Basics of Java Programming Language

Outline

- Installation and setting of JDK for Java programming environment
- JVM, java source file and Java Byte code
- the *java*.*lang* package
- the *Object* class
- Data type and identifiers

Outline…

- Operators
- Decision and Repetition Statements
- Java Standard input, output and error streams
- packages, interfaces, classes, objects and methods in java
- Constructors
- the main method in java applications
- the JAR Tool

What is Java?

What is java?

- Developed by Sun Microsystems (James Gosling)
- A general-purpose object-oriented language
- Based on C/C++
- Designed for easy Web/Internet applications
- Widespread acceptance

Java Features (1)

- **Simple**
  - fixes some clumsy features of C++
  - no pointers
  - automatic garbage collection
  - rich pre-defined class library http://java.sun.com/j2se/1.4.2/docs/api/
- **Object oriented**
  - focus on the data (objects) and methods manipulating the data
  - all functions are associated with objects
  - almost all datatypes are objects (files, strings, etc.)
  - potentially better code organization and reuse

Java Features (2)

- **Interpreted**
  - java compiler generate byte-codes, not native machine code
  - the compiled byte-codes are platform-independent
  - java bytecodes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
- **Portable**
  - same application runs on all platforms
  - the sizes of the primitive data types are always the same
  - the libraries define portable interfaces

Java Features (3)

- **Reliable**
  - extensive compile-time and runtime error checking
  - no pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
  - automatic garbage collection tracks objects usage over time
- **Secure**
  - usage in networked environments requires more security
  - memory allocation model is a major defense
  - access restrictions are forced (private, public)

Java Features (4)

- **Multithreaded**
  - multiple concurrent threads of executions can run simultaneously
  - utilizes a sophisticated set of synchronization primitives (based on monitors and condition variables paradigm) to achieve this
- **Dynamic**
  - java is designed to adapt to evolving environment
  - libraries can freely add new methods and instance variables without any effect on their clients
  - interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented
  - can check the class type in runtime

Java Disadvantages

- **Slower than compiled language such as C**
  - an experiment in 1999 showed that Java was 3 or 4 times slower than C or C++

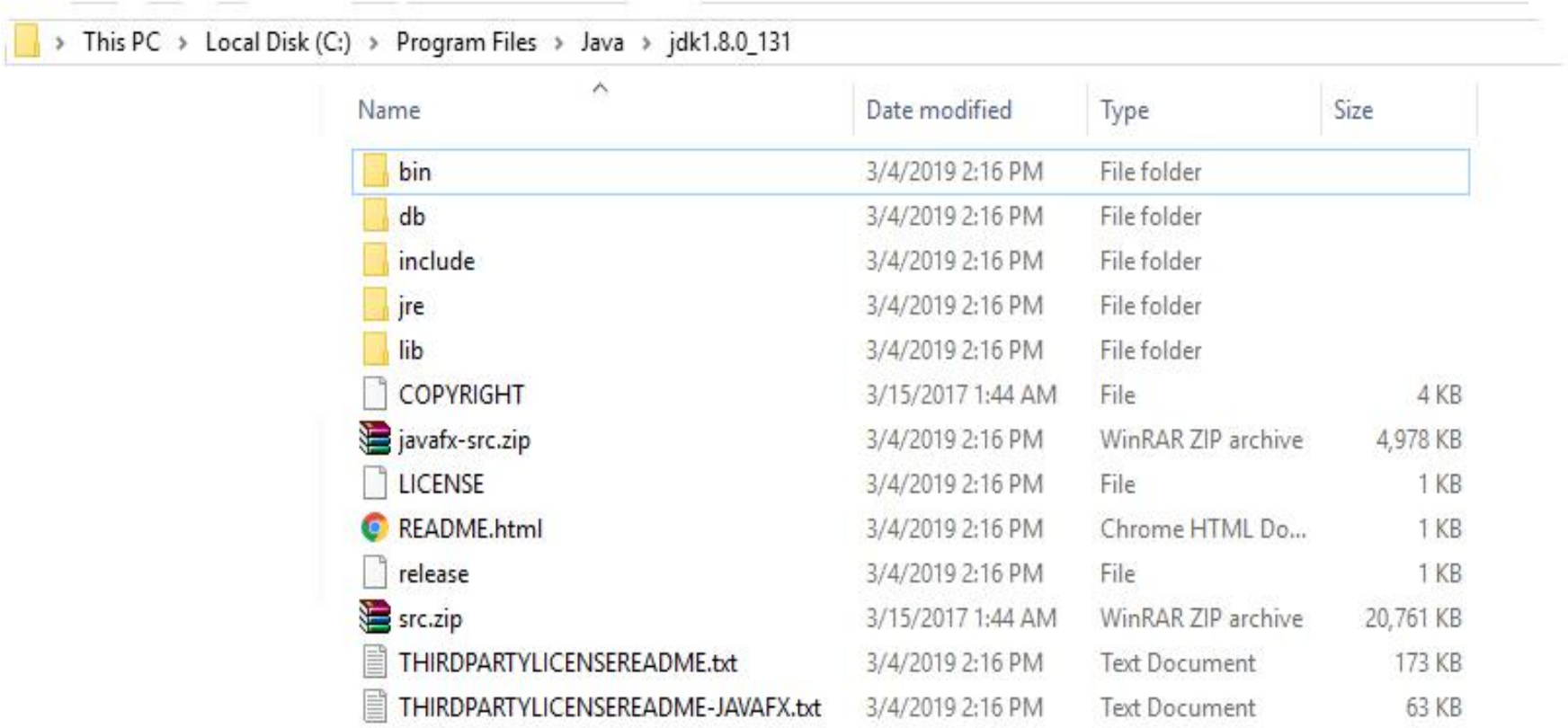
    title of the article: "Comparing Java vs. C/C++ Efficiency Issues to Interpersonal Issues" (Lutz Prechelt)
  - adequate for all but the most time-intensive programs

