Object-Oriented Programming

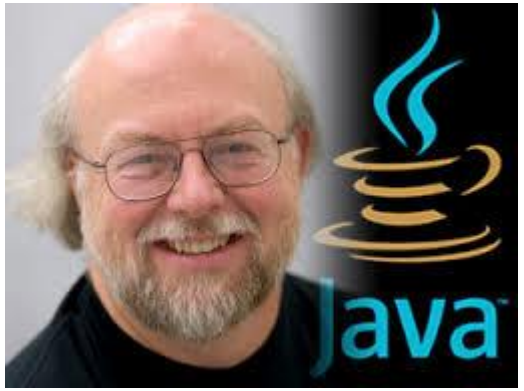
Introduction to Java

Outline

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Java: Brief History & Background



James Gosling

1995, Sun Microsystems

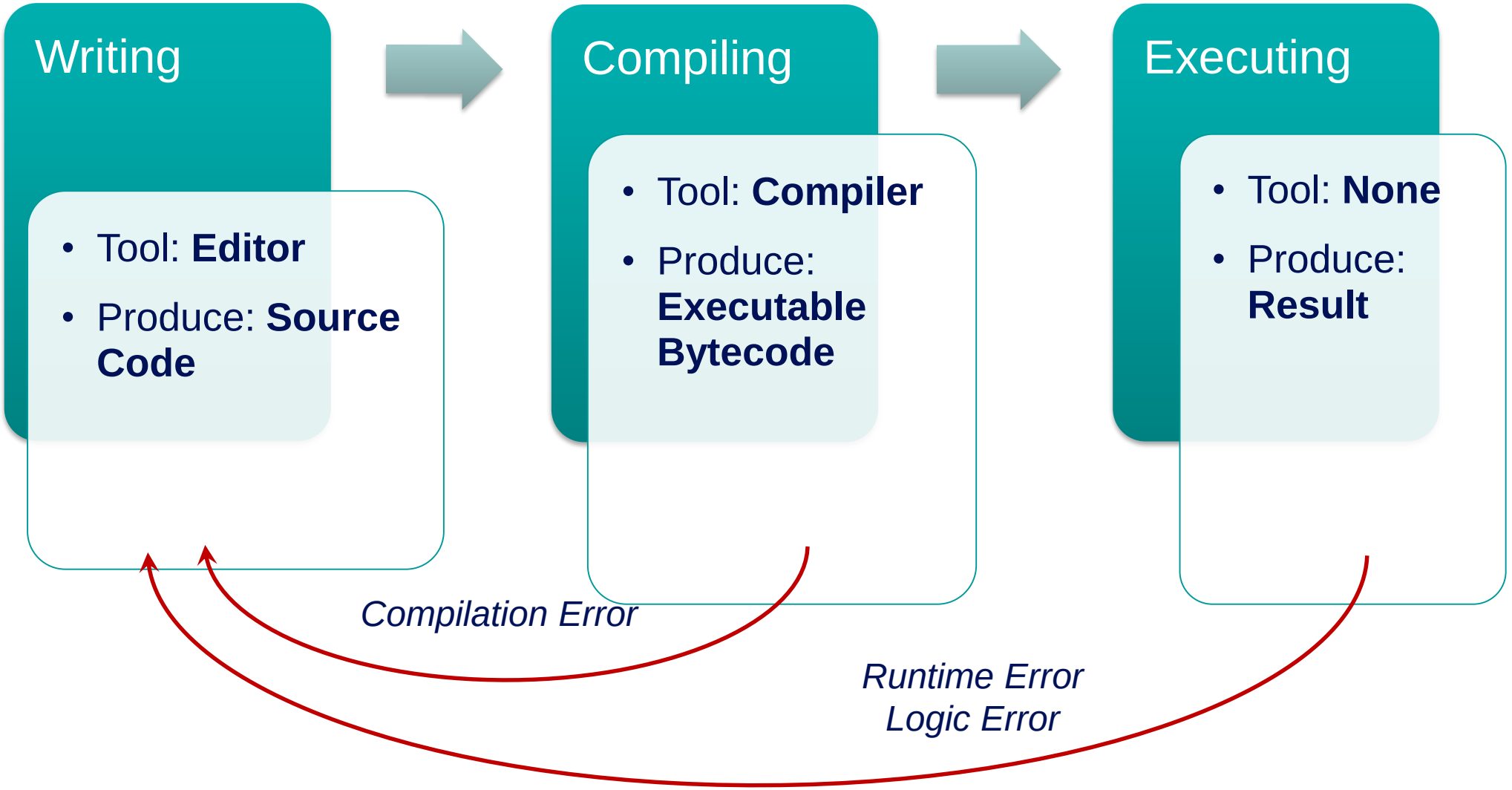
Use C/C++ as foundation

- "Cleaner" in syntax
- Less low-level machine interaction

- Write Once, Run Everywhere™
- Extensive and well documented standard library

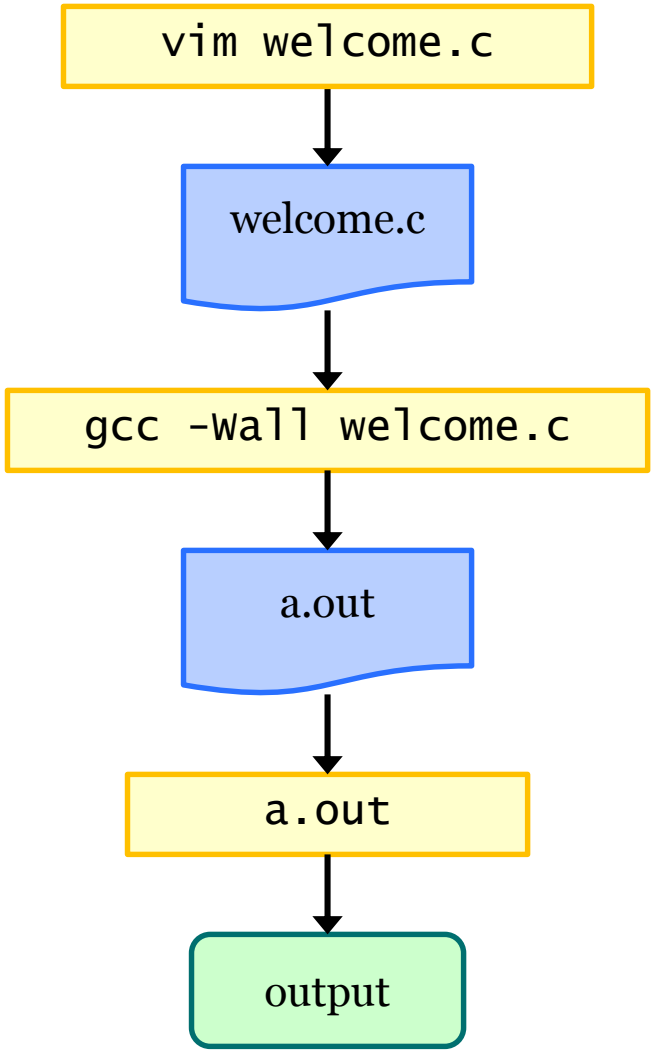
- Less efficient

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions


Writing
Tool: Editor
Produce: Source Code
Compiling
Tool: Compiler
Produce: Executable Bytecode
Executing
