西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object-Oriented Programming

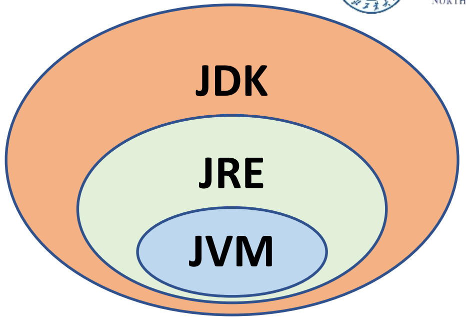
Chapter 3
Fundamental Programming Structures in Java

Dr. Helei Cui

20 Mar 2023

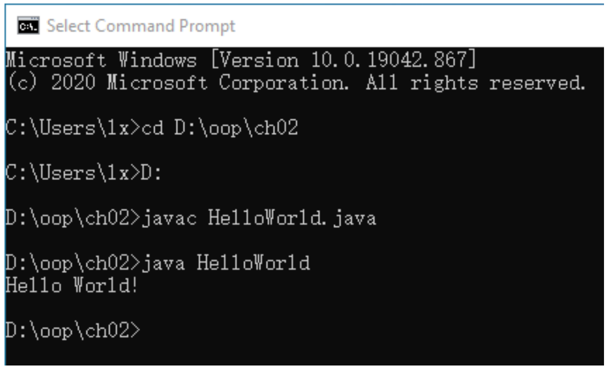
Slides partially adapted from lecture notes by Cay Horstmann

Recap



➢JDK, JRE, JVM?

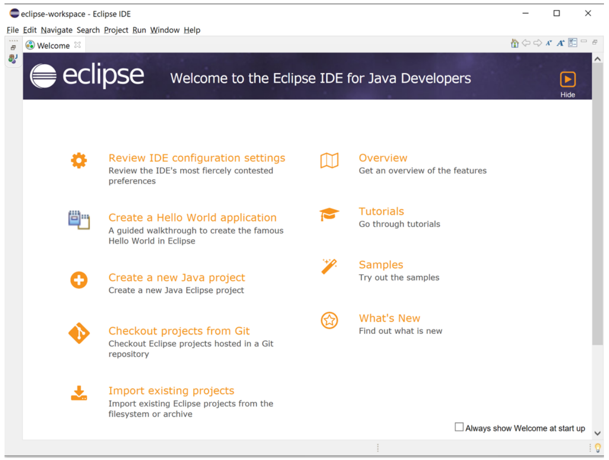
- JDK is a software development kit
- JRE is a software bundle that allows Java program to run
- JVM is an environment for executing bytecode



```
Select Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\lx>cd D:\oop\ch02

C:\Users\lx>D:

D:\oop\ch02>javac HelloWorld.java

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

D:\oop\ch02>
```

➢Run HelloWorld.java

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


eclipse-workspace - Eclipse IDE
File Edit Navigate Search Project Run Window Help
Welcome
eclipse
Welcome to the Eclipse IDE for Java Developers
Hide

Review IDE configuration settings
Review the IDE's most fiercely contested preferences

Create a Hello World application
A guided walkthrough to create the famous Hello World in Eclipse

Create a new Java project
Create a new Java Eclipse project

Checkout projects from Git
Checkout Eclipse projects hosted in a Git repository

Import existing projects
Import existing Eclipse projects from the filesystem or archive