Installation and setting of JDK for Java programming environment

JDK: Java Development Kit

- JDK is an API that needs to be installed on computers to develop java programs.
- There are different JDKs depending on the platform (Windows, Linux,etc)
- You can download and install stable releases of JDK that fits the operating system of your machine from Oracle's website(Oracle.com)
- Once the JDK setup is installed, you can locate the installation directory of JDK
  - In windows, the JDK installation directory could be something that looks like the following:
  - **C:\Program Files\Java\jdk1.8.0_131**

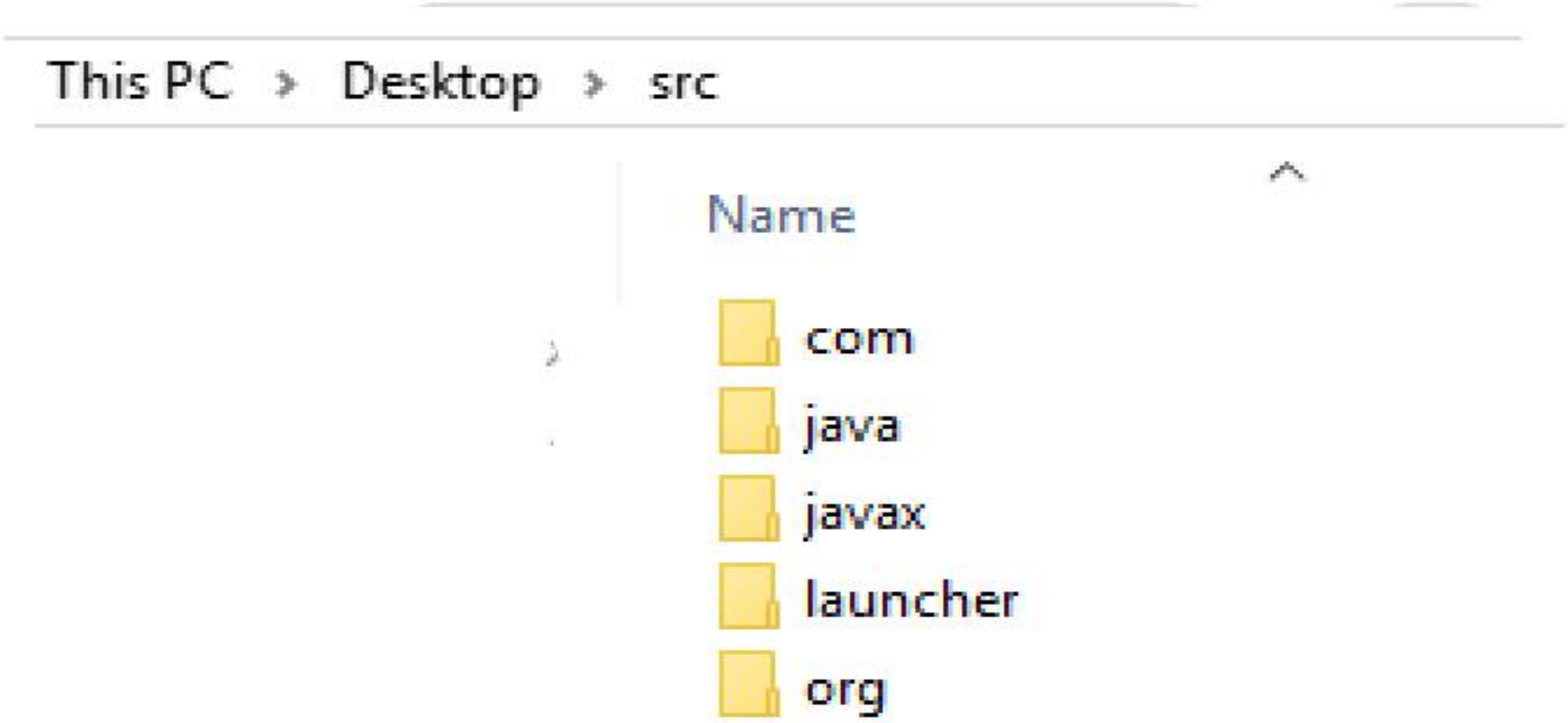
Files/folders under the JDK folder



How could we see the source file of JDK?

