Introduction to Java and Programs, and Elementary Programming

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 2

Announcements

- Assignment 1 will be released today
  - Due Oct 6, 11:59 PM
- Reading:
  - Liang
    - Chapters 1 and 2

Programs

- Computer programs (i.e., software) are instructions to the computer
- You tell a computer what to do through programs
- Computers do not understand human languages, so you need to use computer languages to communicate with them
- Programs are written using programming languages

Programming languages

- Machine language
- Assembly language
- High-level language

Programming languages

- Machine language
  - Machine language is a set of primitive instructions built into every computer
  - The instructions are in the form of binary code, so you must enter binary codes for various instructions
  - Programming with native machine language is a tedious process, and the programs are highly difficult to read and modify
  - For example, to add two numbers, you might write an instruction in binary like this:

    ```
    1101101010011010
    ```

Programming languages

- Assembly language
  - Assembly languages were developed to make programming easy (CSE 30 and ECE 30 are “easy”)
  - Since the computer cannot understand assembly language, a program called assembler is used to convert assembly language programs into machine code
  - For example, to add two numbers, you might write an instruction in assembly code like this:

```
ADDF3 R1, R2, R3
```

Programming languages

- High-level language
  - High-level languages are English-like and easy to learn and program
    - For example, the following is a high-level language statement that computes the area of a circle with radius 5:

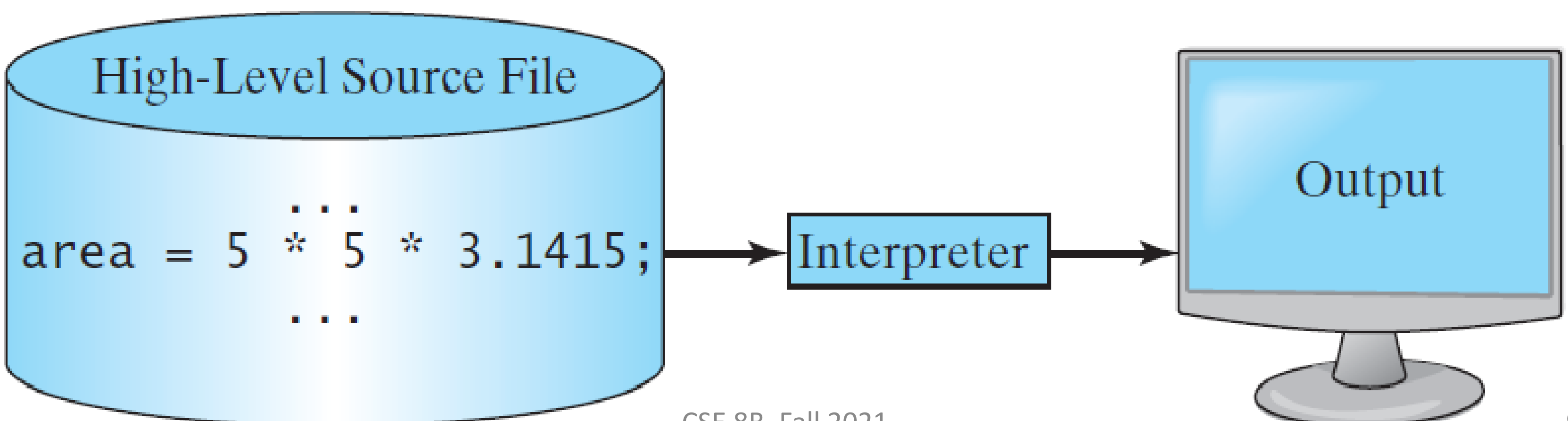
```
area = 5 * 5 * 3.1415;
```

Interpreting/Compiling source code

- A program written in a high-level language is called a source program or source code
- Because a computer cannot understand a source program, a source program must be translated into machine code for execution
- The translation can be done using another programming tool called an interpreter or a compiler