Tool: None
Produce: Result
Compilation Error
Runtime Error
Logic Error

Run Cycle for C Programs

- Writing/Editing Program
  - Use an editor, e.g.: vim
  - Source code must have a `.c` extension
- Compiling Program
  - Use a C compiler, eg: **`gcc`**
  - Default executable file: **`a.out`**
- Executing Binary
  - Type name of executable file



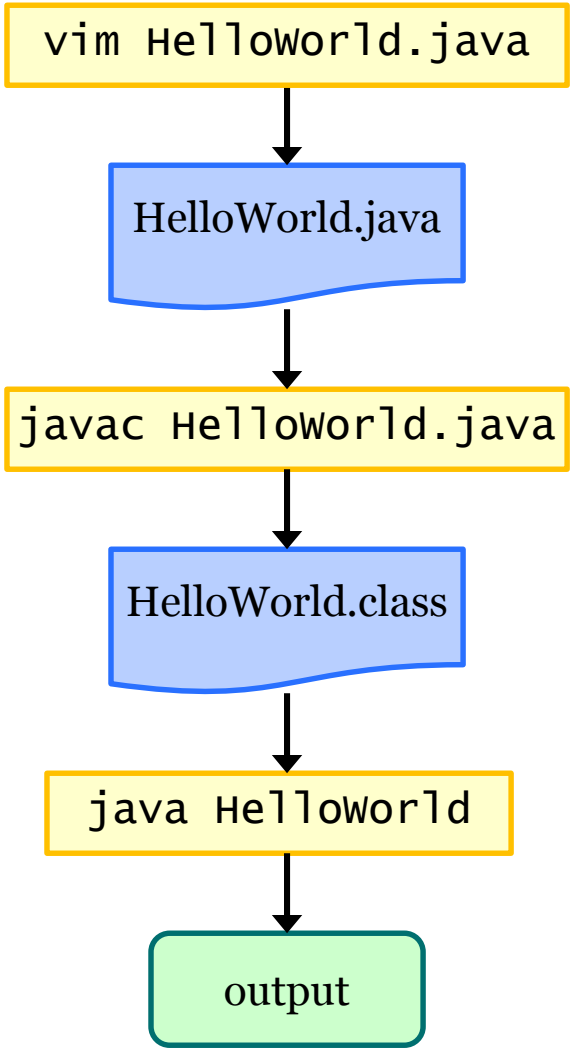
Java: Compile Once, Run Anywhere?

- Normal executable files are directly dependent on the OS/Hardware
  - Hence, an executable file is usually not executable on different platforms
  - E.g: The a.out file compiled on sunfire is not executable on your Windows computer