- In the snapshoot of the previous slide, you see **src.zip** file that contain all the source files of the JDK API
- You can extract the src.zip file to somewhere in your computer (Desktop, for example) and you will get a folder structure that looks like the one shown in the next slide

When the src.zip is extracted to Desktop



Example: accessing source file of java.util.Date

- Open the java folder from the folder structure you got when you extract the src.zip
- Then, open the **util** folder again
- The, open the **Date.java** file using Notepad or other tool
- Similarly, you can explore and open other source file elements of the JDK

Setting JAVA_HOME

- So far you learnt
  - how to install JDK to a machine
  - how to explore source file of JDK
- After installation of JDK, you need to set JAVA_HOME environment variable before writing java programs and be able to compile and run java programs.
- An environmental variable is a variable that could be mapped with a directory whose contained commands could be executed from the command line regardless of the directory in which the commands are called.

Setting JAVA_HOME…

- Open Environmental Variables window by following either of the following steps (at least in windows 10)

#1:

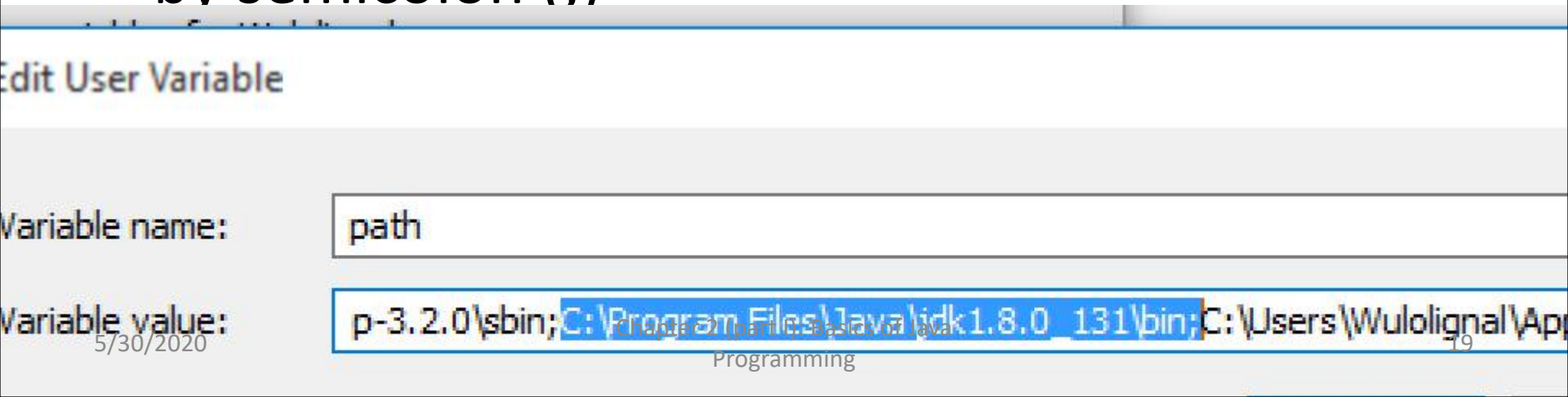
  - Open **control panel** and choose **System and Security** followed by **System**
  - Clicked **Advanced System Setting**

#2: type environmental variable in the search window space

  - Click **Environmental Variables**
  - In the environmental variable dialog box (under user variable), click **New**
  - In the New User Variable dialog box, use **JAVA_HOME** for variable name and use the **installation directory of JDK** (for example, C:\Program Files\Java\jdk1.8.0_131 in my case) for the Variable value

Setting **PATH** environmental variable

- Open environmental variable
- Select path from the list of variables and click the edit button
- Add the {JAVA_HOME/bin} directory to the list of the path variables
- Note that each list element must be separated by semicolon (;)



