



西北工业大学  
NORTHWESTERN POLYTECHNICAL UNIVERSITY

# Object-Oriented Programming

---

## Chapter 3 Fundamental Programming Structures in Java

---

**Dr. Helei Cui**

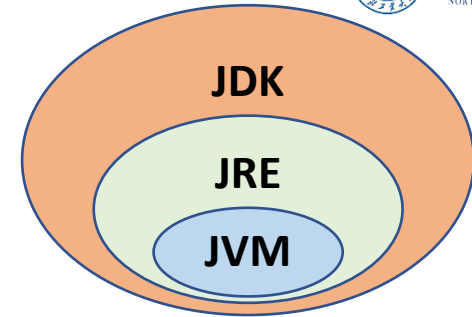
19 Mar 2021

*Slides partially adapted from lecture  
notes by Cay Horstmann*

# Recap

## ➤ JDK, JRE, JVM?

- JDK is a software development kit
- JRE is a software bundle that allows Java program to run
- JVM is an environment for executing bytecode



## ➤ Run HelloWorld.java

- Using Command-Line Tools
- Using an IDE, e.g., Eclipse

```

Select Command Prompt

Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\1x>cd D:\oop\ch02

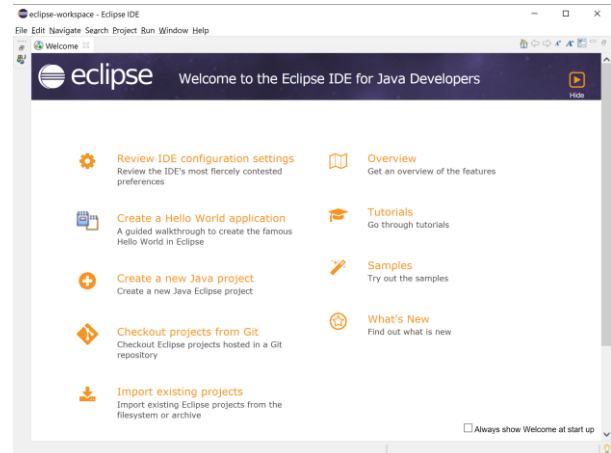
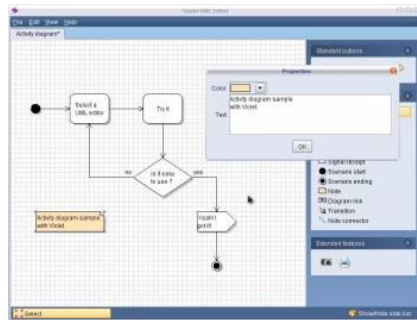
C:\Users\1x>D:

D:\oop\ch02>javac HelloWorld.java

D:\oop\ch02>java HelloWorld
Hello World!

D:\oop\ch02>
  
```

## ➤ Use Violet UML editor



# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

# FirstSample.java

```
public class FirstSample {  
    public static void main(String[] args) {  
        System.out.println("We will not use 'Hello, World!'");  
    }  
}
```

- **“public”**: access modifier
  - Define the accessibility of a class.
- **“class”**: “a container for the program logic”
  - Everything in a Java program must be inside a class.
- **“FirstSample”**: name of the class
  - The file name for the source code must be **the same** as the name of the public class, with **“.java”** appended.

# Rules for class names in Java

1. Names must **begin with a letter**, and after that, they can have **any combination of letters and digits**.
2. The length is essentially **unlimited**.
3. Do not use a **Java reserved word** for a class name.
  - *E.g., public, class, static, void, etc.*
4. The standard naming convention:
  - Class names are **nouns** that start with an uppercase letter.
  - **Camel Case**: If a name consists of multiple words, use an initial uppercase letter in each of the words.
  - *E.g., HelloWorld, FirstSample.*

# The braces { }

```
public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println("We will not use 'Hello, World!'");
    }
}
```

```
public class FirstSample {
    public static void main(String[] args) {
        System.out.println("We will not use 'Hello, World!'");
    }
}
```

# FirstSample.java

```
public class FirstSample {  
    public static void main(String[] args) {  
        System.out.println("We will not use 'Hello, World!'");  
    }  
}
```

- The body of the **main** method contains a statement that outputs a single line of text to the console.
  - Here, we are using the **System.out** object and calling its **println** method.
  - The periods (".") are used to invoke a method.
  - A method can have zero, one or more parameters (arguments).
  - Parentheses are always needed even there is no parameters.
    - E.g., **System.out.println();**

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays



# Comments

- Three types:

1. `//` Single-line comments

2. `/*` Multi-line Comments

The second line of this comment `*/`

3. `/**`

`* This is used to generate documentation automatically`

`* @version 1.0 2021-03-19`

`* @author Harry Cui`

`*/`

- Comments can be used to explain Java code, and to make it more readable.
- Comments do not show up in the executable program.

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

# Java is strongly typed

- All variable must be declared.
  - `<type> <variable>;`
  - *E.g., `int x;`*
- After a variable is declared, you can assign to it.
  - *E.g., `x = 4;`*
- We call a variable which has a class for a type an object.
  - *E.g., `Car c;`*
- Once an object is declared, you can
  - assign to it, often with a creation statement,
  - access its data members, and
  - call its methods.
  - *E.g., `c = new Car("BMW"); c.make = "Audi"; c.getMake();`*

## 3.3.1 Integer types

- For numbers without fractional parts.