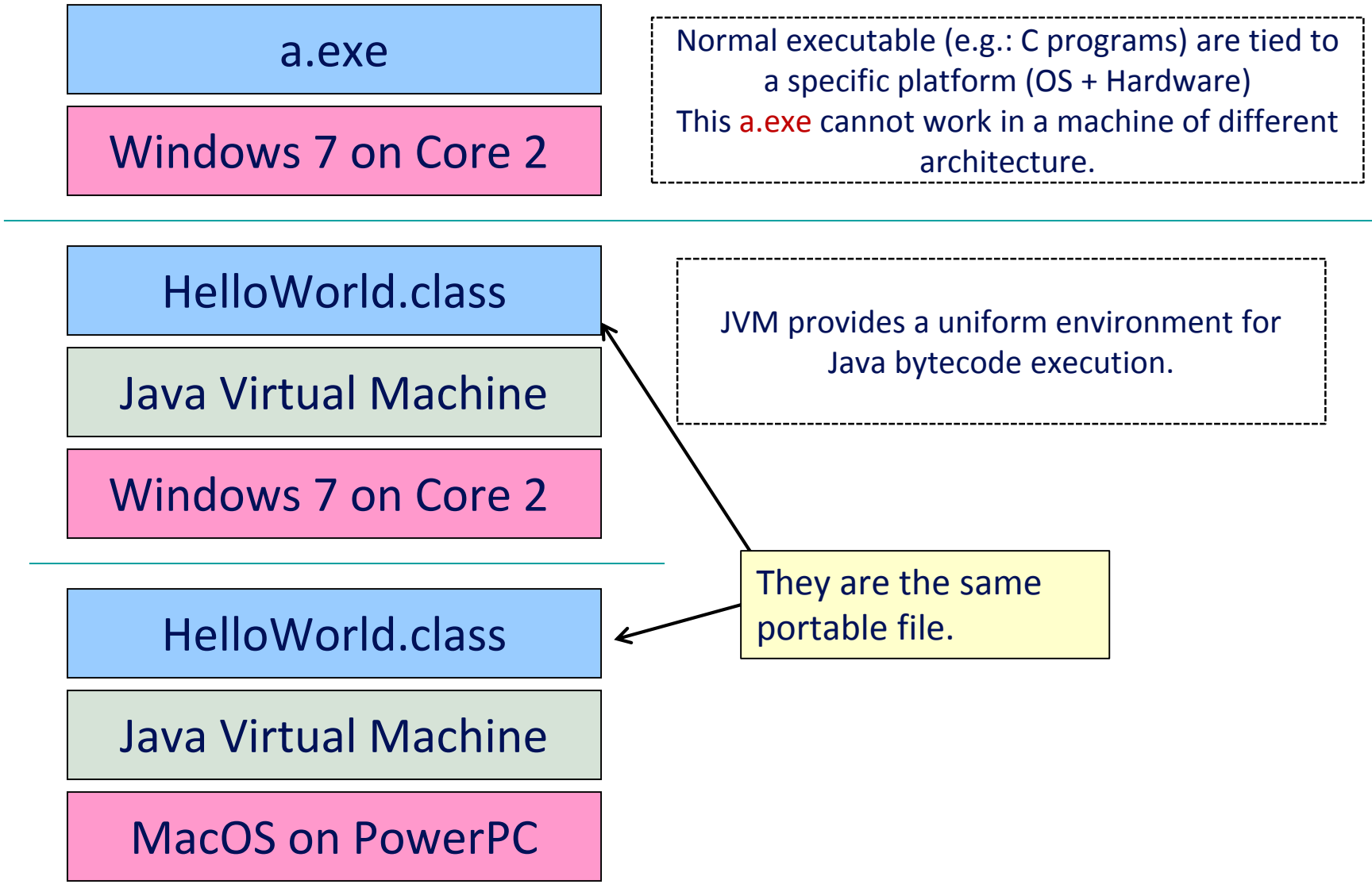
- Java overcomes this by running the executable on an uniform hardware environment simulated by software
  - The hardware environment is know as the Java Virtual Machine (JVM)
  - So, we only need a specific JVM for a particular platform to execute all Java bytecodes without recompilation

Run Cycle for Java Programs

- Writing/Editing Program
  - Use an text editor, e.g: vim
  - Source code must have `.java` extension
- Compiling Program
  - Use a Java compiler, e.g.: **`javac`**
  - Compiled binary has `.class` extension
  - The binary is also known as **Java Executable Bytecode**
- Executing Binary
  - Run on a ***Java Virtual Machine (JVM)***
    - e.g.: **`java HelloWorld`**
      (leave out the **.class** extension)
  - Note the difference here compared to C executable



Java Execution Illustration



1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

- Today: just the basic language components:
  - Basic Program Structure
  - Primitive data types and simple variables
  - Control flow (selection and repetition statements)
  - Input/output statements
- Purpose: ease you into the language
  - You can attempt to “translate” some simple C into Java
- We will gradually cover many other Java features over the next few weeks

Hello World!


```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

HelloWorld.c


```java
import java.lang.*;  // optional

public class HelloWorld {

  public static void main(String[] args) {

     System.out.println("Hello World!");

  }

}
```

HelloWorld.java

Beginners' common mistake:
Public class name not identical to program's file name.

Key Observations (1/2)

- Library in Java is known as **package**
  - Packages are organized into hierarchical grouping
  - E.g., the "**`System.out.println()`**" is defined in the "**`java.lang.System`**"
    - i.e. "**`lang`**" (language) is a package under "**`java`**" (the main category) and "**`System`**" is a class under "**`lang`**"
- To use a predefined library, the appropriate package should be **imported**:
  - Using the "**`import XXXXXX`**;" statement
  - All packages under a group can be imported with a "**`*`**" (the wildcard character)
- Packages under "**`java.lang`**" are imported **by default**
  - Hence, the import statement in this example is optional

- The main() method (function) is now enclosed in a "**class**"
  - More about class will be explained in next lecture
  - There should be only one ***main()*** method in a program, which serves as the execution starting point
  - A source code file may contain **one or more classes**
    - There are restrictions which will be explained later – this is a bit too advanced at this point
    - For the moment, we will restrict ourselves to one class per source code
  - Each class will be compiled into a separate **`XXXX.class`** bytecode
    - The "**`XXXX`**" is taken from the class name ("**`HelloWorld`**" in this example)

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
   - 4.1 Arithmetic Expressions
   - 4.2 Control Flow Statements and Logical Expressions
   - 4.3 Basic Input (Scanner class) and Output
   - 4.4 API
   - 4.5 Math class, Class Attributes
   - 4.6 Object References
   - 4.7 User-defined Functions

Identifier, Variable, Constant (1/2)

- **Identifier** is a **name** that we associate with some program entity (class name, variable name, parameter name, etc.)
- Java Identifier Rule:
  - May consist of letters (‘a’ – ‘z’, ‘A’ – ‘Z’), digit characters (‘0’ – ‘9’), underscore (_) and dollar sign ($)
  - Cannot begin with a digit character
- **Variable** is used to store data in a program
  - A variable must be declared with a specific data type
  - Eg:

```
int countDays;
double priceOfItem;
```

- **Constant** is used to represent a fixed value
  - Eg: `public static final int PASSING_MARK = 65;`
  - Keyword **`final`** indicates that the value cannot change
- Guidelines on how to name classes, variables, and constants: see website
  - https://google.github.io/styleguide/javaguide.html
  - Class name: UpperCamelCase
    - Eg: **`Math, HelloWorld, ConvexGeometricShape`**
  - Variable name: LowerCamelCase
    - Eg: **`countDays, innerDiameter, numOfCoins`**
  - Constant: All uppercase with underscore
    - Eg: **`PI, CONVERSION_RATE, CM_PER_INCH`**

- Summary of numeric data types in Java:

Integer Data Types

Type Name	Size (#bytes)	Range
byte	1	-2^7 to 2^7-1
short	2	-2^{15} to $2^{15}-1$
int	4	-2^{31} to $2^{31}-1$
long	8	-2^{63} to $2^{63}-1$
float	4	Negative: -3.4028235E+38 to -1.4E-45 Positive: 1.4E-45 to 3.4028235E+38
double	8	Negative: -1.7976931348623157E+308 to -4.9E-324 Positive: 4.9E-324 to 1.7976931348623157E+308

- Unless otherwise stated, you are to use:
  - **int** for integers
  - **double** for floating-point numbers

Higher Precedence

`()`	Parentheses Grouping	Left-to-right
`++, --`	Postfix incrementor/decrementor	Right-to-left
`++, --` `+, -`	Prefix incrementor/decrementor Unary +, -	Right-to-left
`*, /, %`	Multiplication, Division, Remainder of division	Left-to-right
`+, -`	Addition, Subtraction	Left-to-right
`=` `+= -= *= /= %=`	Assignment Operator Shorthand Operators	Right-to-left

- Evaluation of numeric expression:
  - Determine grouping using precedence
  - Use associativity to differentiate operators of same precedence
  - Data type conversion is performed for operands with different data type

- When operands of an operation have differing types:
  1. If one of the operands is **double**, convert the other to **double**
  2. Otherwise, if one of them is **float**, convert the other to **float**
  3. Otherwise, if one of them is **long**, convert the other to **long**
  4. Otherwise, convert both into **int**

- When value is assigned to a variable of differing types:
  - **Widening (Promotion):**
    - Value has a smaller range compared to the variable
    - Converted automatically
  - **Narrowing (Demotion):**
    - Value has a **larger range** compared to the variable
    - **Explicit type casting is needed**

- Conversion mistake:

```
double d;
int i;

i = 31415;
d = i / 10000;
```

Q: What is assigned to **d**?

What's the mistake? How do you correct it?

- Type casting:

```
double d;
int i;

d = 3.14159;
i = (int) d; // i is assigned 3
```

Q: What is assigned to **i** if **d** contains 3.987 instead?

The **(int) d** expression is known as **type casting**

Syntax:

```
(datatype) value
```

Effect:
value is converted explicitly to the data type stated if possible.

- Write a simple Java program Temperature.Java:
  - To convert a temperature reading in Fahrenheit, a real number, to Celsius degree using the following formula:

$$celsius = \frac{5}{9} \times (fahrenheit - 32)$$

  - Print out the result
- For the time being, you can hard code a value for the temperature in Fahrenheit instead of reading it from user

4.1 Solution: Fahrenheit to Celsius

Temperature.java

```
public class Temperature {
  public static void main(String[] args) {
    double fahrenheit = 123.5;
    double celsius = (5.0/9) * (fahrenheit - 32);
    System.out.println("Celsius: " + celsius);
  }
}
```

Output:
Celsius: 50.833333333333336

Compare with C:

```
printf("Celsius: %f\n", celsius);
```

- Notes:
  - **5.0/9** is necessary to get the correct result (what will 5/9 give?)
  - "**+**" in the printing statement
    - Concatenate operator, to combine strings into a single string
    - Variable values will be converted to string automatically
  - There is another printing statement, **System.out.print()**, which does not include newline at the end of line (more in section 4.3)

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
   - 4.1 Arithmetic Expressions
   - 4.2 Control Flow Statements and Logical Expressions
   - 4.3 Basic Input (Scanner class) and Output
   - 4.4 API
   - 4.5 Math class, Class Attributes
   - 4.6 Object References
   - 4.7 User-defined Functions

Program Execution Flow

- Java provides an actual **boolean** data type
  - Store boolean value ***true*** or ***false***, which are keywords in Java
  - Boolean expression evaluates to either ***true*** or ***false***

SYNTAX

```
boolean variable;
```

Example

```
boolean isEven;
int input;
// code to read input from user omitted
if (input % 2 == 0)
  isEven = true;
else
  isEven = false;
if (isEven)
  System.out.println("Input is even!");
```

Equivalent:

```
isEven = (input % 2 == 0);
```

Boolean Operators

	Operators	Description
Relational Operators	<	less than
	>	larger than
	<=	less than or equal
	>=	larger than or equal
	==	Equal
	!=	not equal
Logical Operators	&&	and
	||	or
	!	not
	^	exclusive-or

Operands are variables / values that can be compared directly.

Examples:

```
X < Y
1 >= 4
```

Operands are boolean variables/expressions.

Examples:

```
(X < Y) && (Y < Z)
(!isEven)
```

- In ANSI C, there is no boolean type.
  - Zero means ‘false’ and any other value means ‘true’

```c
int x;
... // assume x is assigned a non-negative value
if (x % 3) {
  printf("%d is not divisible by 3.\n", x);
} else {
  printf("%d is divisible by 3.\n", x);
}
```

- In Java, the above is invalid
- Java code:

```java
int x;
... // assume x is assigned a non-negative value
if ((x % 3) != 0) {
  System.out.println(x + " is not divisible by 3.");
} else {
  System.out.println(x + " is divisible by 3.");
}
```

```
if (boolean expression) {
   ...
} else {
   ...
}
```

- **if-else** statement
  - else-part is optional
- Condition:
  - **Must be a *boolean* expression**
  - **Unlike C, integer values are NOT valid**

```
switch (expression) {
  case 1:
       ...
      break;
  case 2:
  case 3:
       ...
  default:
}
```

- **switch-case** statement
- Expression in **switch()** must evaluate to a value of **char**, **byte**, **short** or **int** type
- **break**: stop the fall-through execution
- **default**: catch all unmatched cases; may be optional

Repetition Statements (1/2)

```
while (boolean expression) {
   ... // body
}
```

```
do {
   ... // body
} while (boolean expression);
```

- Valid conditions:
  - Must be a *boolean* expression
- **while** : check condition before executing body
- **do-while**: execute body before condition checking

```
for (A; B; C) {
   ... // body
}
```

- **A**: initialization (e.g. **i = 0**)
- **B**: condition (e.g. **i < 10**)
- **C**: update (e.g. **i++**)
- Any of the above can be empty
- Execution order:
- **A, B, body, C, B, body, C, …**

- In ANSI C, the loop variable must be declared before it is used in a '**for**' loop

```
int i;
for (i = 0; i < 10; i++) {
    ...
}
```

- In Java, the loop variable may be declared in the initialisation part of the 'for' loop
- In example below, the scope of variable **i** is within the 'for' loop only

```
for (int i = 0; i < 10; i++) {
   ...
}
```

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Interacting with the outside world

PACKAGE

```
import java.util.Scanner;
```

SYNTAX

```
//Declaration of Scanner "variable"
Scanner scVar = new Scanner(System.in);

//Functionality provided
scVar.nextInt();

scVar.nextDouble();

  ......
```

Read an integer value from source System.in

Read a double value from source System.in

Other data types, to be covered later

Reading Input: Fahrenheit Ver 2

TemperatureInteractive.java

```
import java.util.Scanner;  // or import java.util.*;

public class TemperatureInteractive {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.print("Enter temperature in Fahrenheit: ");
    double fahrenheit = sc.nextDouble();

    double celsius = (5.0/9) * (fahrenheit - 32);
    System.out.println("Celsius: " + celsius);

  }
}
```

Reading Input: Key Points (1/3)

- The statement

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

  - Declares a variable "**sc**" of **Scanner** type
  - The initialization "**new Scanner(System.in)**"
    - Constructs a **Scanner** object
      - We will discuss more about object later
    - Attaches it to the standard input "**System.in**" (which is the keyboard)
      - This Scanner object **sc** will receive input from this source
    - Scanner can attach to a variety of input sources; this is just a typical usage

- After proper initialization, a Scanner object provides functionality to read value of various types from the input source

- The statement

```
fahrenheit = sc.nextDouble();
```

  - ***nextDouble()*** works like a function (called method in Java) that returns a double value read interactively
  - The Scanner object **sc** converts the input into the appropriate data type and returns it
    - in this case, user input from the keyboard is converted into a double value

Reading Input: Key Points (3/3)

- Typically, only one Scanner object is needed, even if many input values are to be read.
  - The same Scanner object can be used to call the relevant methods to read input values

- **`System.out`** is the predefined output device
  - Refers to the monitor/screen of your computer

SYNTAX

```
// Functionality provided
System.out.print(outputString);

System.out.println(outputString);

System.out.printf(formatString, [items]);
```

```
System.out.print("ABC");
System.out.println("DEF");
System.out.println("GHI");

System.out.printf("Very C-like %.3f\n", 3.14159);
```

Output:

```
ABCDEF
GHI
Very C-like 3.142
```

- Java introduces **`printf()`** in Java 1.5
  - Very similar to the C version
- The format string contains normal characters and a number of specifiers
  - Specifier starts with a percent sign (%)
  - Value of the appropriate type must be supplied for each specifier
- Common specifiers and modifiers:

%d	for integer value
%f	for double floating-point value
%s	for string
%b	for boolean value
%c	for character value

SYNTAX

***%[-][W].[P]*type**

-: For left alignment
W: For width
P: For precision

- One way to calculate the PI (π) constant:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots\ldots\ldots$$

- Write ApproximatePI.java to:
  1. Ask the user for the number of terms to use for approximation
  2. Calculate π with the given number of terms
  3. Output the approximation in 6 decimal places

ApproximatePI.java

```
import java.util.*; // using * in import statement

public class ApproximatePI {

  public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);

