Unit 2:

Basic Java Programming - 1

Object-Oriented Programming (OOP)

CCIT 4023, 2025-2026

U2: Basic Java Programming - 1

- Basic Java Programming and Java Syntax
- Identifier, Variable, and Data Type
  - Primitive Data Types vs. Reference Types
  - Constant
- Expressions and Statements
  - Simple Numerical Expressions
  - Assignment Statement
- Control Flow – Selection Statement
  - `if` , `if-else`, Conditional Ternary Operator `?:`
  - `switch`
  - Relational and Logical Operators
- Standard (Console) Input / Output

**Remark:* This unit provides a very condensed introduction of starting Java programming, and can much be treated as a revision for learners already with basic programming background.

Basic Java Programming and Java Syntax

- When writing programs, program developers should understand the programming language they use, the available resources such as packages/libraries/modules, and the tools for programing

- The **programming language**, including **syntax** and **semantics**
  - *Syntax*: Rules governing the right form of program codes (like grammar in natural language). E.g.
    - Code `(4023 + 36)` follows the Java syntax (*valid* code)
    - Code `(4023 36 +)` does NOT follow the Java syntax (*invalid*)
  - *Semantics*: Meaning and interpretation of program codes. E.g.
    - Code `(4023 + 36)` means addition of two numbers
    - Code `int oop = 4023;` means assigning a value to a variable

- **Development tools and environments** to build and run the programs, such as JDK or some IDE (Integrated Development Environments).

- ***Resources*** available support writing programs faster, better and easier, including Java standard packages in Standard Edition Java SE

Basic Java Programming and Java Syntax
(Basic Syntax)

- Java is designed with similar **syntax of C++ / C style**
- **Data types** : Java programs supports classes and their objects (OO)
  - Java also supports primitive types (e.g. `int`, `float`, `boolean`)
- **Blocks of codes** `{ }` : Java uses blocks with brace pair `{ }`, e.g. for compound statements, class body, and method body
  - Unlike Python, ***indentation in code lines will be ignored in Java***
- **Declaration** of variables (with data type) : Java variables must be first declared with specified data type (*static typing*), before accessing

```
int aNum; // variable (named aNum) declared as int type
```

- **Statement end** `;` : Statements end with a semi-colon symbol (`;`), e.g.

```
aNum = 123; // an assignment statement
```

- **Program Starts** : Application starts at an entry point `main()` method:

```
public static void main (String [] args)
```

Basic Java Programming and Java Syntax
(Basic Form of Operations and Control Flow)

- Java statements within each execution module (e.g. in the method body) are generally executed ***sequentially*** *from top to the bottom*.
  - There are other ***control flow statements*** which may change the sequential execution, including conditional selection, loop, and branching

- ***Selection*** (Conditional) statements

  `if`, `if-else`, `switch` statements

- ***Repetition*** / Iteration / Loop statements

  `while`, `do-while`, `for` statements

- With ***Branching*** Statements

  `break`, `continue`, `return` statements

Basic Java Programming and Java Syntax
(Typical Java Program)

- **Typical Java program** source file may include:
  - `Comments`: At the top of each Java source file
  - `package` statement: To indicate which package (like a Java library) this program is attached to
    - Without a package statement (esp. for small or temporary developments), an unnamed (default) package is attached
  - `import` statements: To use components in a specific package (package is like a Java library)
  - `class` declaration
- Most Java files essentially ***define Java classes***
  - Each Java class is commonly defined in its own Java file, e.g.
    - Java class `HelloWorld` is defined in the file `HelloWorld.java`
- Proper programming practice also includes
  - Proper commenting, indentation, naming convention, etc.

Typical Java Program & its Components

- Even though whitespaces (and comments) will be ignored by the compiler, ***proper indentations (e.g. 4-spaces or a tab) and comments*** are important for developers and readers

```
// STANDARD, with PROPER indentations (and comments)
public class HelloWorld { // a class HelloWorld, for simple demo
    public static void main(String[ ] args){ // entry-point
        System.out.println("Hello\nWorld!"); // display on console
    }
} // END of class HelloWorld declaration
```

```
// POOR, without indentation (even it compiles and runs)
public class HelloWorld {
public static void main(String[ ] args){
System.out.println("Hello\nWorld!");    }
}
```

All Compile and Run

```
C:\>javac HelloWorld.java
C:\>java HelloWorld
Hello
World!
C:\>
```

```
// VERY POOR, with VERY-POOR indentation (even it compiles and runs)
        public     class
HelloWorld {    public static
                        void main(String[
    ]   args  ){   System.out.println        (
"Hello\nWorld!");       }     }
```