Type	Storage	Range (Inclusive)
<code>int</code>	4 bytes	-2,147,483,648 to 2,147,483,647 (just over 2 billion)
<code>short</code>	2 bytes	-32,768 to 32,767
<code>long</code>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>byte</code>	1 byte	-128 to 127

- In Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code.

# Literals in Java

- A literal is a source code representation of a fixed value.
  - They are represented directly in the code without any computation.
  - Literals can be assigned to any primitive type variable.
  - **byte**, **int**, **long**, and **short** can be expressed in *decimal* (base 10), *hexadecimal* (base 16), *octal* (base 8) or *binary* (base 2) number systems.

```
byte a      = 65;  
int  decimal = 100;  
long num    = 100L;    // with suffix L or l  
int  octal   = 0144;    // with prefix 0  
int  hex     = 0x64;    // with prefix 0x or 0X  
int  bin     = 0b1100100; // with prefix 0b or 0B
```

**Data type**

**Literal**

## 3.3.2 Floating-point types

- For numbers with fractional parts.

Type	Storage	Range
<b>float</b>	4 bytes	Approximately $\pm 3.40282347\text{E}+38\text{F}$ (6-7 significant decimal digits)
<b>double</b>	8 bytes	Approximately $\pm 1.79769313486231570\text{E}+308$ (15 significant decimal digits)

- The name double refers to the fact that these numbers have twice the precision of the float type.

```
float  fNum = 3.14F; // with suffix F or f
double dNum = 3.14D; // with suffix D or d (optionally)
```

# Caution: round-off error

```
System.out.println(2.0 - 1.1); // result: 0.8999999999999999 not 0.9
```

- Reason:
  - **Floating-point numbers are represented in the binary number system.** There is no precise binary representation of the fraction  $1/10$ , just as there is no accurate representation of the fraction  $1/3$  in the decimal system.
- Solution:
  - Using the **BigDecimal** class if you need precise numerical computations.

```
System.out.println(BigDecimal.valueOf(2.0).subtract(BigDecimal.valueOf(1.1)));  
// result: 0.9
```

## 3.3.3 The char type

- Used for describing individual characters.

```
char ch = 'A';  
char tm = '\u2122'; // Unicode for the trademark symbol (™)
```

- 'A' is a character constant with a value of 65.
- "A" is a string containing a single character.

```
char ch1 = 'A';  
char ch2 = '\u0041'; // Unicode for the character A
```



# Escape sequences for special characters

- Escape sequence is a character preceded by a backslash (\) and has a special meaning to the compiler.

Escape Sequence	Name	Unicode Value
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000a</code>
<code>\r</code>	Carriage return	<code>\u000d</code>
<code>\"</code>	Double quote	<code>\u0022</code>
<code>\'</code>	Single quote	<code>\u0027</code>
<code>\\</code>	Backslash	<code>\u005c</code>

**Java backspace escape doesn't work?**

<https://stackoverflow.com/questions/3328824/java-backspace-escape>

```
System.out.println("She said \"Hello!\" to me.");  
// output: She said "Hello!" to me.
```

## 3.3.4 Unicode and the char type

- Unicode is an information technology standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
  - It was invented to overcome the limitations of traditional character encoding schemes.
  - Before Unicode, there were many different standards: ASCII in the United States, BIG-5 for Chinese, etc.
  - *See more on “Unicode Encoding! UTF-32, UCS-2, UTF-16, & UTF-8!”*  
<https://www.youtube.com/watch?v=uTJoJtNYcaQ>

Strong recommendation:

Not to use the char type in your programs unless you are actually manipulating UTF-16 code units. You are almost always better off treating strings as abstract data types.

# ASCII vs Unicode in Java (13 min)

## STANDARDS: ASCII vs UNICODE

```
public class DataTypes101
{
    public static void main(String[ ] args)
    {
        int    variable1;
        double variable2;
        char   variable3;
    }
}
```

**Output:**

NO Output

<https://www.youtube.com/watch?v=61Bs7-ycL64>

## 3.3.5 The boolean type

- Used for evaluating logical conditions.
  - Only two values: **true** or **false**
  - No conversion between integers and Boolean values.

```
boolean isJavaFun    = true;
boolean isJavaBoring = false;
System.out.println(isJavaFun);    // Outputs true
System.out.println(isJavaBoring); // Outputs false

// Boolean Expression
int x = 10;
int y = 9;
System.out.println(x > y);        // Outputs true
```

# Primitive vs reference

- Types in Java are divided into two categories - primitive types and reference types.
  - Primitive types: **boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**.
  - All other types are reference types, so classes, which specify the types of objects, are reference types.
- A primitive-type variable can store exactly one value of its declared type at a time.
  - Primitive-type instance variables are initialized by default.
    - Variables of types **byte**, **char**, **short**, **int**, **long**, **float** and **double** are initialized to **0**.
    - Variables of type **boolean** are initialized to **false**.
- Reference-type variables (called references) store the location (address) of an object in the computer's memory.
  - Such variables refer to objects.