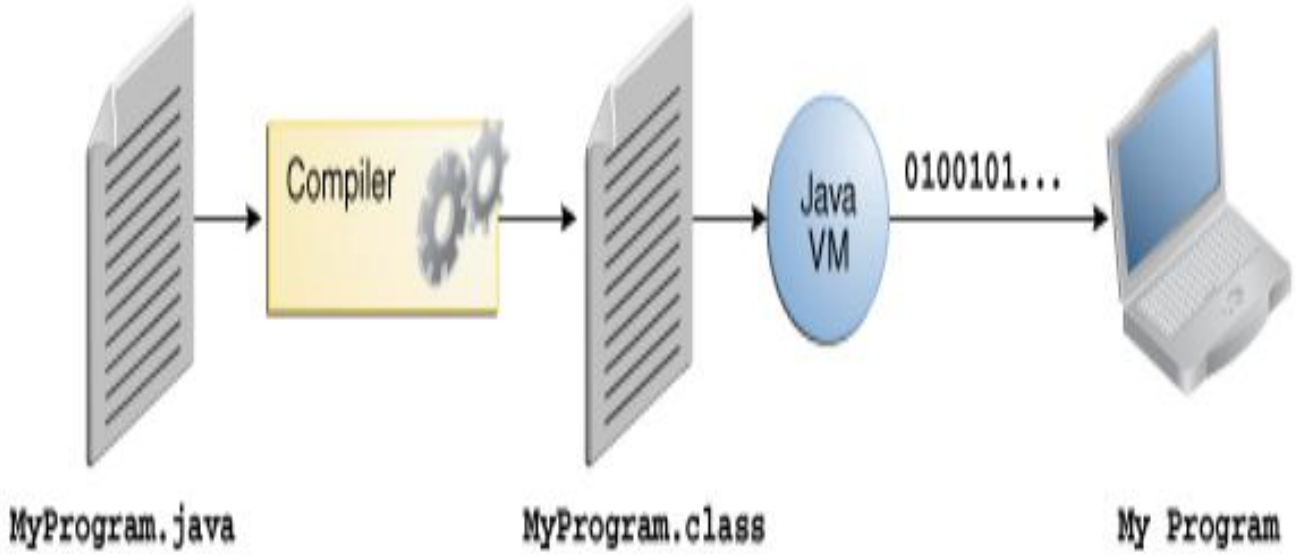
Check proper installation of JDK

- After setting the JAVA_HOME environment variable, you could write the following small java program (and save by the class name.java)and try compiling and executing the program using the **javac** (java compiler) and **java** (java interpreter)command
- By the way, javac and java are commands that are found in the JDK directory, for example in C:\Program Files\Java\jdk1.8.0_131\bin, in my case.

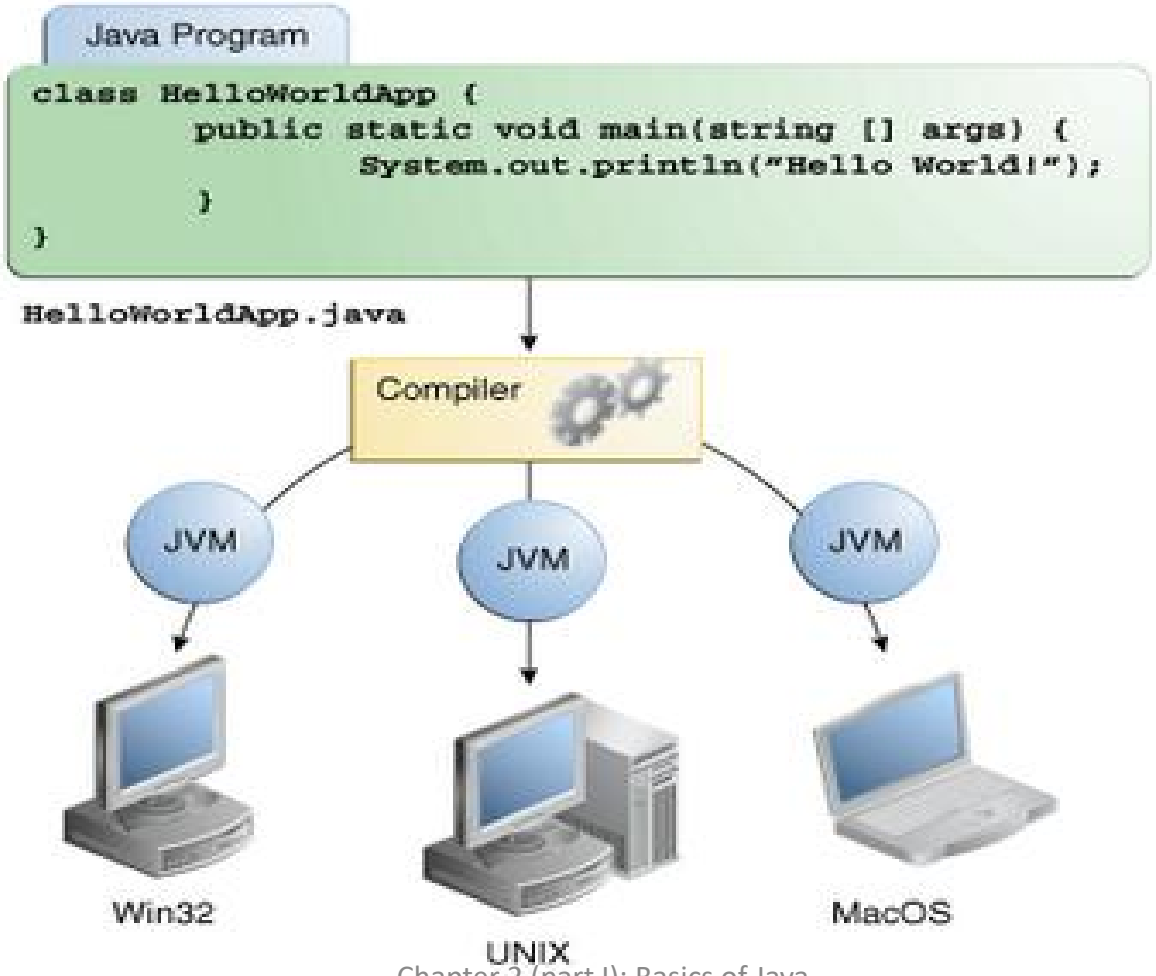
JVM, java source file and Java Byte code