     System.out.print("Enter number of terms: ");
     int nTerms = sc.nextInt();

     int sign = 1;
     int denom = 1;
     for (int i = 0; i < nTerms; i++) {
          pi += 4.0 / denom * sign;
          sign *= -1;
          denom += 2;
     }
     System.out.printf("PI = %.6f\n", pi);
  }
}
```

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots\ldots\ldots$$

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Application Programming Interface

API (1/2)

- The Scanner class you have seen is part of the Java API
  - API: an interface for other programs to interact with a program without having direct access to the internal data of the program
  - Documentation, SE 17: https://docs.oracle.com/en/java/javase/17/docs/api/index.html
  - For Java programmers, it is very important to refer to the API documentation regularly!
- The API consists of many classes
  - You do not need to know all the classes (there are easily a few thousand classes altogether!)
  - You will learn some more classes in this course
- This week reading assignment
  - Read up Scanner class in the API documentation

Scanner (Java SE 17 & JDK 17)

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH: Search

Module java.base
Package java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

`Closeable`, `AutoCloseable`, `Iterator<String>`

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:

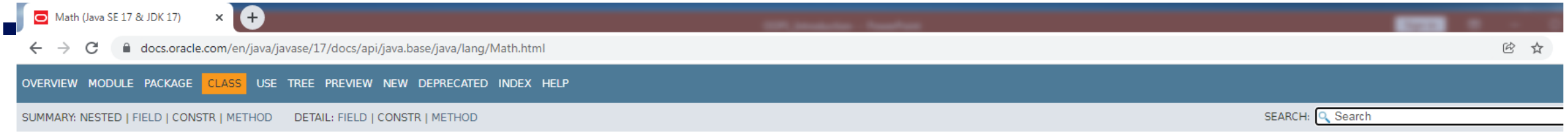
```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Using the Math class

The Math class (1/2)



Module java.base
Package java.lang

Class Math

java.lang.Object
    java.lang.Math

```
public final class Math
extends Object
```

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class StrictMath, all implementations of the equivalent functions of class Math are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the Math methods simply call the equivalent method in StrictMath for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide performance implementations of Math methods. Such higher-performance implementations still must conform to the specification for Math.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point Math methods is measured in terms of *ulps*, units in the last place. For a given flo format, an ulp of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cite worst-case error at any argument. If a method always has an error less than 0.5 ulps, the method always returns the floating-point number nearest the exact result; such a method is *correctly rounded*. A correctly rounded method is generally the floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the Math class, a larger error bound of 1 or 2 ulps is allowed for certain methods. Informally, with a 1 ulp error bo the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be *semi-* whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy automatically meet the monotonicity requirements.

The platform uses signed two's complement integer arithmetic with int and long primitive types. The developer should choose the primitive type to ensure that arithmetic operations consistently produce correct results, which in some cases mean operations will not overflow the range of values of the computation. The best practice is to choose the primitive type and algorithm to avoid overflow. In cases where the size is int or long and overflow errors need to be detected, the methods ad subtractExact, multiplyExact, toIntExact, incrementExact, decrementExact and negateExact throw an ArithmeticException when the results overflow. For the arithmetic operations divide and absolute value, overflow occurs only with a sp minimum or maximum value and should be checked against the minimum or maximum as appropriate.

IEEE 754 Recommended Operations

The Math class (2/2)

- Package: java.lang (default)
- Some useful **`Math`** methods:
  - `abs()`
  - `ceil()`
  - `floor()`
  - `max()`
  - `min()`
  - `pow()`
  - `random()`
  - `sqrt()`

- The **`Math`** class has two class attributes

`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

- A class attribute (or class member) is associated with the class, not the individual instances (objects). Every instance of a class shares the class attribute.
  - We will explain about "objects" later.
- How to use it?
  - Example:

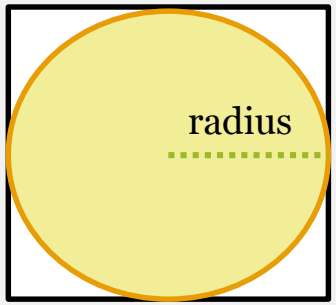
```
double area = Math.PI * Math.pow(radius,2);
```

  - Here, **`Math.PI`** is used as the constant π

The Math class: Demo

TestMath.java

```
// To find the area of the largest circle inscribed
// inside a square, given the area of the square.
import java.util.*;

public class TestMath {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter area of a square: ");
    double areaSquare = sc.nextDouble();

    double radius = Math.sqrt(areaSquare)/2;
    double areaCircle = Math.PI * Math.pow(radius, 2);

    System.out.printf("Area of circle = %.4f\n", areaCircle);
  }
}
```