Overview
Get an overview of the features

Tutorials
Go through tutorials

Samples
Try out the samples

What's New
Find out what is new

Always show Welcome at start up

➢Use UML editor

➢ Violet or diagrams.net

Violet UML Editor

diagrams.net

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)
`int`	4 bytes	-2,147,483,648 to 2,147,483,647 (just over 2 billion)
`short`	2 bytes	-32,768 to 32,767
`long`	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`byte`	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.

```java
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
float	4 bytes	Approximately ±3.40282347E+38F (6-7 significant decimal digits)
double	8 bytes	Approximately ±1.79769313486231570E+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
`\b`	Backspace	`\u0008`
`\t`	Tab	`\u0009`
`\n`	Linefeed	`\u000a`
`\r`	Carriage return	`\u000d`
`\"`	Double quote	`\u0022`
`\'`	Single quote	`\u0027`
`\\`	Backslash	`\u005c`

Java backspace escape doesn't work?
https://stackoverflow.com/questions/3328824/java-backspace-escape

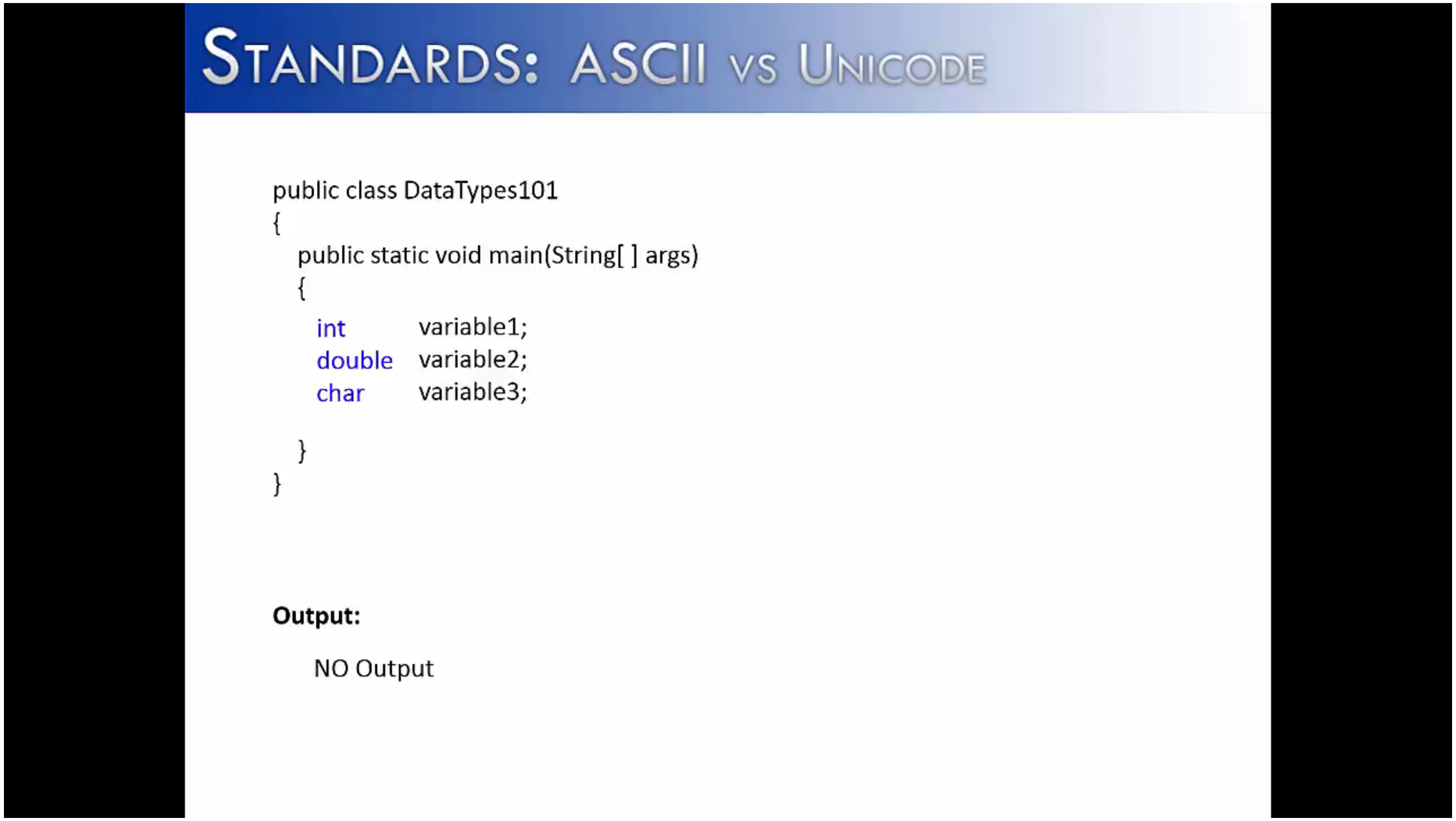
```
System.out.println("She said \"Hello!\" to me.");
// output: She said "Hello!" to me.
```

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
See more on https://www.youtube.com/watch?v=ut74oHojxqo

3.3.5 The boolean type

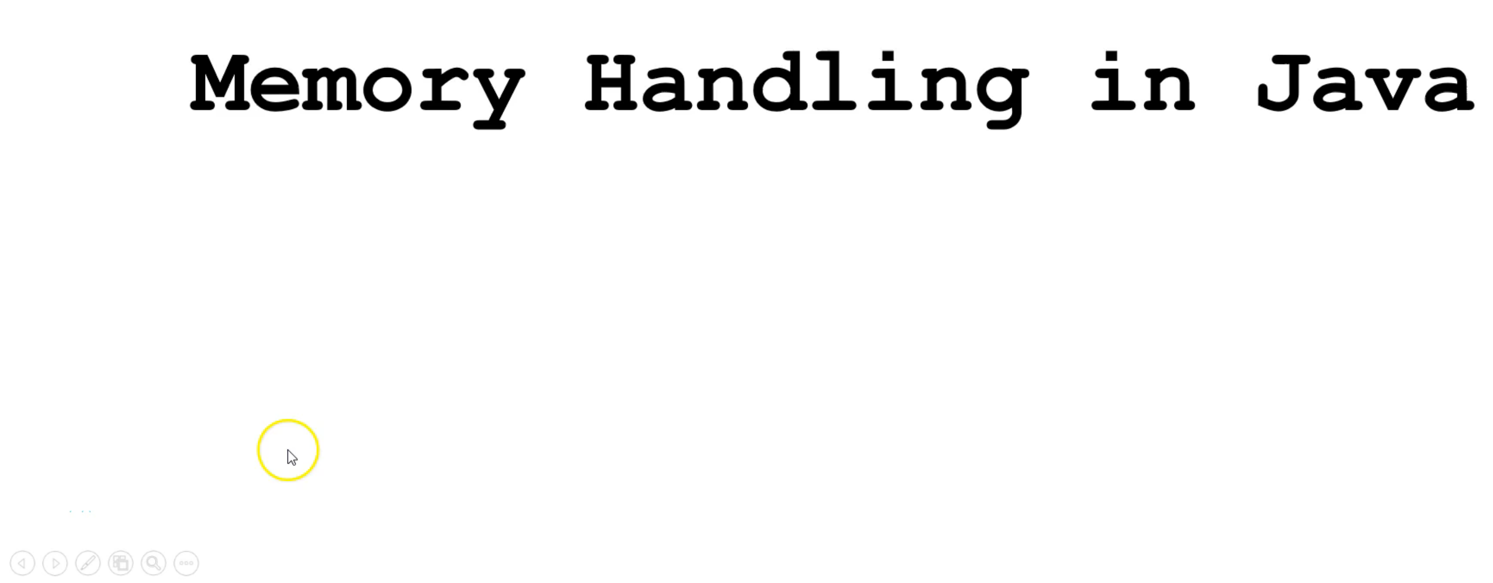
- Used for evaluating logical conditions.
  - Only two values: **`true`** or **`false`**
  - No conversion between integers and Boolean values.

```
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

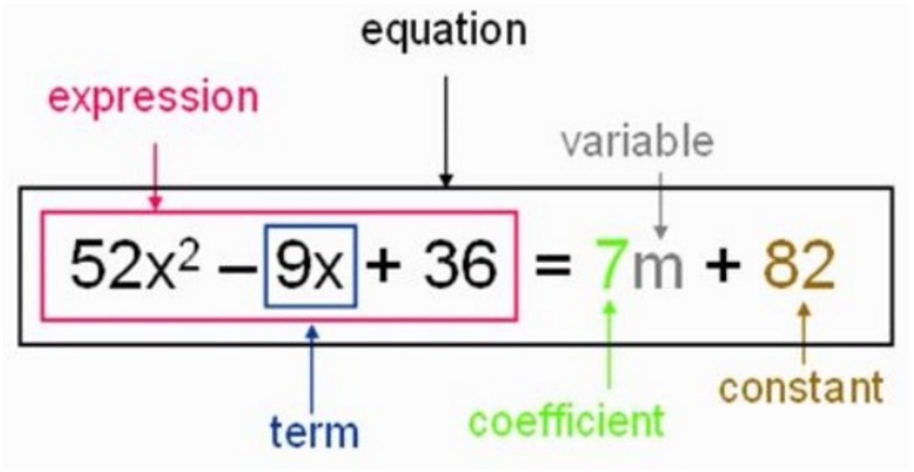
https://www.youtube.com/watch?v=LTnp79Ke8FI

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 - floor(-7 / 3) * 3 = -7 - (-3) * 3 = -7 + 9 = 2

  7 % (-3) = 7 - 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 floor(x).