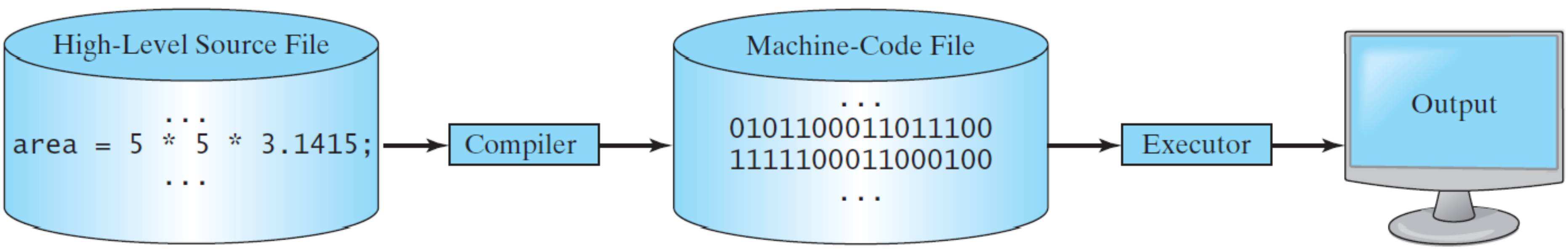
Interpreting source code

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away
- A statement from the source code may be translated into several machine instructions



Compiling source code

- A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed



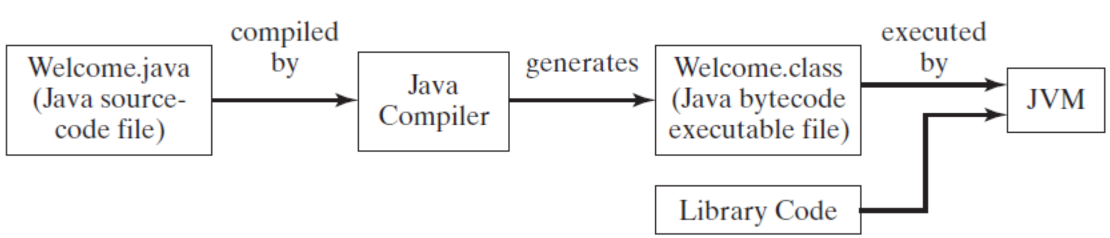
Java

- Java is a high-level language
- Java is a general purpose programming language
- Java can be used to develop standalone applications
- Java can be used to develop applications for web servers

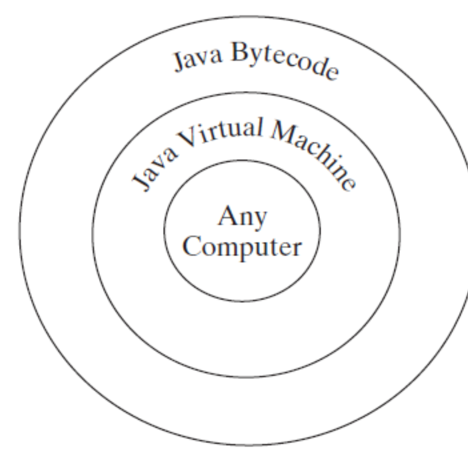
Java

- The *compiler* of Java is called `javac`
  - Java source code is compiled into the Java Virtual Machine (JVM) code called bytecode
- The *interpreter* of Java is called `java`
  - The bytecode is machine-independent and can run on any machine that has a Java interpreter, which is part of the JVM (write once, run anywhere)