Java Development Process

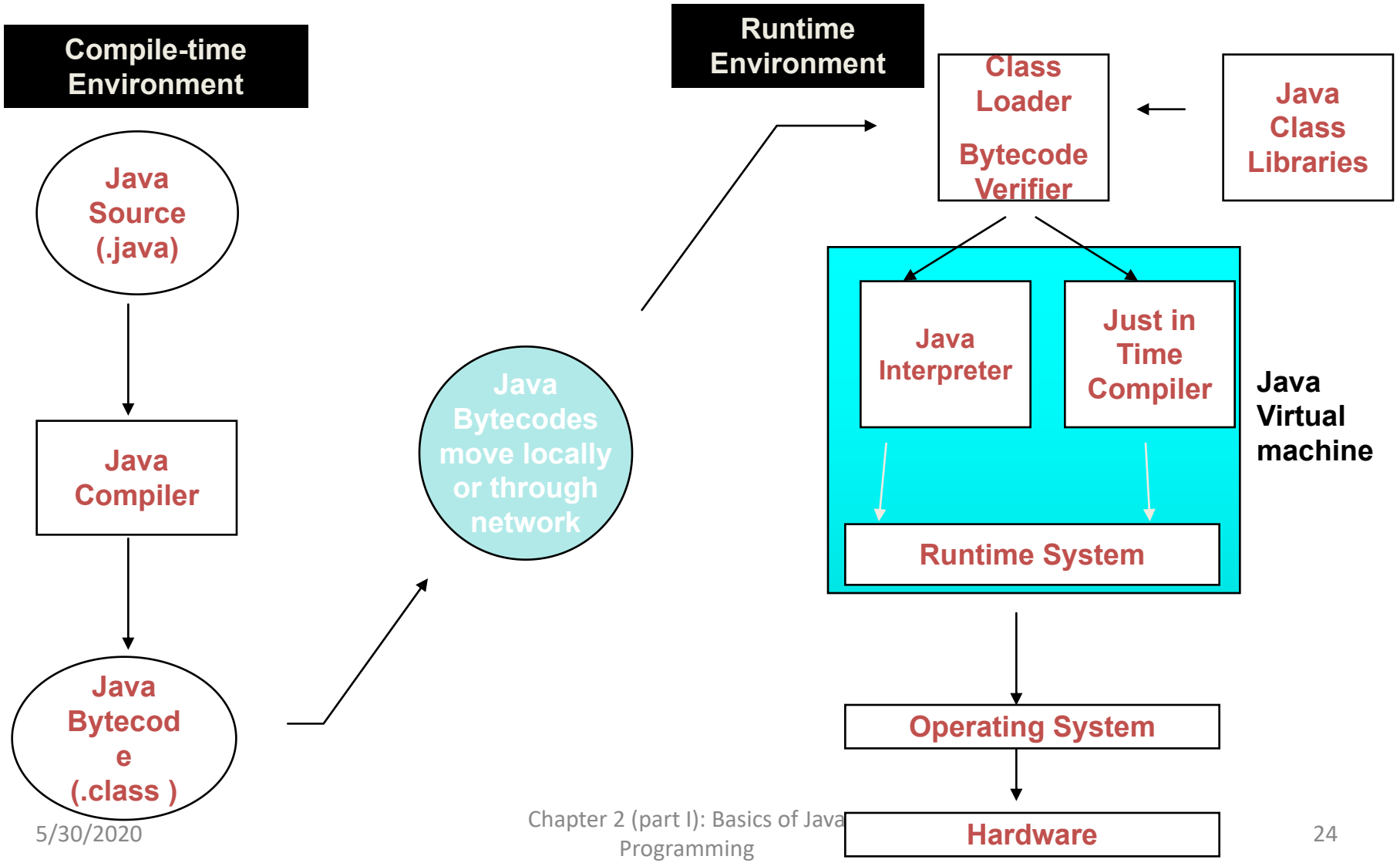
.java => .class => JVM execution



Java Development Process



Java Environment/ Life Cycle of Java Code



Example Java code

```
public class OurFirstJavaProgram {

    public static void main(String[] args)
    {
        System.out.println("This is your first java program");
    }
}
```

Compiling the example code

- The compiler tool, javac, is used to compile java source files (.java files)
- Syntax: javac {class to be compiled}
- For this specific example, the source file is named OurFirstJavaProgram.java (as the class name of the java program is OurFirstJavaProgram)
- Thus, we need to issue javac OurFirstJavaProgram to compile the java source file

Compiling the example code…

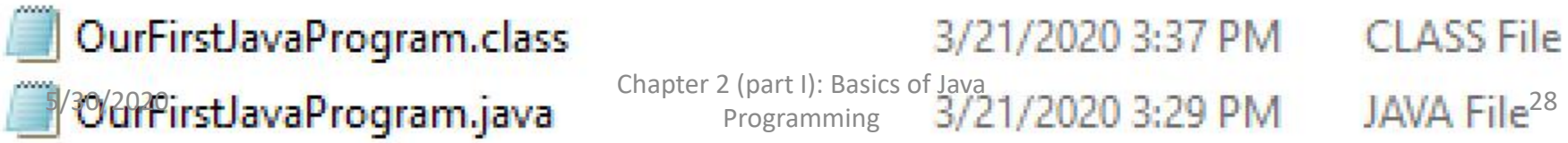
```
C:\Users>javac C:\Users\[redacted]\Desktop\OOP\OurFirstJavaProgram.java
```