# Primitive and Reference Types in Memory (5 min)

## Memory Handling in Java



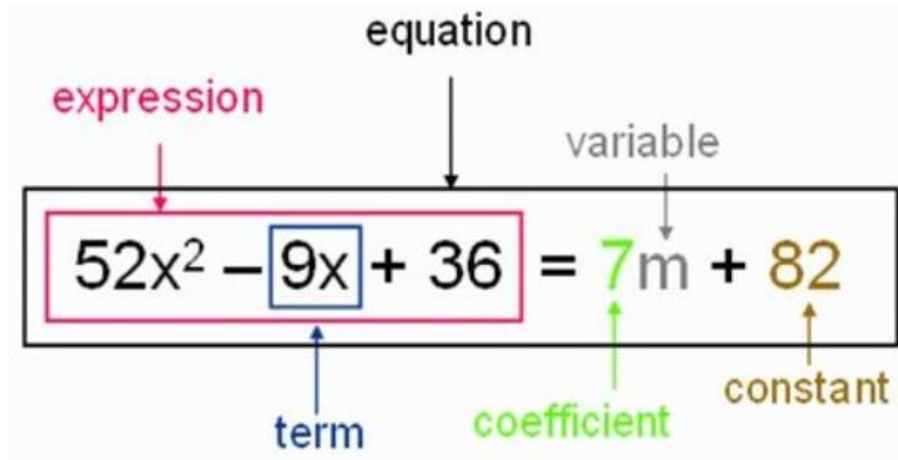
<https://www.youtube.com/watch?v=LTnp79Ke8FI>

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

# Variables vs Constants

- A **constant** is a data item whose value cannot change during the program's execution.
  - Thus, as its name implies - the value is constant.
- A **variable** is a data item whose value can change during the program's execution.
  - Thus, as its name implies - the value can vary.



<https://byjus.com/maths/variables-and-constants-in-algebraic-expressions/>



## 3.4.1 Declaring variables

- General form: **type** **variableName**;

```
double salary;  
long earthPopulation;  
boolean done;  
int i, j;           // correct but not recommended
```

- A variable name must *begin with a letter* and must be a sequence of letters or digits.
  - **letter**: 'A'–'Z', 'a'–'z', '\_', '\$', or any Unicode character that denotes a letter in a language.
  - **digit**: '0'–'9' and any Unicode characters that denote a digit in a language.
  - Symbols like '+' or '@' cannot be used inside variable names, nor can spaces.
  - **Case-sensitive**, e.g., *"aNum"* and *"ANum"* are different.

## 3.4.2 Initializing variables

- You must **explicitly initialize** it by means of an assignment statement.
  - You can never use the value of an uninitialized variable.
  - Otherwise, you would see an **ERROR**, “*variable not initialized*”.
- Using an equal sign =

```
int vacationDays;  
vacationDays = 12;  
int vacationDays = 12; // correct but not recommended
```

- Good style: declare variables as closely as possible to the point where they are first used.

# Local type inference in Java 10

- Can use **var** instead type for local variables:

```
var counter = 0; // an int  
var message = "Greetings, earthlings!"; // a String
```

- Still strongly typed:

```
Counter = 0.5; // Error: can't assign a double to an int
```

- Useful for unwieldy type names:

```
var traces = Thread.getAllStackTraces(); // a Map<Thread,  
StackTraceElement[]>
```

## 3.4.3 Constants

- Using **final** to denote a constant.

```
final double PI = 3.14;  
final int VACATION_DAYS = 12;
```

- Good style: name it in all uppercase with words separated by underscores ("\_").

# Class constants

- Using **static final** to create a constant so it's available to multiple methods inside a single class.

```
public class Constants2 {  
    public static final double CM_PER_INCH = 2.54;  
    public static void main(String[] args) {  
        double paperWidth = 8.5;  
        System.out.println("Paper width in centimeters: "  
            + paperWidth * CM_PER_INCH);  
    }  
}
```

- If the constant is declared **public**, other classes can use it like *Constants2.CM\_PER\_INCH*.

## 3.4.4 Enumerated types

- To describe a variable that only hold a restricted set of values.
  - E.g., sizes of pizza: small, medium, large, and extra large.
  - Only a finite number of named values.

```
public class Example {  
    enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };  
    public static void main(String[] args) {  
        // Now you can declare variables of this type:  
        Size s = Size.MEDIUM;  
        System.out.println(s); // Outputs: MEDIUM  
    }  
}
```

- Variable of type **s** can only hold **size** values or **null**.

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

## 3.5.1 Arithmetic operators

- The usual arithmetic operators:
  - Addition  $+$
  - Subtraction  $-$
  - Multiplication  $*$
  - Division  $/$ 
    - Integer division if both arguments are integers, and floating-point division otherwise.
    - *E.g.,  $15 / 2$  is 7,  $15.0 / 2$  is 7.5.*
  - Integer remainder (a.k.a., modulus)  $\%$ 
    - *E.g.,  $15 \% 2$  is 1.*



# Quick question 1

$$-7 \% 3 = ?$$

- A. 1
- B. -1
- C. -2

$$7 \% -3 = ?$$

- A. 1
- B. -1
- C. -2

# % in Java

$$7 \% 3 = 1$$

$$-7 \% 3 = ?$$

$$7 \% -3 = ?$$

- Using the formula  **$a \% b = a - a / b * b$** , you can get

$$7 \% 3 = 7 - 7 / 3 * 3 = 7 - 2 * 3 = 1$$

$$-7 \% 3 = -7 - (-7) / 3 * 3 = -7 - (-2) * 3 = -1$$

$$7 \% -3 = 7 - 7 / (-3) * (-3) = 7 - (-2) * (-3) = 1$$



# Division /

- In Python, the integer division uses “**floor function**”:

$$-7 \% 3 = -7 - \text{floor}(-7 / 3) * 3 = -7 - (-3) * 3 = -7 + 9 = 2$$

$$7 \% (-3) = 7 - \text{floor}(7 / (-3)) * (-3) = 7 - (-3) * (-3) = 7 - 9 = -2$$

*The floor function takes as input a real number  $x$ , and gives as output the greatest integer less than or equal to  $x$ , denoted  $\text{floor}(x)$ .*