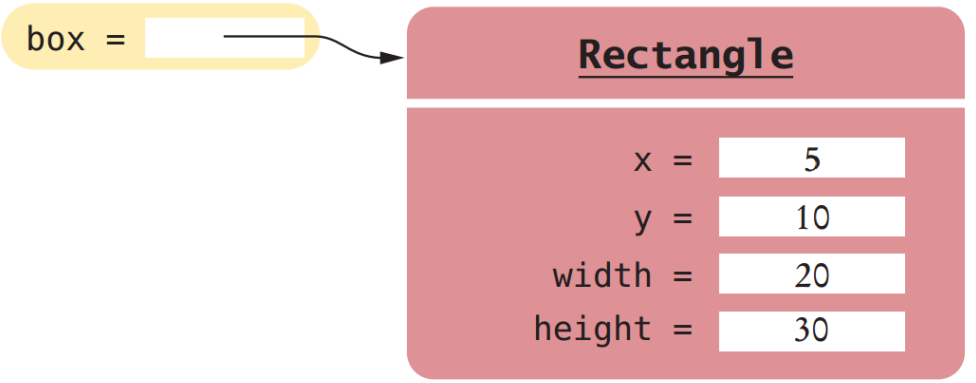


1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Object references

Object references

- In Java, an object variable (that is, a variable whose type is a class) does not actually hold an object.
- It merely holds the *memory location* of an object. The object itself is stored elsewhere.
- There is a reason for this behavior. Objects can be very large. It is more efficient to store only the memory location instead of the entire object.
- We use the technical term object reference to denote the memory location of an object
- When a variable contains the memory location of an object, we say that it refers to an object.
- Technically speaking, the new operator returned a reference to the new object, and that reference is stored in the *box* variable.

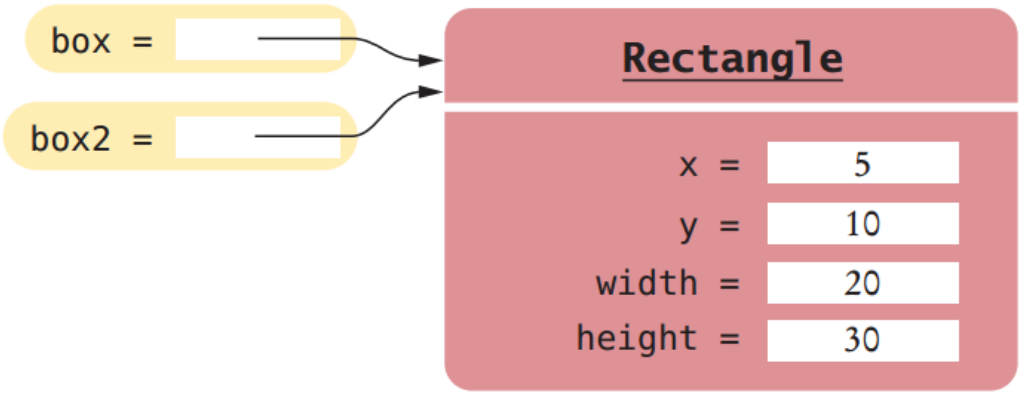



```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Object references

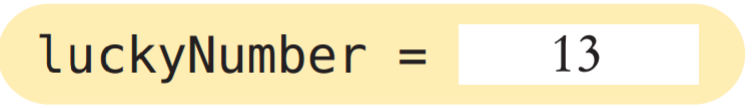
- It is very important that you remember that the *box* variable does not contain the object. It refers to the object.
- Two object variables can refer to the same object

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
```



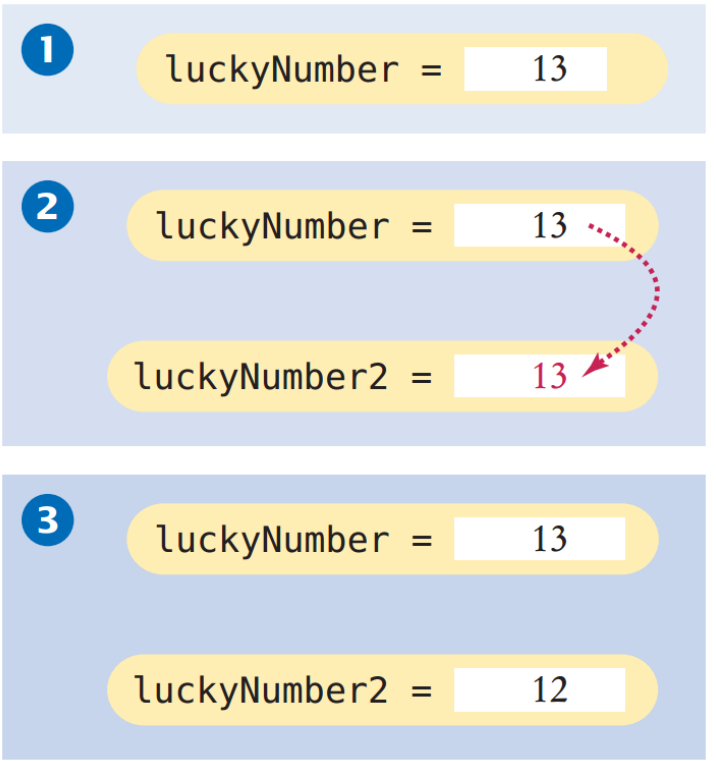
Copy a number

- In Java, numbers are not objects. Number variables actually store numbers.
- When you declare `int luckyNumber = 13;` then the *luckyNumber* variable holds the number 13, not a reference to the number.



- The reason is again efficiency. Because numbers require little storage, it is more efficient to store them directly in a variable.
- When you copy a number, the original and the copy of the number are independent values.