- Note: We could change directory to move to the folder that contain the source file to be compiled or we need to specify the absolute path (full path) of the source file to be compiled
- We chose the second alternative in the above example
- If we change directory to OOP folder, the following command could be issued to compile the source file

```
C:\Users\[redacted]\Desktop\OOP>javac OurFirstJavaProgram.java
```

Compiling the example code…

- If you get a message like *"javac is not recognized as an internal or external command,operable program or batch file",* when you try to compile the source file, it means that the javac is not known. The cause of this error could be incorrect JAVA_HOME setting or missing JAVA_HOME
- Upon successful compilation of the source file, you will notice that a file with filename extension of .class is added under the directory that contain the source file.

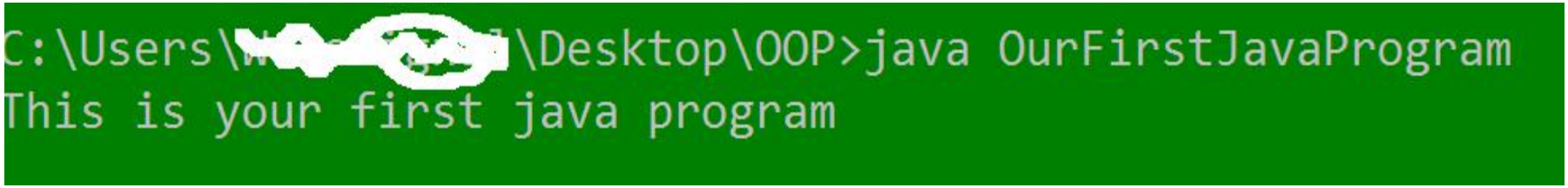


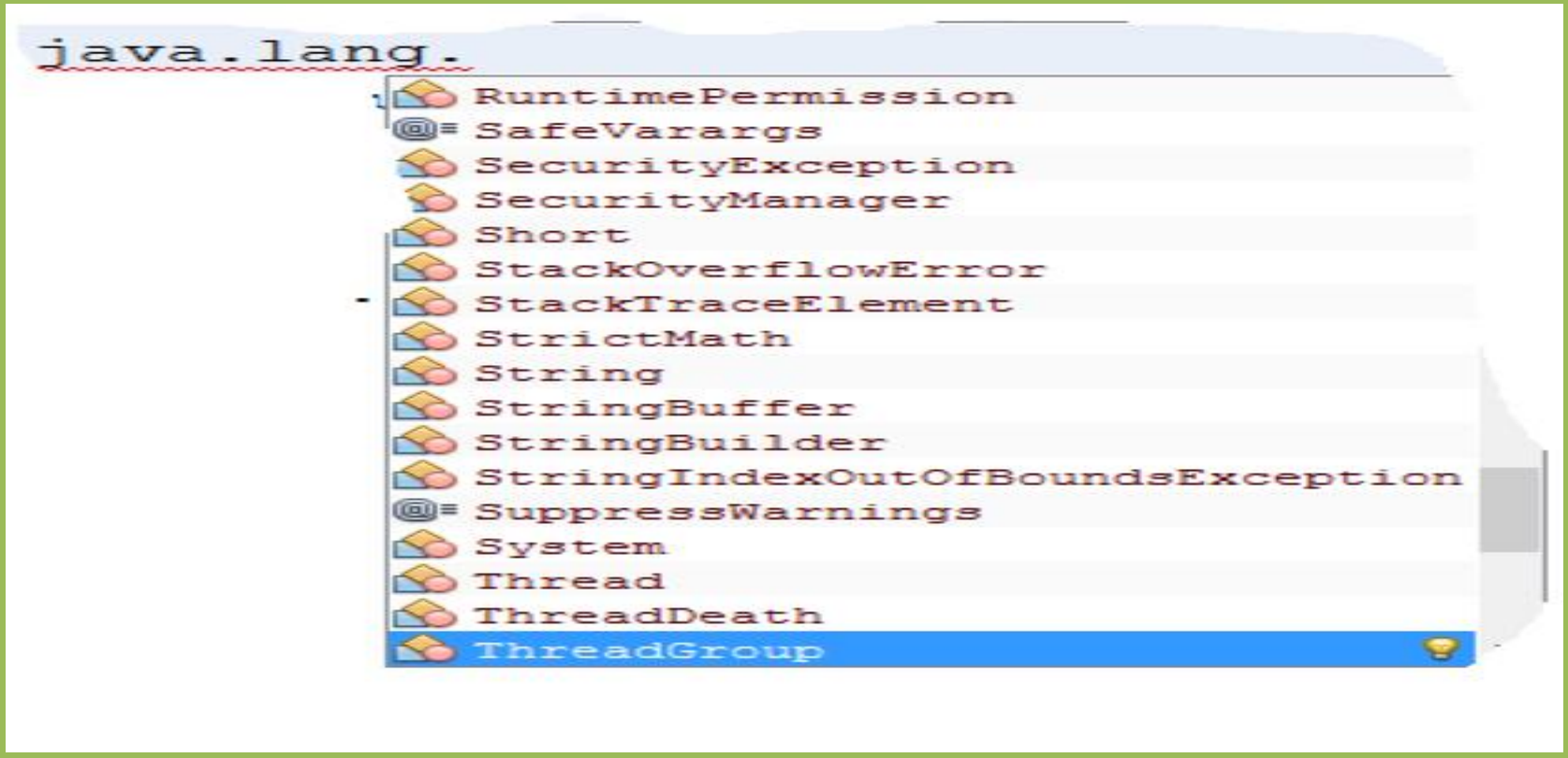
Executing compiled code

- Once a java source file is compiled, the .class file (java byte code) is machine independent and can be executed in any platform.
- This is because of the Just In Time (JIT) interpreter that is provided by the Java Virtual Machine (JVM) of JDK
- In Windows OS, the JIT interpreter converts the .class file to a machine code form that Windows understands and the same is true for other platforms
- And, the program starts from main function of the entry class and continues execution

Executing compiled code…

- Java is a command that is used to interpret the java byte code
- Syntax: java {name of class file without the .class extension}
- For the example we saw above,





The *java.lang* Package

Packages of JDK

- JDK (Java platform) is organized in packages.
- A package could contain classes, interfaces or other programming components.
- The main packages of Java Programming includes:
  - **java.lang** (we focus on this package in the next few slides)
  - java.util
  - java.io
  - java.awt
  - java.awt.image
  - java.applet
  - java.net

The *java.lang* package

- *java.lang* package is one of the main packages of java programming.
- It contains the basics of java programming language
- As opposed to other packages, *java.lang* packages is automatically imported
  - No need of importing the *java.lang* package in a java program

Classes under *java.lang* package

- **System**: The System class contains several useful class fields and methods. It cannot be instantiated.
- Among the facilities provided by the System class are:
  - standard input
  - standard output
  - Standard error output streams
  - access to externally defined properties and environment variables
  - a means of loading files and libraries
  - a utility method for quickly copying a portion of an array

Classes under *java.lang* package…

- Wrapper classes for primitive data types
  - Boolean, Character, Double, Float, Integer, Short, Long, Byte, etc
  - Helps to change the primitives to java objects

```
boolean isValid= true;
Boolean isValid2= new Boolean(isValid);
System.out.println("Class of iSValid2"+isValid2.getClass());
```

```
char ch1 = 'A';//primitive char data
Character ch2= new Character(ch1);//equivalent object of ch1
```

```
Integer i= new Integer(12);