Three Types of Java Comments

- ***Comments*** in program source code are not executable statements, and are ignored by the compiler.
  - Comments are used to provide information or explanation about program codes to readers. It can also be used to "hide" codes.
  - The compiler ignores everything from `/*` to `*/`.
  - The compiler ignores everything from double backslash `//` to the end of the line.

```
/* This is a "multi-line" comment,
   with three lines of text.
*/
```

```
// This is a one-line comment
// This is another one-line comment
// This is a third one-line comment
```

```
/**
 * This is a javadoc comment.
 * JDK javadoc tool uses doc comments for
 * generating HTML API documentations.
 */
```

`import` Statement

- To use or access components in a different specific package (package is a Java library), we often use **`import` statement**:

`import` Statement

Use the imported class `JOptionPane` later

```
// HiWorld.java: A bit complicated Java program
import javax.swing.JOptionPane; // import statement here
public class HiWorld {  // class (named HiWorld) declaration
//...
   JOptionPane.showMessageDialog(null, "Hello World!");
//...
```

- General Syntax: `import` `<package name>` . `<class name>` ;

Package Name
A package is a namespace that organizes a set of related classes (and interfaces).

Class Name
A class name (or other package member), OR an asterisk (*) character to import all in the package

Examples:

```
import     javax.swing.JOptionPane;
Import     javax.swing.JFrame;
Import     java.util.*; // import all classes
Import     java.awt.Component;
```

`import` Statement

- Using a package member (class) from outside its own package, there are 3 ways.
  - Below shows how to use the class `JOptionPane`, outside its package `javax.swing`

1) With an import statement, to import a specific member (class):

```
import javax.swing.JOptionPane;
```

2) With an import statement, to import the entire package:

```
import javax.swing.*;
```

3) ***Without*** *an import statement*, to refer to a member *directly* by Its full qualified name (with the package)

```
javax.swing.JOptionPane.showMessageDialog(
   null, "Hello World!");
```

** Remark: This is convenient for the cases we use the class only once or twice.*

Class Declaration
(Example Class **`Sample1`**)

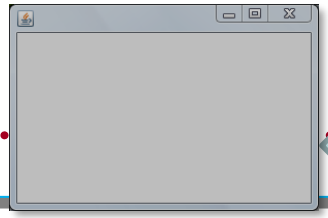
- **Class Declaration** in a Basic Form Syntax:

```
<modifier(s)> class <class name> {   <class body>   }
```

```
/*    Sample Program: Display a Window
      File: Sample1.java
*/
import javax.swing.*;
public class Sample1 {
    // 1. fields (data members/attributes),
    // 2. constructors,
    // 3. methods

    public static void main(String[ ] args) {
      JFrame myWindow; // declare object

      myWindow = new JFrame( ); // create & assign object

      myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      myWindow.setSize(300, 200); // set size of window

      myWindow.setVisible(true);
    }
}
```

Class Declaration

Class Body, may include:
1. **Fields,**
2. **Constructors,**
3. **Methods**



`Sample1.java`

Compile and Run `Sample1`

Method: An Execution Module

- **Method** is a program "module" containing statements to perform operations for specific tasks.
  - *Nature of method in Java is similar to "function" in C or Python*
  - When a method is called (from a caller), the method executes its method body (in general from top to bottom), finishes, and then returns control to where it is called
  - Method may accept input parameters from the caller and may return a value back to the caller

- The specific **`main()`** method (below) acts as the ***execution entry point*** of an application program.

```
public static void main(String[] args){
//..
```

- Java Virtual Machine (JVM) starts up a Java program, by searching and invoking only this specific `main()` method of the class, e.g.
  - Run the HelloWorld program is basically executing the `main()` method of the class `HelloWorld` (in bytecode file `HelloWorld.class`)

Method and Its Body

- Method Basic Form (Syntax):

```
<modifier(s)> <return type> <method name> ( <parameter(s)> ) {
        <method body>
}
```

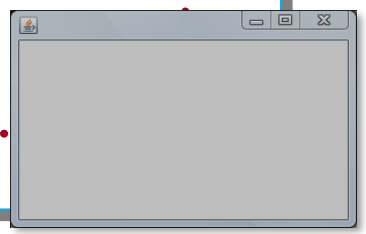
Method

* This is an important `main()` Method, the entry point of execution.