- In Java or C, the integer division uses “**truncation**”:

$$-7 \% 3 = -7 - \text{trunc}(-7 / 3) * 3 = -7 - (-2) * 3 = -7 + 6 = -1$$

$$7 \% (-3) = 7 - \text{trunc}(7 / (-3)) * (-3) = 7 - (-2) * (-3) = 7 - 6 = 1$$

*For positive real numbers, truncation is done using the floor function. But for negative numbers, truncation always rounds towards zero.*

# Quick question 2

How to get the ones, tens, hundreds, and thousands digit in the number 1,234?

$x / 1000$  ---> thousands digit

$x \% 10$  ---> ones digit

$x / 10 \% 10$  ---> tens digit

$x / 100 \% 10$  ---> hundreds digit

## 3.5.2 Mathematical functions and constants

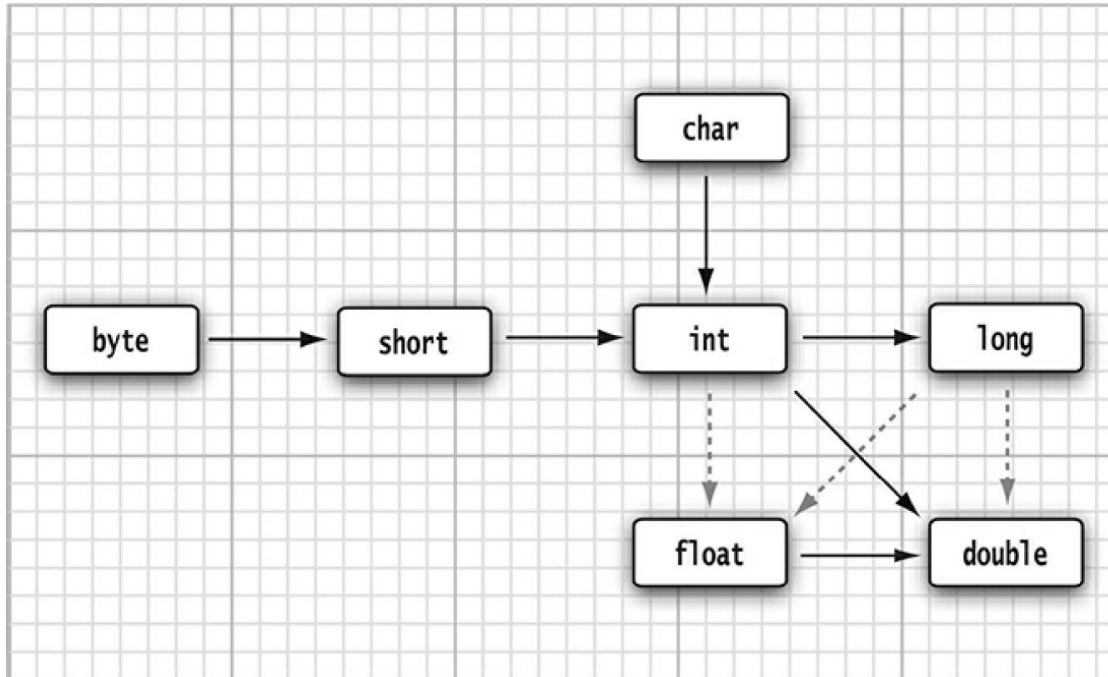
- The **Math** class contains methods for performing basic numeric operations and constants  $\pi$  and **e**.
  - E.g., the elementary exponential, logarithm, square root, and trigonometric functions.

```
double x = 4;  
double y = Math.sqrt(x);  
System.out.println(y);           // prints 2.0  
System.out.println(Math.PI);    // prints 3.141592653589793  
System.out.println(Math.E);     // prints 2.718281828459045
```

The `println` method operates on **the `System.out` object**. But the `sqrt` method in the `Math` class does not operate on any object, which is called a **static method**.

## 3.5.3 Conversions between numeric types

- These conversions are automatic:



Dotted arrows indicate possible precision loss.

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

# Rules

- When two values are combined with a binary operator (such as `n + f` where `n` is an integer and `f` is a floating-point value), both operands are converted to a common type before the operation is carried out.
  - If either of the operands is of type `double`, the other one will be converted to a `double`.
  - Otherwise, if either of the operands is of type `float`, the other one will be converted to a `float`.
  - Otherwise, if either of the operands is of type `long`, the other one will be converted to a `long`.
  - Otherwise, both operands will be converted to an `int`.

## 3.5.4 Casts

- Conversions in which loss of information is possible are done by means of casts.
  - The syntax for casting is to give the target type in parentheses, followed by the variable name.

```
double x = 9.997;  
int nx = (int) x; // nx is 9
```

- Use the **Math.round** method to round a floating-point number to the nearest integer.

```
double x = 9.997;  
int nx = (int) Math.round(x); // nx is 10
```

<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Math.html>



# Caution

- If you try to cast a number of one type to another that is **out of range** for the target type, the result will be a truncated number that has a different value.

```
byte x = (byte) 300;  
System.out.println(x); // Outputs 44
```

- The number 300 in binary form is **100101100**, then byte type only gets 8 digits. So, the x only gets **00101100**, i.e., 44.