System.out.println("Double form of 12: "+i.doubleValue());
```

Classes under *java.lang* package…

- String
  - The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
  - Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:
  - String str = "abc"; is equivalent to:
    - char data[] = {'a', 'b', 'c'};
    - String str = new String(data);
- Object: to be discussed in the next section of this chapter

Classes under *java.lang* package…

- Class<T>:
- Instances of the class **Class** represent classes and interfaces in a running Java application
- **Class** has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine as classes are loaded

```
OurFirstJavaProgram ofp = new OurFirstJavaProgram();

System.out.println("The number of methods of  the class is: " + ofp.getClass().getName(
```

```
Class x= Class.forName("java.util.Date");
System.out.println("X: "+x);
```

Classes under *java.lang* package…

Math: It contains methods for performing basic numeric operations such as the elementary exponential, absolute value, logarithm, ceiling, floor, square root, and trigonometric functions.

```
System.out.println("Cosine of 0:"+Math.cos(0)+"|-4|= "+
        Math.abs(-4)+"\n ceiling of 4.56: "+Math.ceil(4.56)
        "\n Ceiling of 4.12: "+Math.ceil(4.12)+
        "\n Floor of 4.56:"+Math.floor(4.56)+
        "\n Floor of 4.12: "+Math.floor(4.12)+
        "\n exp(1) "+Math.exp(1)+
        "\n ln(e):"+Math.log(Math.E)+
        "\n log10 (1000): "+Math.log10(1000)+
        "\n pow(2,3):"+Math.pow(2, 3));
```

Classes under *java.lang* package…

Throwbale:

- The Throwable class is the superclass of all errors and exceptions in the Java language.
- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
- Similarly, only this class or one of its subclasses can be the argument type in a catch clause.
- For the purposes of compile-time checking of exceptions, Throwable and any subclass of Throwable that is not also a subclass of either RuntimeException or Error are regarded as checked exceptions

Classes under *java.lang* package…

public class **Exception** extends **Throwable**

- The class Exception and its subclasses are a form of **Throwable** that indicates conditions that a reasonable application might want to catch.
- The class Exception and any subclasses that are not also subclasses of **RuntimeException** are *checked exceptions*.
- Checked exceptions need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Classes under *java.lang* package…

public class **RuntimeException** extends **Exception**

- RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.
- RuntimeException and its subclasses are *unchecked exceptions*.
- Unchecked exceptions do *not* need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary

Classes under *java.lang* package…

public @interface **Override**

- Indicates that a method declaration is intended to override a method declaration in a supertype.
- If a method is annotated with this annotation type compilers are required to generate an error message unless at least one of the following conditions hold:
  - The method does override or implement a method declared in a supertype.
  - The method has a signature that is override-equivalent to that of any public method declared in **Object**.