Method Body

* Similar to C's function the method body includes statements which are sequentially executed from top to bottom in general, when the method is called.

```
import javax.swing.*;

public class Sample1 {

    public static void main(String[ ] args) {
        JFrame myWindow; // declare object

        myWindow = new JFrame( ); // create & assign object

        myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        myWindow.setSize(300, 200); // set size of window

        myWindow.setVisible(true);
    }
}
```



Java Programs and Source Codes
("Hello World" Sample Program)

- To start learning Java programming, let us have a look at the Java codes of a very simple "Hello World" program:
  - This simple Java program code displays the string message on the standard output (console) when it is executed/run.
  - Compile the source code (in file `HelloWorld.java`) declaring a class `HelloWorld` creates the bytecode of this class (file `HelloWorld.class`)
  - Run the HelloWorld program is basically executing the `main()` method of this class `HelloWorld` (in bytecode file `HelloWorld.class`)

```
// A simple program source code, Hello World
public class HelloWorld {
    public static void main(String[ ] args){
        System.out.println("Hello\nWorld!");
    }
}
```

COMPILE, on console

```
C:\>javac HelloWorld.java
C:\>
```

RUN, on console

```
C:\>java HelloWorld
Hello
World!
C:\>
```

- This simple program already includes element sets of 2 basic forms:
  - *Words*: e.g. **`public class static void HelloWorld main`**
  - *Symbols*: e.g. **`{ ( ; ) }`**

Java Programs and Source Codes

- Java file, the program source code, generally includes *a sequence of characters*
  - Java program codes are ***case-sensitive***: `aBC` and `ABC` are different
  - Most Java files essentially define Java classes
- To Java compiler, characters for *WhiteSpaces* and *Comments* will be ignored and discarded
    - WhiteSpaces include series of characters of
      - space, horizontal tab, form feed, and line terminator
      - *indentation* (empty space at the beginning of a line) is a type of whitespace
- Other characters may also form *Keyword, Identifier, or Literal*
  - Keyword is a reserved word predefined by the programming language for specific purpose, and cannot be used as identifier, e.g. `int` and `while`
  - Identifier is word used as a specific name for a variable, method, class
  - Literal represents a (fixed) value of certain type

Identifier, Variable, and Data Type

- An identifier could be a specific name given to a variable, method, class, etc. E.g.: **`myCircle , calArea() , Circle`**
- ***Rules of identifier naming*** in Java
  - A sequence of characters that consists of letter, digits, underscores "`_`", and dollar signs "`$`", BUT must not start with a digit
  - Cannot be a reserved keyword, `true`, `false`, or `null`
    - `true` and `false` are the literals of primitive type `boolean`
    - `null` is the only literal of null reference type, referencing to nothing
  - Invalid identifiers lead to error, if not following these rules.
- A literal is source code representation of a (fixed) value of data: primitive data type, reference type, `String` type, etc. E.g.:

```
123 , "a string message" // literals of int , String
```

- Symbols are often used for governing the syntax of languages, e.g.

```
; // indicate a statement end in Java
{ } // represent a block in Java
```

Keyword / Reserved Word

- 51 character sequences are reserved for use as keywords and cannot be used as identifiers
  - There are extra 10 “restricted keywords” newly introduced for “module” declarations: `exports`, `module`, `open`, `opens`, `provides`, `requires`, `uses`, `with`, `to` and `transitive`. They may be used as identifiers anywhere else in the code.

`abstract`	**`continue`**	**`for`**	**`new`**	**`switch`**
`assert`***	**`default`**	**`goto`***	**`package`**	**`synchronized`**
`boolean`	**`do`**	**`if`**	**`private`**	**`this`**
`break`	**`double`**	**`implements`**	**`protected`**	**`throw`**
`byte`	**`else`**	**`import`**	**`public`**	**`throws`**
`case`	**`enum`******	**`instanceof`**	**`return`**	**`transient`**
`catch`	**`extends`**	**`int`**	**`short`**	**`try`**
`char`	**`final`**	**`interface`**	**`static`**	**`void`**
`class`	**`finally`**	**`long`**	**`strictfp`****	**`volatile`**
`const`*	**`float`**	**`native`**	**`super`**	**`while`**

* *not used;*
** *added in 1.2*
*** *added in 1.4*
**** *added in 5.0*

- **`true`** and **`false`** are not keywords, but rather `boolean` literals.
- **`null`** is not a keyword, but rather the `null` literal.
- **`var`** and **`yield`** are not keywords (**`var`** is an identifier with special meaning as the type of a local variable declaration and **`yield`** has special meaning in a **`yield`** statement.)

Identifiers and Their Naming

- *Conventions of identifier naming* following some popular practices make programs more understandable by making them easier to read, although they are not rules:
  - Use a meaningful description of what the value of a variable stands for or what a specific method does. E.g.
    - `int average;` is better then `int a;` to represent average value
    - `getAverage()` is better then `methodA()` to represent the method to get the average value.
  - Do not use the dollar sign `"$"`, and do not start with underscore `"_"`
  - A class name starts with an Upper-Letter, e.g. **`J`**`Frame`, **`S`**`tring`
  - A constant name has ALL Upper-Letters, e.g. **`BLUE`**, **`PI`**
  - Other names start with Lower-Letter, e.g. **`a`**`Str`, **`g`**`etArea()`
  - Use “CamelCase” style for naming with multiple words: internal words start with capital letters, e.g. **`O`**`ur`**`G`**`reat`**`C`**`lass`, a**`L`**`ong`**`S`**`tr`, get**`B`**`ig`**`A`**`rea()`