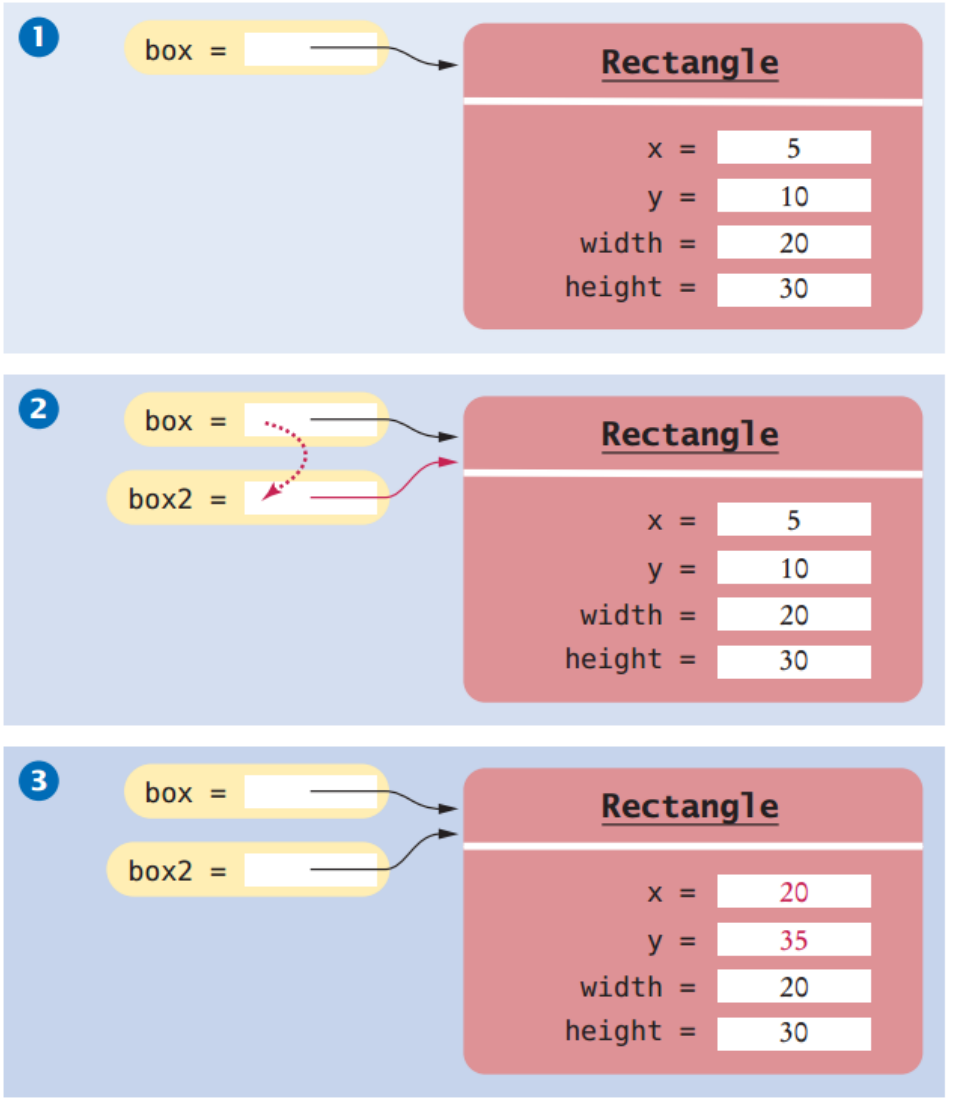
```
int luckyNumber = 13;
int luckyNumber2 = luckyNumber;
luckyNumber2 = 12;
```



- when you copy an object reference, both the original and the copy are references to the same object.

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
box2.translate(15, 25);
```

- Because box and box2 refer to the same rectangle after step 2, both variables refer to the moved rectangle after the call to the translate method.



1. Java: Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
    - 4.1 Arithmetic Expressions
    - 4.2 Control Flow Statements and Logical Expressions
    - 4.3 Basic Input (Scanner class) and Output
    - 4.4 API
    - 4.5 Math class, Class Attributes
    - 4.6 Object References
    - 4.7 User-defined Functions

Reusable and independent code units

- In Java, C-like function is known as **static/class method**
  - Denoted by the "**`static`**" keyword before return data type
  - Another type of method, known as **instance method** will be covered later

Factorial.java

```
public class Factorial {
  // Pre-cond: n >= 0, Returns n!
  public static int factorial (int n) {
      if (n == 0) {
          return 1;
      } else {
          return n * factorial(n-1);
      }
  }

  public static void main(String[] args) {
     int n = 5;  // You can change it to interactive input
     System.out.printf("Factorial(%d) = %d\n", n, factorial(n));
  }
}
```

If *n* is too big, say 40, what will happen? Why?

- All parameters in Java are passed by value (as in C):
  - A copy of the actual argument is created upon method invocation
  - The method parameter and its corresponding actual parameter are two independent variables
- In order to let a method modify the actual argument:
  - **Object reference data type** is needed (similar to pointer in C)
  - Will be covered later

Java Elements

Data Types:

- *Numeric Data Types: byte, short, int, long, float, double*
- *Boolean Data Type: boolean*
- *Characters: char*

Expressions:

- *Arithmetic Expression*
- *Boolean Expression*

Control Flow Statements:

- *Selection Statements: if-else, switch-case*
- *Repetition Statements: while, do-while, for*

Classes:

- *Scanner*
- *Math*

- Java naming convention
  - Class name in UpperCamelCase
    - Eg: “class SumIntegers”, “class Vehicle”, “class GeometricShape”
  - Variable names in LowerCamelCase
    - Eg: “int count”, “double boxHeight”, “char checkCode”
  - Constant names in uppercase, words separated by underscore
    - Eg: “KMS_PER_MILES”, “PI”, “PASSING_MARK”

- This week, the Java programs shown do not truly use object-oriented programming (OOP) features
- We will learn some OOP concepts next weeks

When you see this icon at the top right corner of the slide, it means that in the interest of time the slide might be skipped over in lecture and hence is intended for your own reading.

Thank you!