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

  -7 % 3 = -7 - trunc(-7 / 3) * 3 = -7 - (-2) * 3 = -7 + 6 = -1

  7 % (-3) = 7 - 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

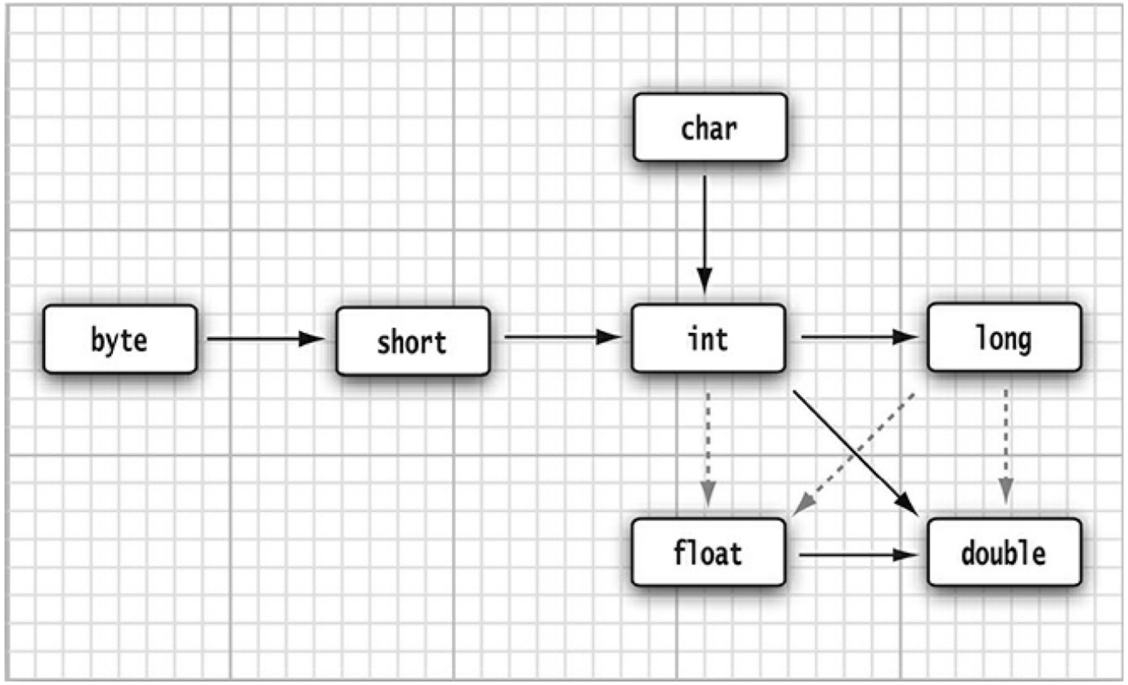
- The `Math` class contains methods for performing basic numeric operations and constants π and e.
  - E.g., the elementary exponential, logarithm, square root, and trigonometric functions.

```
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/17/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

x	!x
0	1
1	0

AND

x	y	x&&y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	X||y
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

- >> (right shift) << (left shift)
- They shift a bit pattern right or left.
    - *Assuming A is 60 (0011 1100), then*
        - *A << 2 will give 240 which is 1111 0000*
        - *A >> 2 will give 15 which is 1111*
    - *E.g., int fourthBitFromRight = (n & (1 << 3)) >> 3;*
- >>> (zero fill right shift)
- It fills the top bits with zero, unlike >> which extends the sign bit into the top bits.
    - *Assuming A is 60 (0011 1100), then*
        - *A >>>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: 11111111111111111111111111110001

C = 1073741809 in binary: 111111111111111111111111110001
```

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
`[]` `.` `()` (method call)	Left to right
`!` `~` `++` `--` `+` (unary) `-` (unary) `()` (cast) `new`	Right to left
`+` `-` `*` `/` `%`	Left to right
`<<` `>>` `>>>`	Left to right
`<` `<=` `>` `>=` `instanceof`	Left to right
`==` `!=`	Left to right
`&` `|` `^` `&&` `||`	Left to right
`?:`	Right to left
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=` `>>>=`	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 ™.

- 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 ith code unit.
- To get the ith 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/17/docs/api/java.base/java/lang/String.html

3.6.9 Building strings

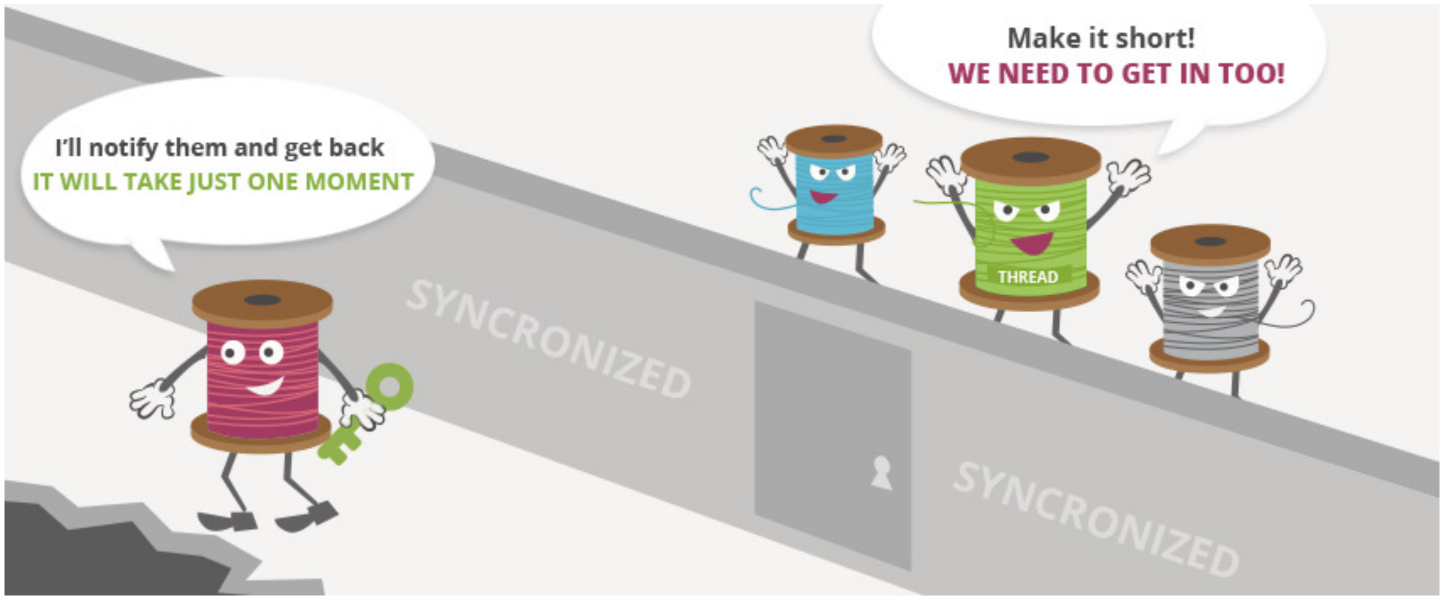
- The `StringBuilder` class is to build up strings from shorter strings, such as keystrokes or words from a file.
  - Using string concatenation is inefficient because every time you concatenate strings, a new String object is constructed.

```
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
`d`	Decimal integer	`200`
`x`	Hexadecimal integer	`c8`
`o`	Octal integer	`310`
`f`	Fixed-point floating point	`15.9`
`e`	Exponential floating-point	`1.59e+01`
`s`	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);
```