Variables and Data Types

- Handling data is essential and important in programming
- **Variables** are used to store data values of specified data type
- Variables in Java must be ***declared*** first, before further processing
  - We need to specify the data type of the variable in declaration, and this data type of a variable could not be changed once declared
  - Their values (of its declared type) can later be assessed and changed
- Syntax for declaring variables (*variable declaration statement*):

  `<data type> <variable name(s)> ;`

  where <`variable name(s)`> is an identifier, or a sequence of identifiers separated by commas, e.g.

```
int x; // variable of a primitive type
String myStr; // var. of a class type, String
double n1, n2, n3; // a sequence of identifiers of same type
// some codes here
double n3; // ERR: Duplicated: variable n3 is already defined.
```

Variables and Data Types

- **Data types** of the Java programming language are divided into two major categories:
  - Primitive types, e.g. `int`, `float`, `double`, `boolean`
  - Reference types, e.g. classes `String`, `HiWorld` (in our example)
- ***Primitive types*** include
  - Numeric types: `byte`, `short`, `int`, `long`, `char`, `float` and `double`
  - Boolean type: `boolean`
- ***Reference types*** include
  - class types, interface types, and array types

Variables and Data Types

- Numeric types include integers and floating points
  - Signed Integers:
    - `byte` (8-bit, `-128` to `127`, inclusive)
    - `short` (16-bit, `-32768` to `32767`, inclusive)
    - `int` (32-bit, `-2147483648` to `2147483647`, inclusive)
    - `long` (64-bit, `-9223372036854775808` to `9223372036854775807`, inclusive)
  - Unsigned integer:
    - `char` (16-bit representing UTF-16 code: `'\u0000'` to `'\uffff'` inclusive, that is, from `0` to `65535`)
  - Floating point number:
    - `float` (32-bit)
    - `double` (64-bit)

Variables and Literal Values

- **Literal** is source code representation of a fixed value of certain type
  - literals are represented directly in your code without requiring computation, e.g.

```
123                    // literals of primitive type int
"a string message" // literals of class String
```

- Variable could be assigned with proper literals, e.g.

```
int decV = 26; // integer number
double dV = 123.4;  // double literal
float fV  = 123.4f; // float literal, ends with f
boolean bV = true;  // boolean literal
char cV = 'C';      // character literal
String aStrV = "a string message" // literals of class String
int[] numArray = {4023,4,0,2,3}; // array of integer elements
```

  - Array is a sequence of elements of the same type (a bit similar to Python list).
  - Elements in array are accessed with index number (starting from 0), similar to how we access elements in Python list.
    - E.g. `numArray[0]` refers to `4023` in the above array, element with index 0
  - *More details on array will be discussed in later unit.*

Declaring and Initializing (Assigning) Variables

- When the declaration is made, memory space is allocated to store the value of the variable based on its data type and the variable can be accessed
- By convention, variable names are in lowercase. E.g. `number`, `numberOne`
- We may also declare a variable and initialize its value in one-line step if we want. E.g.

```
int x = 1;
int y = 2;
String myStr = new String("We LOVE OOP");
```

or

```
int x = 1, y = 2;  // variables of same type
String myStr = new String("We LOVE OOP");
```

Accessing Variables

- Accessing variables could be done by:
  - ***Getting*** */ obtaining* its current value *(* make sure it holds a value)*
  - ***Setting*** */ Assigning* it with a new value
- For example, with variables `x` (of `int` type) and `myStr` (of `String` type) below:
  - Get (obtain) values from **`x`** and **`myStr`** (a `String` object):

```
int abc = x + 123;
String someStr = myStr;
```

  - Set (assign) `x` and `myStr` with new values using assignment operator **=**

```
x = 123;
myStr = new String("We LOVE OOP");
```

More on Accessing Variables of Objects

- In Object-Oriented programming, objects of specific class type are often modeling and representing certain entities with data attributes (***fields***) and operations (***methods***)
- Accessing these fields and methods (method calling) of an object is often done with the ***dot-notation***, below syntax:

```
<object name>.<field name>
<object name>.<method name>(<argument(s)>)
```

- E.g. Below shows examples of accessing a variable (named `myStr`) of `String` object, by calling its methods

```
myStr = new String("We LOVE OOP"); // create a string
myStr.charAt(0); // returns the character at index 0 → 'W'
myStr.length(); // returns length/size of string object → 11
```