## 3.5.5 Combining assignment with operators

- “**x += 4;**” is equivalent to “**x = x + 4;**”
- Also **-=**, **\*=**, **/=**, **%=**, and so on.
- Note:
  - If the operator yields a value whose type is different from that of the left-hand side, then it is enforced to fit.

```
int x = 1;  
x += 3.5;           // Equivalent to x = (int)(x + 3.5)  
System.out.println(x); // Outputs 4
```

## 3.5.6 Increment and decrement operators

- **++** and **--**
  - Postfix form: **m++** adds 1 to the current value of the variable **m**, and **m--** subtracts 1 from it.
  - Prefix form: **++m** and **--m**.

```
int m = 4, n;  
n = ++m; // m = m + 1; n = m; --> n and m are 5  
n = --m; // m = m - 1; n = m; --> n and m are 3  
n = m++; // n = m; m = m + 1; --> n is 4, m is 5  
n = m--; // n = m; m = m - 1; --> n is 4, m is 3
```

## Quick question 3

**int m = 5; System.out.println(m++);**

- A. Outputs 5
- B. Outputs 6
- C. Outputs 7

```
int m = 5;  
m++;  
System.out.println(m);
```

**int m = 5; System.out.println(--m);**

- A. Outputs 3
- B. Outputs 4
- C. Outputs 5

```
int m = 5;  
--m;  
System.out.println(m);
```

We recommend against using ++ (or --) **inside expressions** because this **often leads to confusing code and annoying bugs.**

## 3.5.7 Relational and boolean Operators

- Relational operators:
  - `==` (equality test) `!=` (inequality test)
  - `<` (less than) `<=` (less than or equal)
  - `>` (greater than) `>=` (greater than or equal)
- Boolean operators:
  - `&&` (logical and) `||` (logical or) `!` (logical not)
  - *E.g., `expression1 && expression2`*
  - If the truth value of the first expression has been determined to be false, then it is impossible for the result to be true. Thus, the value for the second expression is not calculated.

```
x != 0 && 1 / x > x + y
// The second part is never evaluated if x equals zero.
// Thus, 1/x is not computed if x is zero, and no divide-by-zero
// error can occur.
```

# Truth table

- A truth table is a mathematical table used in logic.

*NOT*

<i>x</i>	<i>!x</i>
0	1
1	0

*AND*

<i>x</i>	<i>y</i>	<i>x&amp;&amp;y</i>
0	0	0
0	1	0
1	0	0
1	1	1

*OR*

<i>x</i>	<i>y</i>	<i>x  y</i>
0	0	0
0	1	1
1	0	1
1	1	1

# The ternary ?: operator

- **condition ? expression1 : expression2**
- It evaluates to the first expression if the condition is true, to the second expression otherwise.

```
int a = 10;  
int b = 20;  
  
int max = a >= b ? a : b;
```

## 3.5.8 Bitwise Operators

- `&` (bitwise and) `|` (bitwise or) `^` (bitwise xor) `~` (bitwise compliment)
- These operators work on bit patterns.
  - E.g., *`int fourthBitFromRight = (n & 0b1000) / 0b1000;`*
  - It gives you a 1 if the fourth bit from the right in the binary representation of `n` is 1, and 0 otherwise.
  - Using `&` with the appropriate power of 2 lets you mask out all but a single bit.
- When applied to boolean values, the two bitwise operators (`&` and `|`) also returns a boolean value.
  - These operators are like the logical operators (`&&` and `||`), except that the bitwise operators (`&` and `|`) are not evaluated in “short circuit” fashion - that is, both arguments are evaluated before the result is computed.



## 3.5.8 Bitwise Operators

- $\gg$  (right shift)  $\ll$  (left shift)
- They shift a bit pattern right or left.
  - Assuming  $A$  is 60 (0011 1100), then
    - $A \ll 2$  will give 240 which is 1111 0000
    - $A \gg 2$  will give 15 which is 1111
  - E.g.,  $\text{int fourthBitFromRight} = (n \& (1 \ll 3)) \gg 3;$
- $\ggg$  (zero fill right shift)
- It fills the top bits with zero, unlike  $\gg$  which extends the sign bit into the top bits.
  - Assuming  $A$  is 60 (0011 1100), then
    - $A \ggg 2$  will give 15 which is 0000 1111

# Quick question 4

```
int A = 60;
int B = A >> 2;
int C = A >>> 2;

System.out.println("A = " + A + " in binary: " +
Integer.toBinaryString(A));

System.out.println("B = " + B + " in binary: " +
Integer.toBinaryString(B));

System.out.println("C = " + C + " in binary: " +
Integer.toBinaryString(C));

// output
A = 60 in binary: 111100
B = 15 in binary: 1111
C = 15 in binary: 1111
```