3.7.3 File input and output

- **To read from a file, construct a `Scanner` object like this:**

```
Scanner in = new Scanner(Path.of("myfile.txt"),
StandardCharsets.UTF_8);
```

  - If the file name contains backslashes, remember to escape each of them with an additional backslash.
  - *E.g., "c:\\mydirectory\\myfile.txt".*
  - Then you can use the `Scanner` method to read lines or integers.

```
String msg = in.nextLine();
System.out.println(msg);
```

3.7.3 File input and output

- **To write to a file, construct a `PrintWriter` object.**

```
PrintWriter out = new PrintWriter("myfile.txt",
StandardCharsets.UTF_8);
```

  - If the file does not exist, it is created.
  - Then you can use the `print`, `println`, and `printf` commands as you did when printing to `System.out`.

```
out.println("This is a new string!");
out.close();
```

Using buffered writer/reader (6 min)

https://www.youtube.com/watch?v=hgF21imQ_Is

Why BufferedReader/BufferedWriter

- It is recommended to use buffered I/O streams as opposed to `Scanner` and `PrintWriter` classes:
  - **They have significantly larger buffer memory** than `Scanner` and `PrintWriter`. It is recommended to use `BufferedReader` if you want to get long strings from a stream, and use `Scanner` if you want to parse specific type of token from a stream.
  - **Buffered Streams are synchronous** while unbuffered are not. This means you can work with multiple threads when using Buffered Streams.
  - **`Scanner` is memory and CPU heavy** when compared to `BufferedReader` because it internally uses "regular expressions" for matching your "`nextXXX()`" as opposed to just reading everything till the end of line as in the case of a regular Reader.
  - **`BufferedReader` is a bit faster** as compared to `Scanner`.
  - See more https://medium.com/@isaacjumba/why-use-bufferedreader-and-bufferedwriter-classses-in-java-39074ee1a966

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

3.8.1 Block scope

- A block, or compound statement, consists of a number of Java statements, surrounded by a pair of braces.
  - Blocks define the scope of your variables. A block can be nested inside another block.

```
public static void main(String[] args) {
    int n;
    . . .
    {
        int k;
        . . .
    } // k is only defined up to here
}
```

3.8.1 Block scope

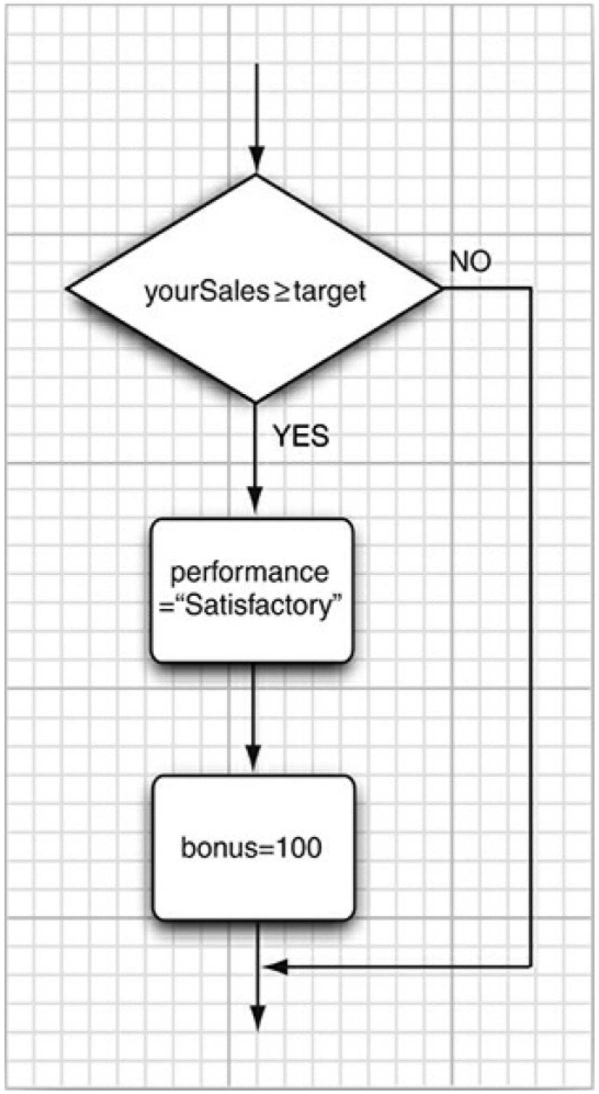
- You may not declare identically named variables in two nested blocks.

```
public static void main(String[] args) {
    int n;
    . . .
    {
        int k;
        int n; // ERROR – can’t redefine n in inner block
        . . .
    }
}
```

