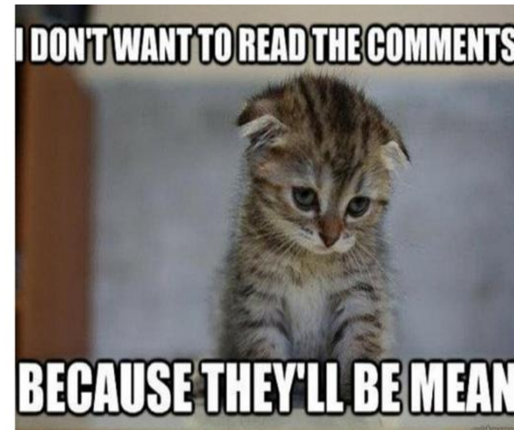
String Concatenation Quiz

- + is addition for numeric types
- + is concatenation for `String` type
- numeric types convert to `String` if needed
  - Strings never (automatically) go back to number

expression	resulting type	resulting value
`"testing " + 1 + 2 + 3`	`String`	`"testing 123"`
`"3.1" + 4159`	`String`	`"3.14159"`
`"2" + " + " + "3"`	`String`	`"2 + 3"`
`1 + 2 + 3 + "66"`	`String`	`"666"`

Comments

- The best programs are self-documenting.
  - Clean style
  - Well-chosen names
- Comments are written into a program as needed to explain the program.
  - They are useful to the programmer, but they are ignored by the compiler.
- // comment to end of line
- /*
  multi-line comment
- */
- /**
- * javadoc comment
- */



Summary

- Variables
  - Naming Conventions
- Data Types
  - Primitive Data Types
    - Review: int, double
    - New: boolean, char
  - The `String` Class
  - Type Conversion
- Expressions
  - Assignment
  - Mathematical
  - Boolean
- Sequential Execution