** More details in Accessing Objects will be given in later units.*

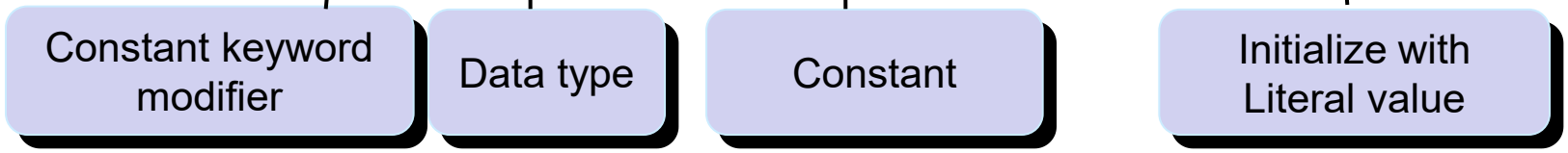
Constants

- **Constant** could be treated as a special variable which can hold data, but *cannot be changed* once initialized.
- By naming convention, constants are represented in all UPPERCASES E.g. `PI` , `MAX_VALUE` , `MONTH_IN_YEAR`
- Benefit: Using a constant can avoid specifying the same value multiple times, and also avoid mistakenly changing the value as variables
- Syntax for constants are similar way as variables, with keyword **`final`**

```
final <data type> <variable name(s)> = <constant value>;
```

- Examples:

```
final double PI              = 3.14159 ;
final int    MONTH_IN_YEAR   = 12      ;
```



Expressions and Statements

- **Expression** is a sequence of variables, operators, symbols and method invocations, that evaluates to a result value
  - 2 examples of numeric expression below:

```
( 9 - 6 ) * 3    // → results a numeric value of 9
aNum * ( 369 + getGrade() )  // → results a numeric value
```

    - suppose `aNum` is a variable of number, `getGrade()` is a method which returns a number, and this expression evaluates to a number value

- **Statement** could be treated a basic unit of execution in Java program, often terminated with a semicolon (`;`), such as simple declaration statements and assignment statements:

```
int courseID;     // declaration statement
couseID = 4023;   // assignment statement
```

  - A group of statements enclosed by a brace-pair `{ }` is called a ***Block***, and is often used to form a compound statement.

Expressions and Statements

```
if (temperature >= 30) {
    System.out.println("It is hot!");
    if (temperature > 36)
        System.out.println("... and VERY hot!!");
}
else // temperature < 30
    System.out.println("It is NOT hot.");
```

- The sample codes above show how a Java program consists of expression, statement and block
  - Example ***Expression*** *(evaluate to* `boolean` ***value****,* `true` ***or*** `false`*)*:
    - `temperature >= 30` , `temperature > 35`
  - Example ***Statement***:
    - `System.out.println("It is hot!");`
  - Example ***Block*** *(as a compound statement)*:
    - The **bolded portion** in the sample codes within brace-pair `{ }`;

Simple Numerical Expression

- Numerical Expression is often formed with numbers and arithmetic operators that evaluates to a numeric value:

`+`	Additive operator
`-`	Subtraction operator
`*`	Multiplication operator
`/`	Division operator
`%`	Remainder (modulus) operator

- Examples: given `x=2; y=4; z=8;`

```
x - y + z          (evaluate to) → 6
x - ( y + z )      (evaluate to) → -10
z * y / x          (evaluate to) → 16
z % x              (evaluate to) → 0
9 % 6              (evaluate to) → 3
```

**Remark: More details will be given in later unit.*

Assignment Statements

- We assign a value to a variable with the assignment operator **=** using an **assignment statement**
- The general syntax is `<variable> = <expression> ;`
  - Expression (the Right-Hand-Side) is evaluated first, the expression result value is then assigned to the variable (the Left-Hand-Side)
- It is common to say:
  - It assigns / sets a (expression) value to a variable.
  - It assigns / sets the variable (with) a value.
- Examples:

```
// direct assignment
        age = 18;
// assignment with expressions (Right-Hand-Side)
        sum = firstNumber + secondNumber;
        avg = (one + two + three) / 3.0;
```

Shorthand Assignment Operators

- Often we may need to update the value of a numerical variable, e.g. adding a new value to the current sum:

```
sum = sum + newValue
```

- We may instead use the shorthand assignment operator as below:

```
sum += newValue;
```

```
sum = sum + newValue;
```

is equivalent to

```
sum += newValue;
```

Operators	Usage	Meaning
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Increment ++ and Decrement -- Operators

- Similar to C, C++ and others, Java supports increment ++ and decrement -- operators, for adding or subtracting one to a variable.

Operators	Usage	Meaning
++	a++	a = a + 1 *or* a += 1
--	a--	a = a - 1 *or* a -=1

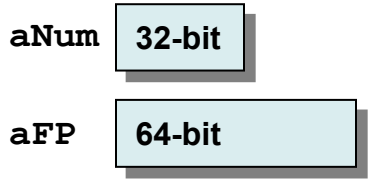
* Increment ++ and decrement -- operators may also precede its operand (comparatively less common), e.g.