3.8.2 Conditional Statements

- `if (condition) statement`
  - The condition must be surrounded by parentheses.
  - For multiple statements, you should use a block.

```
if (yourSales >= target) {
    performance = "Satisfactory";
    bonus = 100;
}
```



if/else statement

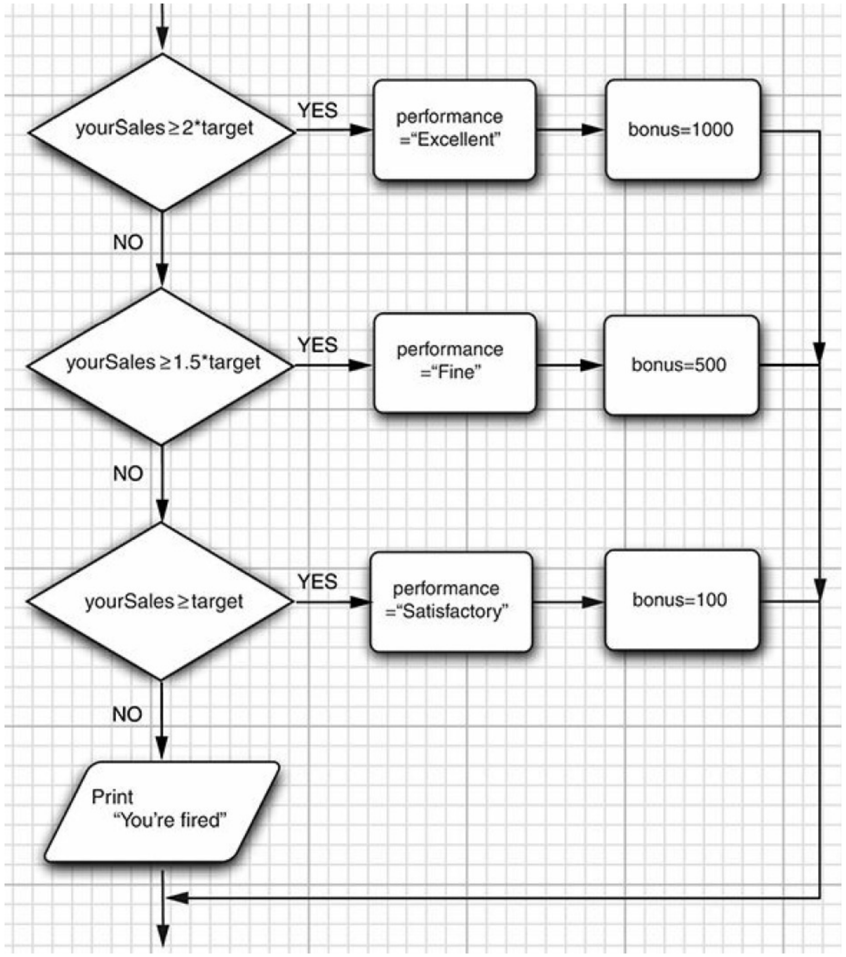
- `if (condition) statement1 else statements`
  - The condition must be surrounded by parentheses.
  - For multiple statements, you should use a block.

```
if (yourSales >= target) {
    performance = "Satisfactory";
    bonus = 100 + 0.01 * (yourSales - target);
} else {
    performance = "Unsatisfactory";
    bonus = 0;
}
```

It is a good idea to use braces "{}" to clarify the structure.

else if statement

- `if ... else if ...`



```
if (yourSales >= 2*target) {
    // ...
} else if (yourSales >= 1.5*target) {
    // ...
} else if (yourSales >= target) {
    // ...
} else {
    // ...
}
```

3.8.3 Loops

- **while** (condition) statement
  - Executes a statement (which may be a block statement) while a condition is true.
  - Tests the condition at the top. So the code in the block might never be executed.

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

3.8.3 Loops

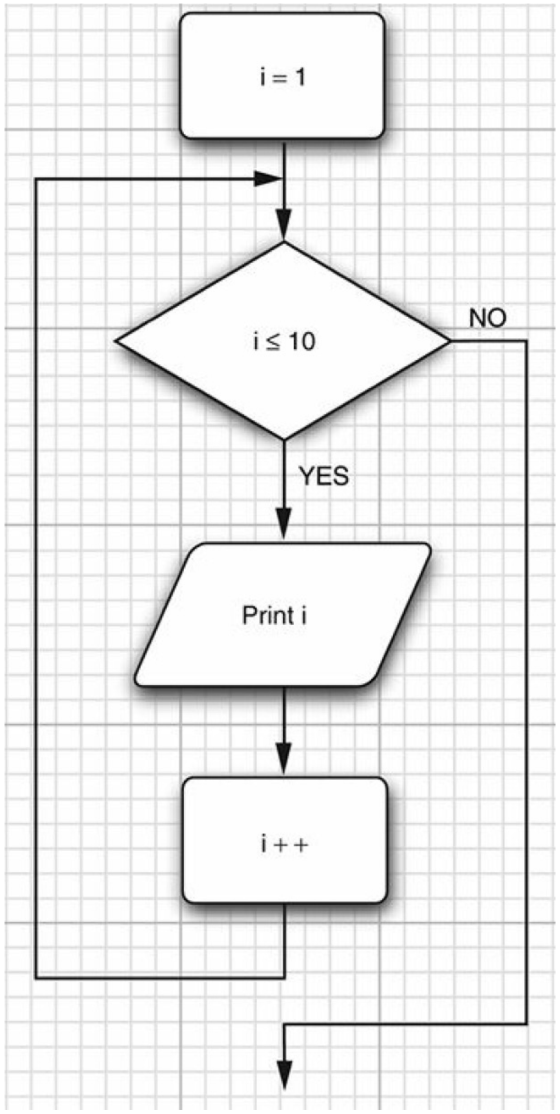
- **do** statement **while** (condition);
  - Ensures that a statement of a block is executed at least once.
  - If the condition is true, it repeats the statement and retests the condition, and so on.

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```

3.8.4 Determinate loops

- The **for** loop is a general construct to support iteration controlled by a counter or similar variable that is updated after every iteration.

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```



for loop

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

- The first slot (`int i = 1;`) of the for statement usually holds the counter initialization.
- The second slot (`i <= 10;`) gives the condition that will be tested before each new pass through the loop.
- The third slot (`i++`) specifies how to update the counter.

Unwritten rule: the three slot should only initialize, test, and update the same counter variable.

Caution

```
for (double x = 0; x != 10; x += 0.1) {
    System.out.println(x);
}
```

- **This might never stop!**
- Because of roundoff errors, the final value might not be reached exactly.
  - In this example, x jumps from 9.99999999999998 to 10.09999999999998 because there is no exact binary representation for 0.1.