# Quick question 5

```
int A = -60;
int B = A >> 2;
int C = A >>> 2;

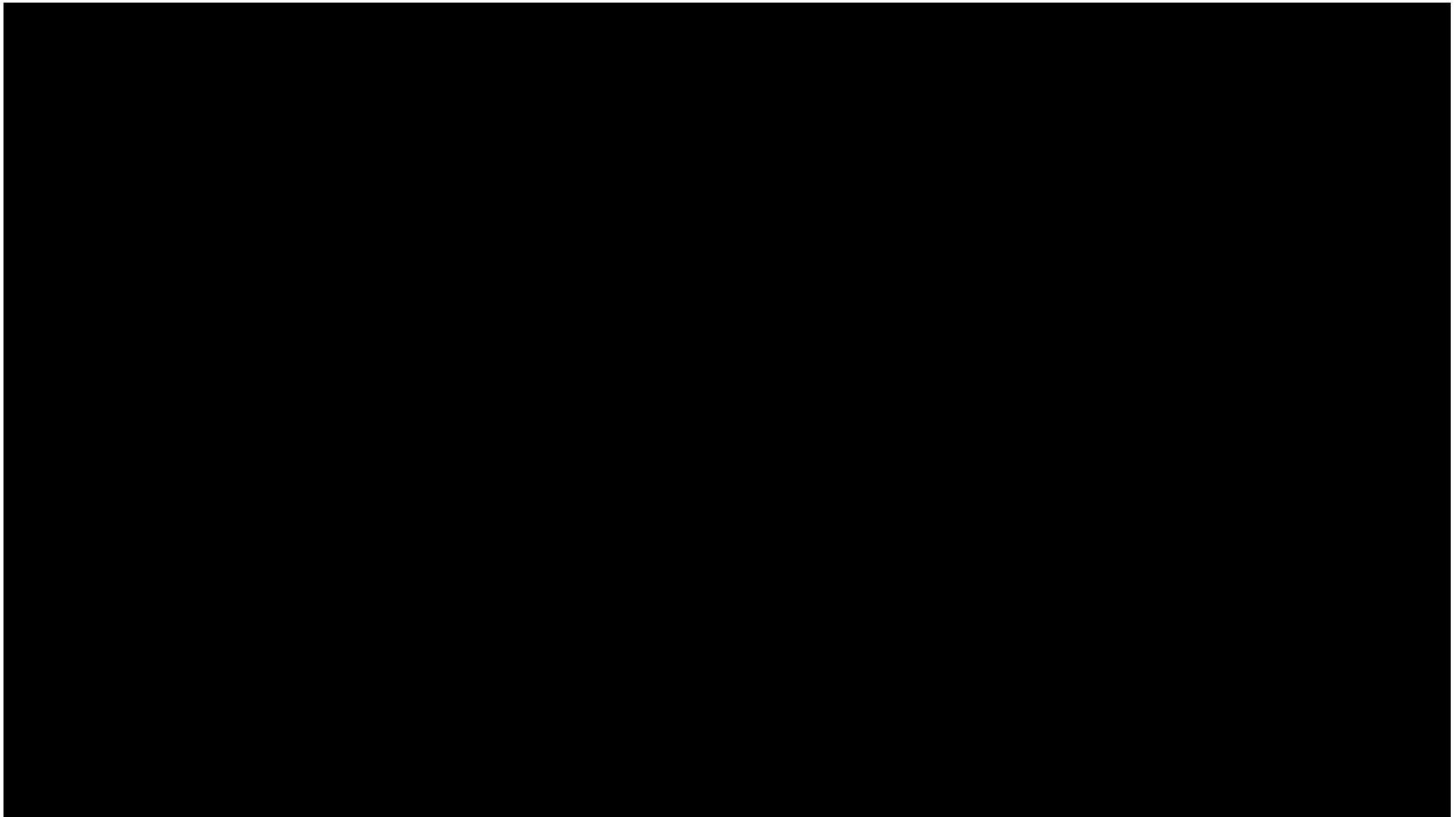
System.out.println("A = " + A + " in binary: " +
Integer.toBinaryString(A));

System.out.println("B = " + B + " in binary: " +
Integer.toBinaryString(B));

System.out.println("C = " + C + " in binary: " +
Integer.toBinaryString(C));

// output
A = -60 in binary: 11111111111111111111111111000100
B = -15 in binary: 1111111111111111111111111110001
C = 1073741809 in binary: 1111111111111111111111111110001
```

# Two's complement (4 min)



<https://www.youtube.com/watch?v=Z3mswCN2FJs>

## 3.5.9 Parentheses and Operator Hierarchy

- If no parentheses are used, operations are performed in the hierarchical order indicated.
- Operators on the same level are processed from left to right, except for those that are right-associative.

Operators	Associativity
<code>[] . ()</code> (method call)	Left to right
<code>! ~ ++ -- + (unary) - (unary) () (cast) new</code>	Right to left
<code>+ - * / %</code>	Left to right
<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>	Left to right
<code>&lt; &lt;= &gt; &gt;= instanceof</code>	Left to right
<code>== !=</code>	Left to right
<code>&amp;   ^ &amp;&amp;   </code>	Left to right
<code>?:</code>	Right to left
<code>= += -= *= /= %= &amp;=  = ^= &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</code>	Right to left

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

# Strings in Java

- Sequences of Unicode characters.
  - E.g., "Java\u2122" consists of the five Unicode characters J, a, v, a, and <sup>TM</sup>.
- Java does not have a built-in string type. Instead, the standard Java library has a predefined **String** class.
  - String literals enclosed in double quotes " " .
  - Each quoted string is an instance of the String class.

```
String e = "";           // an empty string
String greeting = "Hello"; // a string consisting of "Hello"
```

## 3.6.1 Substrings

- You can extract a substring from a larger string with the **substring** method of the String class.

```
String greeting = "Hello";  
String s = greeting.substring(0, 3); // s is "Hel"
```