++a or --a,

Declaration & Assignment (Primitive Type vs. Reference Type)

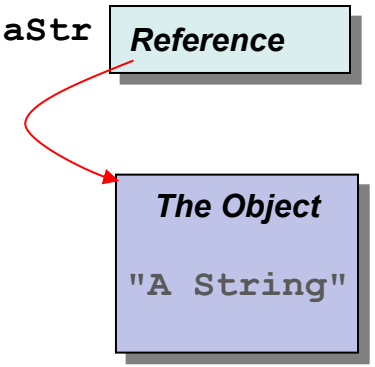
- Declaring *primitive* type data will reserve memory to hold the actual data of that type, e.g.

```
int aNum; // memory reserved for int
double aFP; // memory for double
```



- Declaring *reference* type data will ONLY reserve memory for further referencing actual data of reference type
  - Creating (and assigning) the actual data (e.g. objects of a class) are made separately, e.g.

```
String aStr; // memory for referencing (addr)
aStr = new String("A String");
```

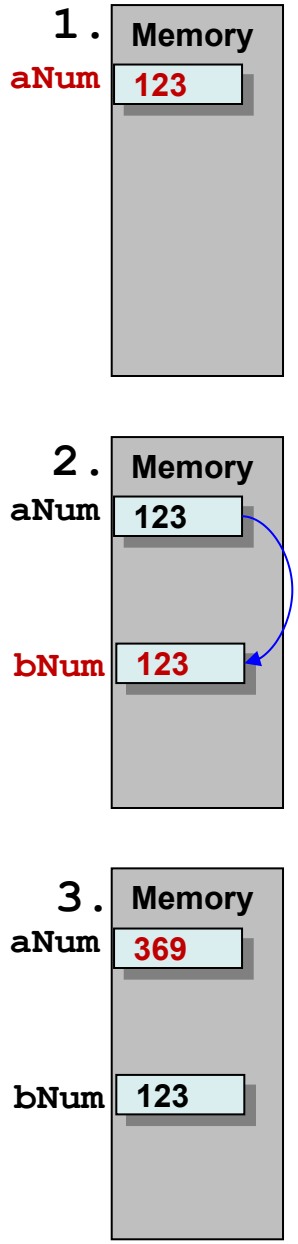


Declaration & Assignment (Primitive Type vs. Reference Type)

- Assigning and reassigning values to ***primitive* type** variable update its *value of specific primitive type* in memory directly

```
int aNum = 123; // 1.
int bNum = aNum; // 2.
aNum = 369; // 3.
```

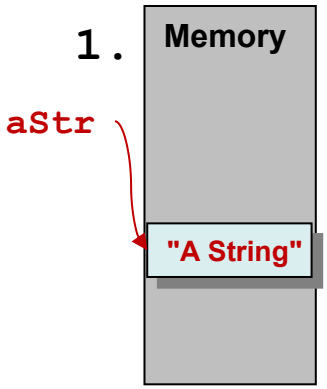
- Example code above:
  1. Declare a new variable `aNum` (primitive type `int`) and assign an `int` value `123` to it
  2. Declare another new variable `bNum` (primitive type `int`) and assign the same value of `aNum` (which is `123`) to it
  3. Reassign a new value to `aNum`, and updates its value to `369`

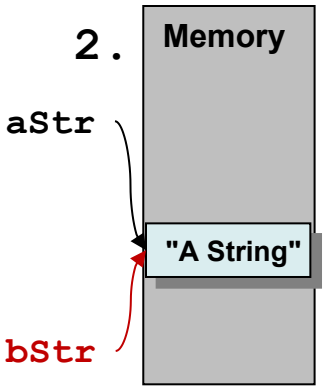


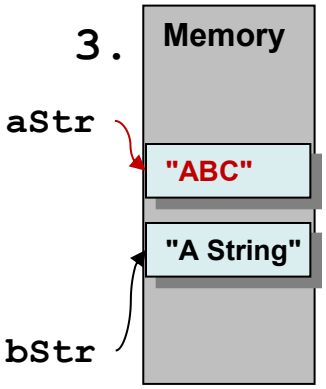
Declaration & Assignment (Primitive Type vs. Reference Type)

- Assigning and reassigning values to ***reference* type** variable update its *value of reference*

```
String aStr = new String("A String"); // 1.
String bStr = aStr; // 2.
aStr = new String("ABC");; // 3.
```







- Example code above:
  1. Declare a new variable `aStr` (reference type class `String`) and assign it a *reference value* to a newly created string object in memory
  2. Declare another new variable `bStr` of same type (`String`) and assign the same *reference* value of `aStr` (*reference to the same object*) to it
  3. Reassign a new value to `aStr`, and updates its *reference value* to another newly created string object in memory