Be careful with testing for equality of floating-point numbers in loops.

Variable scope in for loop

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
} // i no longer defined here
```

```
int i;
for (i = 1; i <= 10; i++) {
    System.out.println(i);
} // i is still defined here
```

```
for (int i = 1; i <= 10; i++) {
    ...
}
for (int i = 1; i <= 10; i++) { // This is OK
    ...
}
```

for loop and while loop

- **A for loop can be viewed as a convenient shortcut for a while loop.**

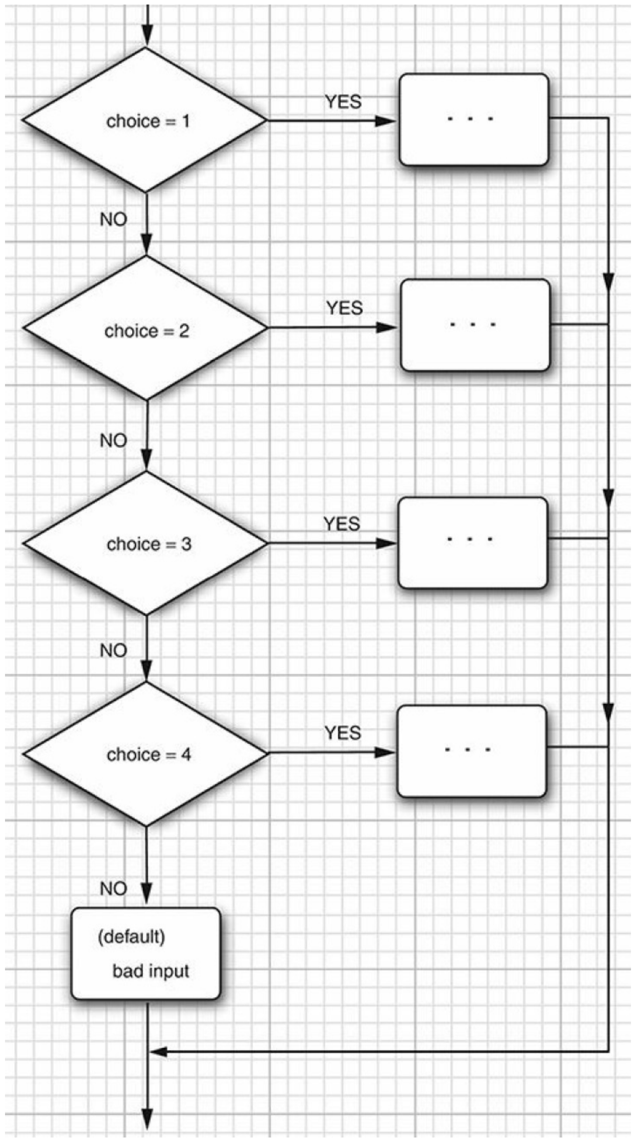
```
for (int i = 10; i > 0; i--) {
    System.out.println("Counting down . . . " + i);
}
```

```
int i = 10;
while (i > 0) {
    System.out.println("Counting down . . . " + i);
    i--;
}
```

3.8.4 Multiple selections

- **switch statement**
  - Deals with multiple selections with many alternatives.

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2,
3, 4) ");
int choice = in.nextInt();
switch (choice) {
   case 1:      . . .      break;
   case 2:      . . .      break;
   case 3:      . . .      break;
   case 4:      . . .      break;
   default: // bad input
                . . .      break;
}
```



Caution

```
int num = 2;
switch (num) {
    case 1:
        System.out.print("One");
    case 2:
        System.out.print("Two");
    case 3:
        System.out.print("Three");
    default:
        System.out.print("Others");
}
// Outputs TwoThreeOthers
```

- **When there is no “break;”, execution will fall through to the next alternative!**

The case label

- A case label can be:
  - A constant expression of type char, byte, short, or int;
  - An enumerated constant;
  - Starting with Java 7, a string literal.

```
String input = . . .;
switch (input.toLowerCase()) {
    case "yes": // OK since Java 7
    ...
}
Size sz = . . .;
switch (sz) {
    case SMALL: // no need to use Size.SMALL
    ...
}
```

3.8.5 Statements that break control flow

- `break` statement
  - Used to “jump out” of a switch statement.
  - **Can also be used to jump out of a loop.**

```
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        break;
    }
    System.out.print(i);
}
// Outputs 0123
```

3.8.5 Statements that break control flow

- **`continue`** `statement`
  - Used to **break one iteration (in the loop)**, if a specified condition occurs, and continues with the next iteration in the loop.

```
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        continue;
    }
    System.out.print(i);
}
// Outputs 012356789
```

Quick question 6

Write a Java program to check whether a number is a prime or not.

- A prime number is a number which is divisible by only two numbers: 1 and itself.
- So, if any number is divisible by any other number, it is not a prime number.
- E.g., 29 is a prime number, 33 is not a prime number.

```java
import java.util.Scanner;
public class CheckPrime {
    public static void main(String[] args) {
        System.out.print("Please input an integer: ");
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        boolean flag = false;
        for (int i = 2; i <= num / 2; ++i) {
            if (num % i == 0) {    // condition for non-prime number
                flag = true;
                break;
            }
        }
        if (!flag) {
            System.out.println(num + " is a prime number.");
        } else {
            System.out.println(num + " is not a prime number.");
        }
    }
}
```

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

3.9 Big numbers

- **The precision of the basic integer and floating-point types is not sufficient!**
- Solution: the `java.math` package has classes for dealing with numbers with an arbitrarily long sequence of digits.
    - `BigInteger` implements arbitrary-precision integer arithmetic.
    - `BigDecimal` does the same for floating-point numbers.
    - Use the static `valueOf` method to turn a number into a big number.

```
BigInteger a = BigInteger.valueOf(100);
BigDecimal b = BigDecimal.valueOf(2.0);

// You can also use a constructor with a string parameter
BigInteger c = BigInteger.valueOf("12345678901234567890");
```

BigInteger

- There are also constants:
  - `BigInteger.ZERO`
  - `BigInteger.ONE`
  - `BigInteger.TEN`
  - `BigInteger.TWO` (since Java 9)
- You **cannot** use the familiar mathematical operators like +, -, *, /, % to combine big numbers.
  - Use `add`, `subtract`, `multiply`, `divide`, `mod` methods.

```
BigInteger a = BigInteger.valueOf(100);
BigInteger b = BigInteger.valueOf(200);
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2)));
// d = c * (b + 2)
```