- The second parameter of substring is the first position that you do not want to copy.
  - In the above example, it means from position **0 inclusive** to position **3 exclusive**.
- The string **s.substring(a, b)** always has length **b - a**.
  - E.g., the substring "Hel" has length 3 - 0 = 3.*



## 3.6.2 Concatenation

- You can use **+** to join (concatenate) two strings.

```
String firstName = "Harry";  
String lastName = "Cui";  
String fullName = firstName + lastName; // "HarryCui"
```

- No space between the words: The **+** operator joins two strings in the order received, exactly as they are given.
- When you concatenate a string with a value that is not a string, the latter is converted to a string.
  - *Every Java object can be converted to a string!*

```
int age = 16;  
String rating = "PG" + age; // "PG16"  
System.out.println("Age is " + age); // outputs "Age is 16"
```

## 3.6.2 Concatenation

- If you need to put multiple strings together, separated by a delimiter, use the **static** **join** method:

```
String all = String.join(" / ", "S", "M", "L", "XL");  
// all is the string "S / M / L / XL"
```

- As of Java 11, there is a **repeat** method:

```
String repeated = "Java".repeat(3);  
// repeated is "JavaJavaJava"
```

## 3.6.3 Strings are immutable

- The **String** class gives **no methods** that let you change a character in an existing string!
  - The objects of the **String** class are **immutable**.

```
String greeting = "Hello";  
greeting = "Help"; // greeting = greeting.substring(0, 3) + "p!";
```

Changing the contents of a **String** variable will make it refer to a different string.

## 3.6.4 Test strings for equality

- To test whether two strings are equal, use the **equals** method, i.e., **s.equals(t)**.
  - Returns **true** if the strings **s** and **t** are equal, **false** otherwise.
  - The strings **s** and **t** can be string variables or string literals.
- To test whether two strings are identical except for the upper/lowercase letter distinction, use **equalsIgnoreCase** method.

```
"Hello".equals(greeting);  
"Hello".equalsIgnoreCase(greeting);
```

- Do not use the **==** operator, which only test if the strings are stored in the same location.

```
String str = "Hello";  
System.out.println("Hello" == str);
```

# The == operator?

- Do not use the == operator, which only test if the strings are stored in the same location.

```
String str = "Hello";  
System.out.println("Hello"== str); // Outputs true
```

- If the virtual machine always arranges for equal strings to be shared, then you could use the == operator for testing equality.
  - But only string literals are shared, not strings that are the result of operations like + or substring.

```
String str2 = "Helloabc".substring(0,5);  
System.out.println(str == str2);           // false  
System.out.println(str.equals(str2));       // true
```

## 3.6.5 Empty and null strings

- The empty string `""` is a string of length 0.
  - `if (str.length() == 0)`
  - `if (str.equals(""))`
- An empty string is a Java object which holds the string length (namely, 0) and an empty contents.
- However, a String variable can also hold a special value, called `null`, that indicates that no object is currently associated with the variable.
  - `if (str == null)`
  - So, when you want to test if `str` is neither `null` nor empty.

```
if (str != null && str.length() != 0)
```

## 3.6.6 Code points and code units

- Java strings are sequences of **char** values.
  - The **char** data type is a code unit for representing Unicode code points in the UTF-16 encoding.
  - The most commonly used Unicode characters can be represented with a single code unit.
  - The supplementary characters require a pair of code units.
- **s.length()** is the number of code units (not Unicode characters, which means code unit size is 8 bits, just for simplicity, this method counts the number of characters in string s).
- **s.charAt(i)** is the  $i^{\text{th}}$  code unit.
- To get the  $i^{\text{th}}$  code point:

```
int index = s.offsetByCodePoints(0, i);  
int cp = s.codePointAt(index); // returns the Unicode value  
of the character at the specified index in a string.
```

- To get all code points:

```
int[] codePoints = str.codePoints().toArray();
```

## 3.6.7 The String API

- Many other useful String methods.
- `trim()` yields a new string, trimming leading and trailing white space.
- `toLowerCase()` yields a new string that converts all uppercase characters to lowercase.
- `indexOf()`, `lastIndexOf()` find the location of a substring.
- Check out the online API documentation.

<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/String.html>



## 3.6.9 Building strings

- The `StringBuilder` class is to build up strings from shorter strings, such as keystrokes or words from a file.
  - Using string concatenation is inefficient because every time you concatenate strings, a new `String` object is constructed.