Classes under *java.lang* package…

public interface **Runnable**

- The **Runnable** interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.
- This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

Classes under *java.lang* package…

public interface **Runnable** …

- In addition, Runnable provides the means for a class to be active while not subclassing Thread.
- A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.
- In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.
- This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Classes under *java.lang* package…

public class **Thread** extends **Object** implements **Runnable**

- A *thread* is a thread of execution in a program.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Classes under *java.lang* package…

public abstract class **Enum<E extends Enum<E>>** extends **Object** implements **Comparable**<E>, **Serializable**

- This is the common base class of all Java language enumeration types

Example

```
import java.util.Scanner;
 class OurFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("Some uses of the java.lang.System class");
        Scanner scan = new Scanner(System.in);
        System.out.println("What is your name?");
        String name = scan.next();
        if (name.length() < 3) {
            System.err.println("Opps,Ethiopian Name can not be less than three characters l
        } else {
            System.out.println("Welcome to java, " + name);
            System.out.println("JAVA HOME is: " + System.getenv("JAVA_HOME"));
            System.out.println("java.version: " + System.getProperty("java.version"));
            System.out.println("Environmental Variables:\n" + System.getenv());
            String[] arr1 = {"one", "two", "three", "four"};
            String[] arr2 = {"1", "2", "3", "4"};
            System.arraycopy(arr1, 0, arr2, 0, 3);
            System.out.println("Array 2:" + arr2[0] + " " + arr2[1] + " " + arr2[2] + " " +
            //System.out.println("Array 2:"+arr);
            System.exit(0); }}}
```

the *Object* class

java.lang.Object

- This is one of the classes under the java.lang class
- Object class is a super-class(base class) of all java classes
  - Every java class implicitly extends **Object** class
- What do you think is the output of the following statement?

```
class OurFirstJavaProgram {
    public OurFirstJavaProgram()
    {
    super();
    }
   public static void main(String[] args) {
       OurFirstJavaProgram ofp= new OurFirstJavaProgram();
       System.out.println("The number of methods of the class is: "+ofp.getClass().getMethods().len
```

java.lang.Object…

```
class OurFirstJavaProgram extends Object{
    public OurFirstJavaProgram()
    {
    super();
    }
   public static void main(String[] args) {
```

```
class OurFirstJavaProgram{
    public OurFirstJavaProgram()
    {
    super();
    }
   public static void main(String[] args) {
```

Data Types and Identifiers

- Data types
  - 8 primitive types:
    - boolean, byte, short, int, long, float, double, char
  - Class types, either provided by Java, or made by programmers
    - String, Integer, Array, Frame, Object, Person, Animal, …
  - Array types
- Variables
  - *dataType identifier [= Expression]:*
  - Example variable declarations and initializations:

```
int x;     x=5;
boolean b = true;
Frame win = new Frame();
String x = “how are you?”;
```

```
int[] intArray;
intArray = new int[2];
intArray[0] = 12;
intArray[1] = 6;
Person pArray = new Person[10];
```

Operators

Expressions and Operators

- *Expressions* are statements that return a value
- *Operators* are special symbols that are commonly used in expressions
- Arithmetic and tests for equality and magnitude are common examples of expressions.
  - Because they return a value, you can assign that result to a variable or test that value in other Java statements.
- Operators in Java include :
  - arithmetic, various forms of assignment, increment and decrement, and logical operations.

Arithmetic Operators

Operator	Meaning	Remarks	Example
+	addition	takes two operands	5 + 6
-	subtraction	takes two operands; it also takes one operand when one negates a number	8 - 4; -7
*	multiplication	takes two operands	8 * 4
/	division	takes two operands; integer division results in an integer as any remainder is ignored	9 / 2
%	modulus	takes two operands ; gives the remainder once the operands have been evenly divided	4 % 3 =1; 31%9=4

Assignment Operators

Expression	meaning
x = val	Assign the value *val* to the variable x
x=y=z=val	Assign the value *val* to all the variables x, y and z
x += y	x= x + y
x -= y	x= x - y
x *= y	x= x * y
x /= y	x= x / y

Increment and Decrement

Operator	meaning	Example
y = x++	assigning x to y before x is incremented by 1	x=0; y=x++// x=1 & y=0
y = ++x	Increment x by 1 and assign the new value to y	x=0; Y=++x// x=1 & y= 1
y = x--	assigning x to y before x is decremented by 1	x=5; Y=x--;// x=4 & y=5
y =-- x	decrement x by 1 and assign the new value to y	x=5; y=--x;// x=4 & y= 4

Comparison Operators

Operator	Meaning	Example
==	Equal	x == 3
!=	Not equal	x != 3
<	Less than	x < 3
>	Greater than	x > 3
<=	Less than or equal to	x <= 3
>=	Greater than or equal to	x >= 3

Logical operators

Operator	Meaning	Remark
X && Y	AND	will be true only if both operands tests are also true
X || Y	OR	result in true if either or both of the operands is also true
X ^ Y	XOR	returns true only if its operands are different