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/math/BigInteger.html
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/math/BigDecimal.html

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

3.10.1 Declaring arrays

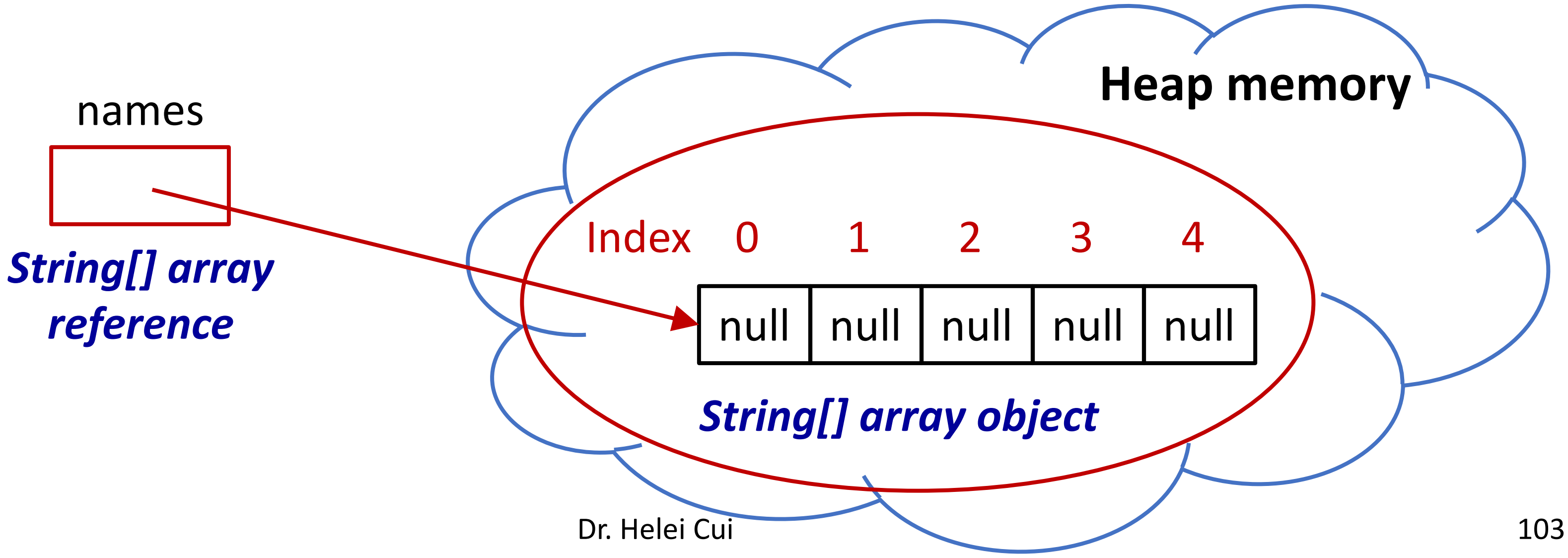
- An array is a data structure that stores a collection of values of the same type.
  - Specify the array type (that is the element type followed by `[]`) and the array variable name, e.g., `int[] a`.
    - However, this only declares the variable a, not yet initialized with an actual array.
  - Use the `new` operator to initialize, e.g., `a = new int[100]`.
    - The array length need not be a constant, e.g., `new int[n]`;
    - But you cannot change its length once you create it.

```
int[] a = new int[100]; // declares and initializes an array
int n = 100;
int[] b = new int[n];   // the length is n
```

3.10.1 Declaring arrays

- When you create an array of numbers (e.g., `int`, `double`), all elements are initialized with ***zero***.
- Arrays of `boolean` are initialized with ***false***.
- Arrays of objects are initialized with the special value `null`, which indicates that they do not (yet) hold any objects.

```
String[] names = new String[5]; // strings are all null
```



3.10.1 Declaring arrays

- You can create an array object and supplying initial values without using the new operator and specifying the length.

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
String[] authors = {
    "James Gosling",
    "Bill Joy",
    "Guy Steele",
    // add more names here and put a comma after each name
};
```

- Anonymous array:

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
// is shorthand for
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

Empty array

- It is legal to have arrays of length 0.
  - It is useful if you write a method that computes an array result, and the result happens to be empty.
- Construct an empty array:
  - *E.g.,* *`new elementType[0]`* *or* *`new elementType[] {}`*.

Note that an array of length 0 is not the same as `null`.

3.10.2 Accessing array elements

- Access an array element via an integer index, e.g., `a[i]`.
  - The array elements are numbered from *0* to *length-1*.
    - *Accessing* a[length] *causes an "array index out of bounds" exception.*
  - You can use a loop to fill the elements in an array.

```
int[] a = new int[100];
for (int i = 0; i < 100; i++) {
    a[i] = i; // fills the array with numbers 0 to 99
}

for (int i = 0; i < 10; i++) names[i] = "";

for (int i = 0; i < a.length; i++) { // uses array.length
    System.out.println(a[i]);
}
```

3.10.3 The "for each" loop

- `for (variable : collection) statement`
  - Sets the given **variable** to each element of the **collection** and then executes the **statement** (which, of course, may be a block).
  - The **collection** expression must be an array or an object of a class that implements the `Iterable` interface, such as `ArrayList`.

```
for (int element : a) {
    System.out.println(element);
} // looks more concise and less error-prone
```

  - You can read this loop as "*for each element in a*".