Control Flow - Selection Statements

- Statements are generally executed from top to bottom
- However, control flow statements may break up this execution flow to enable our program to conditionally execute particular blocks of code under different situations, and they are:
  - Selection (or decision-making) statements, including
    - `if`, `if-else`, `switch`
  - Repetition (or looping) statements, including
    - `while`, `do-while`, `for`
  - Branching statements, including
    - `break`, `continue`, `return`

Control Flow - Selection Statements

- Java has three basic types of selection statements:
- The simple **`if`** statement either performs an action (the coming true block) if a condition (`boolean` expression) is `true`, or skips it if the condition is `false`, e.g.

```
if ( <boolean expression> ) {

        <true block>

}
```

* If the true-block has only one single simple statement line, the brace pair **`{}`** (curly brackets) is optional

- The **`if-else`** statement performs an action (the true block) if a condition is true, and performs a different action (the false block) if the condition is false
- The **`switch`** statement performs one of many actions, depending on the matching value of an expression

The Simple `if` Statement

- The `if` statement is the most basic form of control flow
  - It executes a certain section of code (in the true-block), only if a particular condition evaluates to `true`.
  - General Syntax:

```
if ( <boolean expression> ) {
     <if body true-block>
}
<other statements>
```

True-Block:
Statements are executed if `<boolean expression>` is true (`true` condition). Otherwise, they are simply not executed.
* If the true-block has only one single simple statement liine, the `{}` (brace-pair / curly braces) is optional

- Examples for two forms (with or without brace-pair):

```
// CASE 1: Simple statement, NO brace-pair required for one-line body
   if (testScore > 60)
      JOptionPane.showMessageDialog(null, "Good!" );
// CASE 2: Compound statement: brace-pair {} required for multi-lines
   if (testScore > 80) {
      JOptionPane.showMessageDialog(null, "Very Good!" );
      JOptionPane.showMessageDialog(null, "Let's celebrate!" );
   }
```

The `if-else` Statement

- The **`if-else`** statement executes either a certain section in the true-block, or in the false-block.
  - General Syntax:

```
if ( <boolean expression> )
        <true-block>
else
        <false-block>
<other statements>
```

True-Block:
Statements are executed if `<boolean expression>` is true (`true` condition).

False-Block:
Statements are executed if `<boolean expression>` is false (`false` condition).

- Example:

```
if (testScore > 60)
   JOptionPane.showMessageDialog(null, "Good!" );
else {
   JOptionPane.showMessageDialog(null, "Not that good!" );
   JOptionPane.showMessageDialog(null, "Keep working hard!" );
}
```

Relational Operators

- **Relational operators** determine if one operand is greater than, less than, etc. another operand, and produce a `boolean` result (`true` or `false`)

`<`	less than
`<=`	less than or equal to
`==`	equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to

Examples: *(Given* `int aScore = 20;` *)*

```
aScore < 60                          → true
aScore * 2 >= 350                    → false
30 < w / (h * h)
x + y != 2 * (a + b)
2 * Math.PI * radius <= 360.0
```

Logical Operators

- **Logical operators** perform logical-NEGATION on a boolean expression, logical-AND and logical-OR operations on two boolean expressions.
  - All produce a `boolean` result (`true` or `false`)

    `!` (**NOT**, logical negation)

    `&&` (**AND**, logical conjunction)

    `||` (**OR**, logical disjunction)

Examples: (Given `int aScore = 20;` *)*

```
!(aScore < 60)                          → false
(aScore > 30) && (aScore < 60)          → false
(aScore < 30) || (aScore > 60)          → true
```

The Nested-`if` Statement

- The <then> and <else> block of an **`if`** statement can contain any valid statements, including other **`if`** statements
- An **`if`** statement that contains another **`if`** statement is called a nested-**`if`** statement

```
if (testScore >= 60) { // compound statement, with {}
    if (testScore < 80)
       System.out.println("You did pass");
    else
       System.out.println("You did a great job");
}
else //test score < 60
    System.out.println("You did not pass");
```

`boolean` Variables

- **`boolean`** is a primitive data type in Java
- The result of a boolean expression is either `true` or `false`
- Declare a variable of data type `boolean`, and assign a boolean value as below:

```
boolean pass, done;
pass = 60 < x;
done = true;
if (pass) {
        //...
} else {
        //...
}
```

Conditional Ternary Operator `?:`

- The conditional ternary operator `?:` can be thought of as shorthand for an "if-then-else" statement which gives a desired result value
- This conditional ternary operator requires three operands (vs. binary operator requires two operands):
  1) the-boolean-condition, **`condExp`**
  2) value-for-true-condition, **`tVal`**
  3) value-for-false-condition, **`fVal`**
- General syntax: **`<condExp> ? <tVal> : ffVal>`**
  - The code above first evaluates if `condExp` (a `boolean` expression) is `true` or not (thus `false`)
  - If `condExp` is `true`, then the result will be `tVal`
  - Else (it is `false`), the result will be `fVal`