Compile source code, interpret bytecode

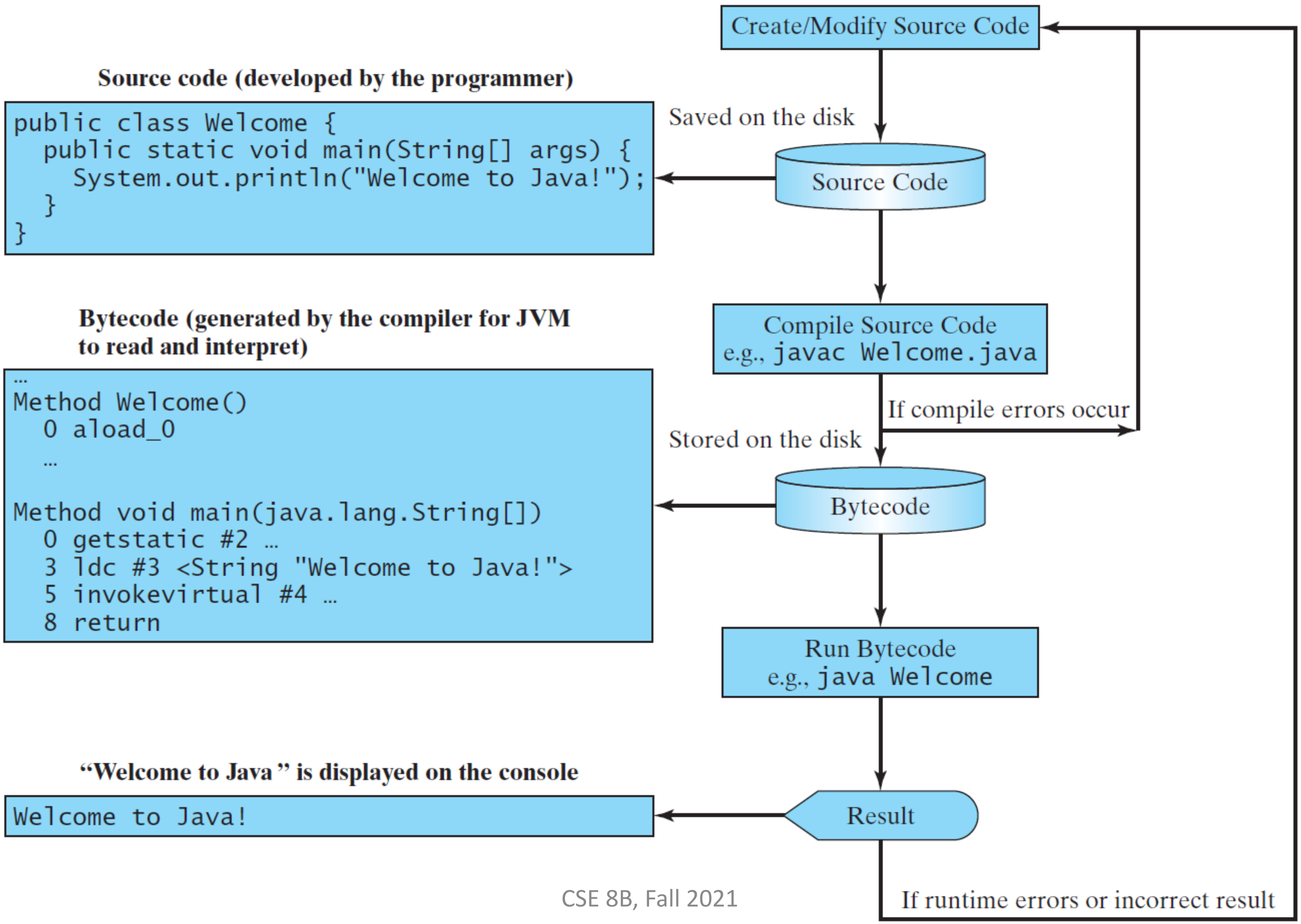


(a)



(b)

Developing, compiling, and running Java programs



Programming errors

- Syntax errors
  - Detected by the compiler
- Runtime errors
  - Causes the program to abort
- Logic errors
  - Produces incorrect result

Anatomy of a Java program

- Class name
- Main method
- Statements
- Statement terminator
- Reserved words
- Comments
- Blocks

Class name

- Every Java program must have at least one class
- Each class has a name
- Naming convention: capitalize the first letter of each word in the name class (e.g., `ComputeArea`)
- This class name is `Welcome`

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Main method

- This line defines the main method
- In order to run a class, the class must contain a method named main
- The program is executed from the main method

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Statement

- A statement represents an action or a sequence of actions
- This is a statement to display the greeting “Welcome to Java!”

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Statement terminator

- Every statement in Java ends with a semicolon

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Reserved words

- Reserved words or keywords are words that have a specific meaning to the compiler and cannot be used for other purposes in the program
- For example, when the compiler sees the word class, it understands that the word after class is the name for the class

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Blocks

- A pair of braces in a program forms a block that groups components of a program

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Class block

Method block

Blocks

- Two different block styles

Next-line style

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

Special symbols

Character	Name	Description
{}	Opening and closing braces	Denotes a block to enclose statements.
()	Opening and closing parentheses	Used with methods.
[]	Opening and closing brackets	Denotes an array.
//	Double slashes	Precedes a comment line.
" "	Opening and closing quotation marks	Enclosing a string (i.e., sequence of characters).
;	Semicolon	Marks the end of a statement.