```
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
} // a traditional for loop achieves the same effect
```

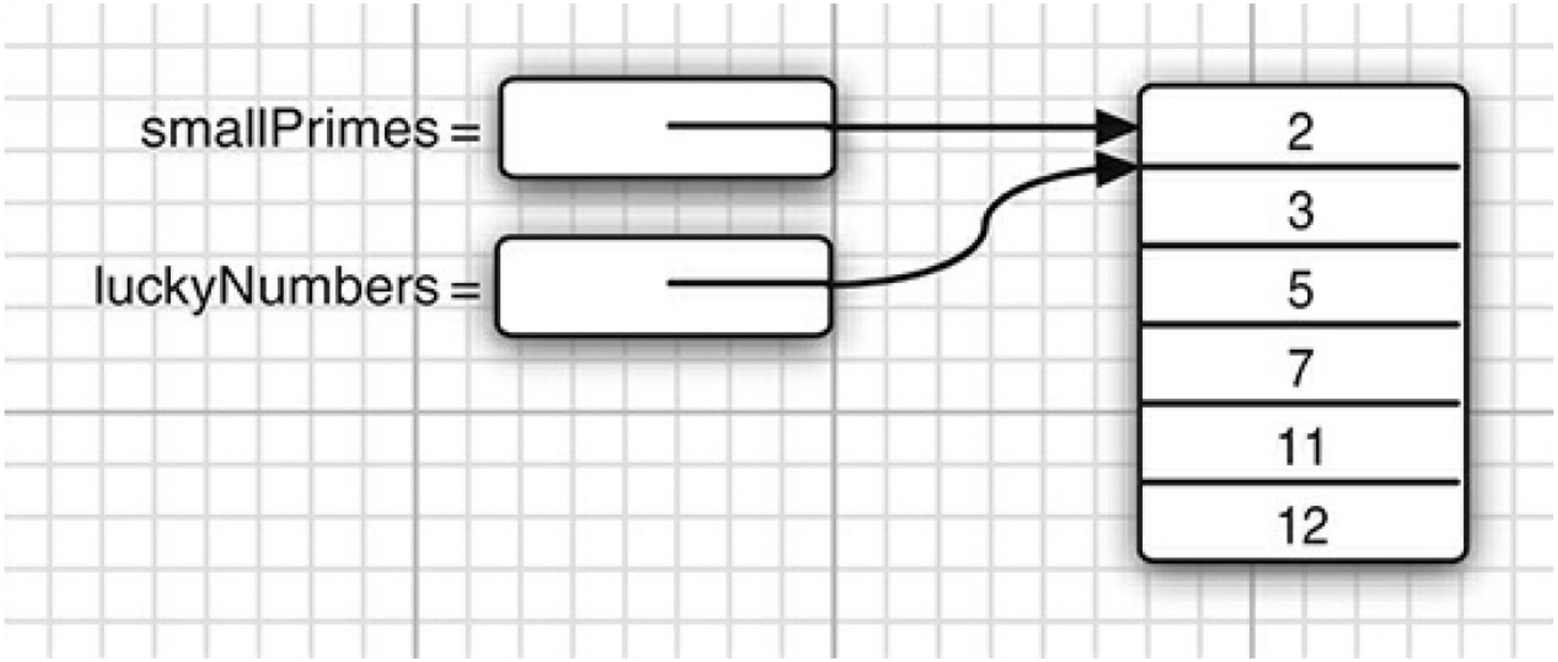
Output an array

- There is an even easier way to print all values of an array, using the `toString` method of the `Arrays` class.
- The call `Arrays.toString(a)` returns a string containing the array elements, enclosed in brackets and separated by commas, such as "[2, 3, 5, 7, 11, 13]".

```
int[] a = new int[] { 1, 2, 3, 4, 5, 6 };
System.out.println(Arrays.toString(a));
// [1, 2, 3, 4, 5, 6]
```

3.10.4 Array copying

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```



When copy one array variable into another, both variables refer to the same array.

Copy all values into a new array

- If you actually want to copy all values of one array into a new array, use the `copyOf` method in the `Arrays` class.

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers,
luckyNumbers.length);
```

  - The second parameter is *the length of the new array*.
  - A common use of this method is to increase the size of an array.

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, 2 *
luckyNumbers.length);
```

  - The additional elements are filled with `0` if the array contains numbers, `false` if the array contains boolean values.
  - Conversely, if the length is less than the length of the original array, only the initial values are copied.

3.10.5 Command-line parameters

- Every Java program has a main method with a "`String[] args`" parameter.
  - This parameter indicates that the main method receives an array of strings, namely, the arguments specified on the command line.

```
public class Message {
    public static void main(String[] args) {
        if (args.length == 0 || args[0].equals("-h"))
            System.out.print("Hello,");
        else if (args[0].equals("-g"))
            System.out.print("Goodbye,");
        // print the other command-line arguments
        for (int i = 1; i < args.length; i++)
             System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

3.10.5 Command-line parameters

- If you run the program as follows

```
> java Message -g wonderful world
```

- Then the `args` array has the following contents.
  - `args[0]: "-g"`
  - `args[1]: "wonderful"`
  - `args[2]: "world"`
- The program prints the message:

```
> Goodbye, wonderful world!
```

3.10.6 Array sorting

- Use Arrays.sort() method to sort the array via a tuned QuickSort algorithm.

```java
import java.util.Arrays;

public class SortExample {
    public static void main(String[] args) {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};
        System.out.printf("Original arr[] : %s\n", Arrays.toString(arr));
        Arrays.sort(arr);
        System.out.printf("Modified arr[] : %s\n", Arrays.toString(arr));
    }
}
// Original arr[] : [13, 7, 6, 45, 21, 9, 101, 102]
// Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]
```

3.10.7 Multidimensional arrays

- They use more than one index to access array elements.
  - Used for tables and other more complex arrangements.
  - `int[][]` is an array of arrays or a two-dimensional array:

```
int[][] magicSquare = new int[ROWS][COLUMNS]; // without initializer
```

  - You can initialize it without a call to `new`, if you know the elements:

```
int[][] magicSquare = {
   {16, 3, 2, 13},
   {5, 10, 11, 8},
   {9, 6, 7, 12},
   {4, 15, 14, 1}
};
```

  - Use two indexes to access element:
    - *E.g., `magicSquare[1][2] is 11`.*

Visit all elements in a 2-D array

```
for (int[] row : magicSquare) {
    for (int value : row) {
        System.out.print(value + " ");
    }
    System.out.println();
}
```

- A “for each” loop does not automatically loop through all elements in a two-dimensional array.
- Instead, it loops through the rows, which are themselves one-dimensional arrays.
- Try `Arrays.deepToString()` method:

```
System.out.println(Arrays.deepToString(a));
// [[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

3.10.8 Ragged arrays

- Java has no multidimensional arrays at all, only one-dimensional arrays.
    - Multidimensional arrays are faked as “arrays of arrays.”
- **If the rows have different lengths, the array is “ragged” :**

```
int[][] triangle = new int[5][];
for (int i = 0; i < 5; i++) {
    triangle[i] = new int[i+1];
}
System.out.println(Arrays.deepToString(triangle));
// [[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Recap

- 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™ 17

New Java SE Releases