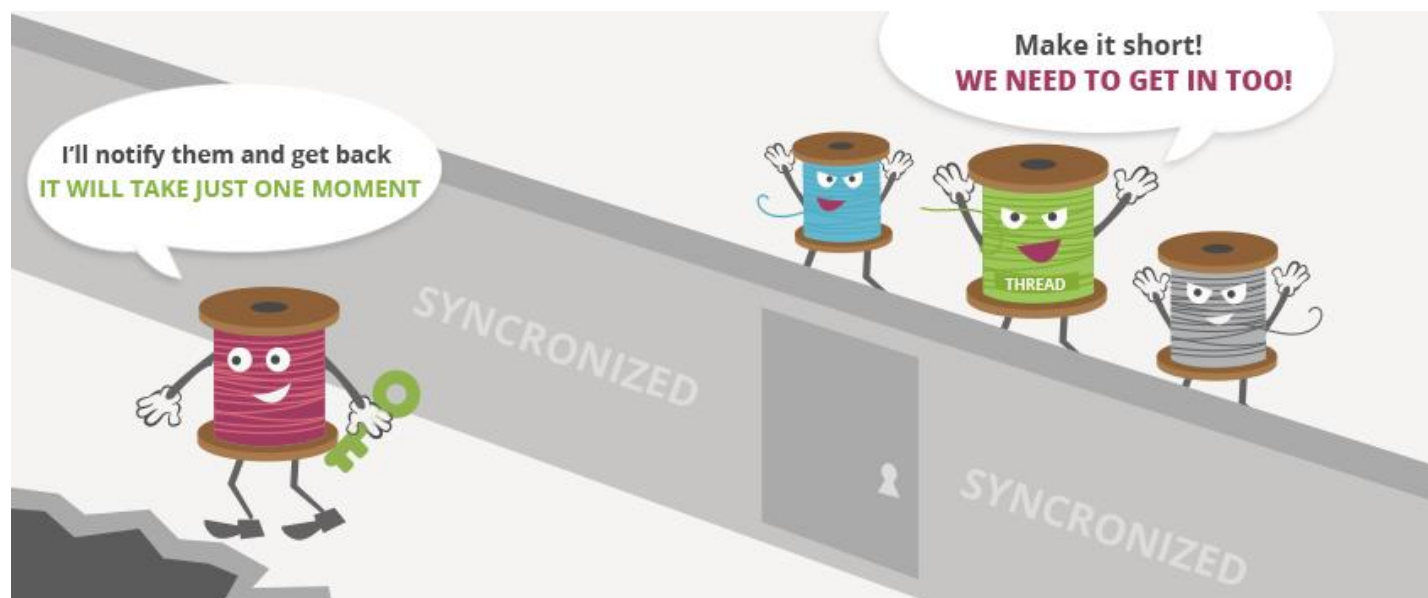
```
StringBuilder builder = new StringBuilder();
builder.append("This ");
builder.append("is ");
builder.append("a ");
builder.append("message.");

String completedString = builder.toString();
System.out.println(completedString);
```

# StringBuilder vs StringBuffer

StringBuffer	StringBuilder
Thread-Safe	Not Thread-Safe
Synchronized	Not Synchronized
Since Java 1.0	Since Java 1.5
Slower	Faster

- The APIs of both classes are identical.



<https://www.techyourchance.com/wp-content/uploads/2016/11/observer-thread-synchronization.jpg>

# Contents

- 3.1 A Simple Java Program
- 3.2 Comments
- 3.3 Data Types
- 3.4 Variables and Constants
- 3.5 Operators
- 3.6 Strings
- 3.7 Input and Output
- 3.8 Control Flow
- 3.9 Big Numbers
- 3.10 Arrays

## 3.7.1 Reading Input

- To read console input, you first construct a **Scanner** that is attached to **System.in**.

```
Scanner in = new Scanner(System.in);
```

- Now you can use the various methods of the **Scanner** class.

```
System.out.print("What is your name? ");  
String name = in.nextLine(); // reads a line of input  
String firstName = in.next(); // reads a single word  
  
System.out.print("How old are you? ");  
int age = in.nextInt(); // reads an integer
```

# InputTest.java

```
import java.util.*;
public class InputTest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // get first input
        System.out.print("What is your name? ");
        String name = in.nextLine();

        // get second input
        System.out.print("How old are you? ");
        int age = in.nextInt();

        // display output on console
        System.out.println("Hello, " + name + ". Next year,
you'll be " + (age + 1));
    }
}
```

# Reading password

- The **Scanner** class is not suitable for reading a password from a console since the input is plainly visible to anyone. Java 6 introduces a **Console** class specifically for this purpose.
- To read a password, use the following code:

```
Console cons = System.console();  
String username = cons.readLine("User name: ");  
char[] passwd = cons.readPassword("Password: ");
```

- Input processing with a **Console** object is not as convenient as with a **Scanner**.
  - You must read the input a line at a time.
  - There are no methods for reading individual words or numbers.
  - *May not be available in you IDE.*

## 3.7.2 Formatting output

```
double x = 10000.0 / 3.0;  
System.out.print(x);           // Outputs 3333.3333333333335
```

- This is a problem if you want to format your output, just like the `printf()` in C.
  - Fortunately, Java 5 brought back this useful method.

```
System.out.printf("%8.2f", x); // Outputs " 3333.33"
```

- You can also supply multiple parameters to `printf()`.

```
System.out.printf("Hi, %s. You'll be %d", name, age);
```

# Conversions for printf()

- Each of the format specifiers that start with a % character is replaced with the corresponding argument.
  - The conversion character that ends a format specifier indicates the type of the value to be formatted.
  - E.g., **f** is a floating-point number, **s** a string, and **d** a decimal integer.

Conversion Character	Type	Example
<b>d</b>	Decimal integer	200
<b>x</b>	Hexadecimal integer	c8
<b>o</b>	Octal integer	310
<b>f</b>	Fixed-point floating point	15.9
<b>e</b>	Exponential floating-point	1.59e+01
<b>s</b>	String	Hello



# Flags and String.format()

- You can also specify flags that control the appearance of the formatted output.
  - *E.g., the comma flag adds group separators.*

```
System.out.printf("%,.2f", 10000.0 / 3.0); // Outputs 3,333.33
```

- Using **String.format()** to create a formatted string without printing it.

```
String msg = String.format("Hi, %s. You'll be %d", name, age);
```