Identifiers

- Identifiers are the names that identify the elements such as classes, methods, and variables in a program
- An identifier is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($)
- An identifier must start with a letter, an underscore (_), or a dollar sign ($)
- An identifier cannot start with a digit
- An identifier cannot be a reserved word
  - List of reserved words
    - Liang, Appendix A
    - https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html
    - https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.9
- An identifier cannot be `true`, `false`, or `null`
- An identifier can be of any length

Variables

- Variables are used to represent values that may be changed in the program

```
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius " +
   radius);

// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius " +
   radius);
```

Declaring variables

```
int x;         // Declare x to be an
               // integer variable
double radius; // Declare radius to
               // be a double variable
char a;        // Declare a to be a
               // character variable
```

Assignment statements

```
x = 1;            // Assign 1 to x
radius = 1.0;     // Assign 1.0 to radius
a = 'A';          // Assign 'A' to a
```

Declaring and initializing in one step

```
int x = 1;

double radius = 1.0;

char a = 'A';
```

Named constants

- Naming convention: capitalize all letters in constants, and use underscores to connect words

```
final datatype CONSTANTNAME = VALUE;

final double PI = 3.14159;
final int MAX_VALUE = 3;
```

Variable and method names

- Naming convention: Use lowercase. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name
  - For example, the variables `radius` and `area`, and the method `computeArea`.

Numerical data types

Name	Range	Storage Size
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

Number literals

- A literal is a constant value that appears directly in the program

```
int i = 34;
long x = 1000000;
double d = 5.0 + 1.0;
```

`34`, `100000`, `5.0`, and `1.0` are literals

Integer literals

- An integer literal can be assigned to an integer variable as long as it can fit into the variable
- A compilation error would occur if the literal were too large for the variable to hold
  - For example, the statement `byte b = 1000` would cause a compilation error, because `1000` cannot be stored in a variable of the `byte` type
- An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (equals -2147483648) to $2^{31}-1$ (equals 2147483647)
- To denote an integer literal of the `long` type, append it with the letter `L` or `l`
  - `L` is preferred because `l` (lowercase L) can easily be confused with `1` (the digit one)

Floating-point literals

- Floating-point literals are written with a decimal point
- By default, a floating-point literal is treated as a `double` type value
  - For example, `5.0` is considered a `double` value, not a `float` value
- You can make a number a `float` by appending the letter `f` or F, and make a number a `double` by appending the letter d or D
  - For example, you can use `100.2f` or `100.2F` for a float number, and `100.2d` or `100.2D` for a double number

Scientific notation

- Floating-point literals can also be specified in scientific notation
  - For example, `1.23456e+2` (same as `1.23456e2`) is equivalent to `123.456`, and `1.23456e-2` is equivalent to `0.0123456`
- `E` or `e` represents an exponent

Numeric operations

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

`double` vs `float`

- The `double` type values are more accurate than the `float` type values
  - For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`

7 digits

Floating-point accuracy

- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy
- For example,

  `System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);`

  displays `0.5000000000000001`, not `0.5`, and

  `System.out.println(1.0 - 0.9);`

  displays `0.09999999999999998`, not `0.1`
- Integers are stored precisely
  - Calculations with integers yield a precise integer result

Integer division

- Warning: resulting fractional part (i.e., values after the decimal point) are truncated, *not rounded*
  – For example 5 / 2 yields an integer 2

Remainder operator

- Example: an even number `%  2` is always `0` and an odd number `%  2` is always `1`
  - You can use this property to determine whether a number is even or odd
- Example: If today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression.

```
Saturday is the 6th day in a week
A week has 7 days
(6 + 10) % 7 is 2
The 2nd day in a week is Tuesday
After 10 days
```

Augmented assignment operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and decrement operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

Conversion rules

- When performing a binary operation involving two operands of *different* types, Java automatically converts the operand based on the following rules
  1. If one of the operands is `double`, the other is converted into `double`
  2. Otherwise, if one of the operands is `float`, the other is converted into `float`
  3. Otherwise, if one of the operands is `long`, the other is converted into `long`
  4. Otherwise, both operands are converted into `int`

Type casting

Implicit casting

`double d = 3;` (type widening)

Explicit casting

`int i = (int)3.0;` (type narrowing)

`int i = (int)3.9;` (fraction part is truncated, not rounded!)

range increases

→

byte, short, int, long, float, double

Reading numbers from the console

1. Create a Scanner object

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

2. Use the method nextDouble() to obtain to a double value. For example,

```
System.out.print("Enter a double value: ");
Scanner input = new Scanner(System.in);
double d = input.nextDouble();
```

Reading numbers from the console

```
Scanner input = new Scanner(System.in);
int value = input.nextInt();
```

Method	Description
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Explicit import and implicit Import

- At top of source file

```
import java.util.Scanner; // Explicit Import

import java.util.* ; // Implicit import
```

Next Lecture

- Selections
- Mathematical functions, characters, and strings
- Reading:
  - Liang
    - Chapters 3 and 4