Conditional Ternary Operator ?:

An example of comparing two approaches:

- Using conventional `if-else` statement

```
if (  num1 < num2  )
   smallerNum = num1;
else
   smallerNum = num2;
```

- Using Conditional Operator ?:

```
smallerNum = (num1 < num2 ) ? num1 : num2;
```

1) the-boolean-condition, **condExp**

2) value-for-true-condition, **tVal**

3) value-for-false-condition, **fVal**

Conditional Ternary Operator ?:

Another example of comparing two approaches:

- Using conventional `if-else` statement

```
if (  testScore < 60  )
  JOptionPane.showMessageDialog(null, "You did not pass" );
else
  JOptionPane.showMessageDialog(null, "You did pass" );
```

- Using Conditional Operator ?:

```
JOptionPane.showMessageDialog(null,
(testScore < 60) ? "You did not pass" : "You did pass" );
```

1) the-boolean-condition, **condExp**

2) value-for-true-condition, **tVal**

3) value-for-false-condition, **fVal**

The `switch` Statement

- In case of exact matching of a number of values required, using a long chain of nested `if-else` is tedious.
- Instead, `switch` statement gives a clearer alternative.
- The **`switch` statement** first evaluates its expression, then executes all statements that follow the matching `case`.

```
int gradeLevel;
// ... gradeLevel got updated
switch (gradeLevel) {
    case 1: System.out.print("Go to the Gymnasium");
            break;
    case 2: System.out.print("Go to the Science Auditorium");
            break;
    case 3: System.out.print("Go to Harris Hall Rm A3");
            break;
}
```

Expression

This statement is executed if the expression matches 1 (`gradeLevel` is 1)

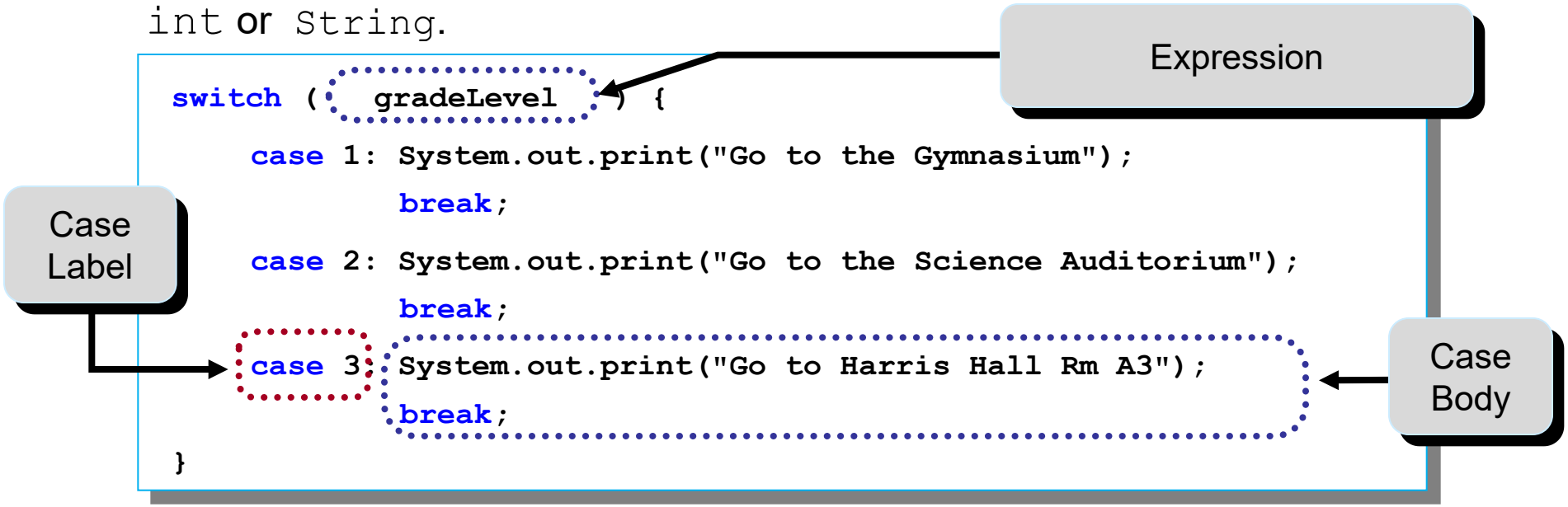
This statement is executed if the expression matches 3 (`gradeLevel` is 3)

The `switch` Statement

- General Syntax:

```
switch ( <expression> ) {
        case <label 1> : <case body 1>
        // ...
        case <label n> : <case body n>
}
```

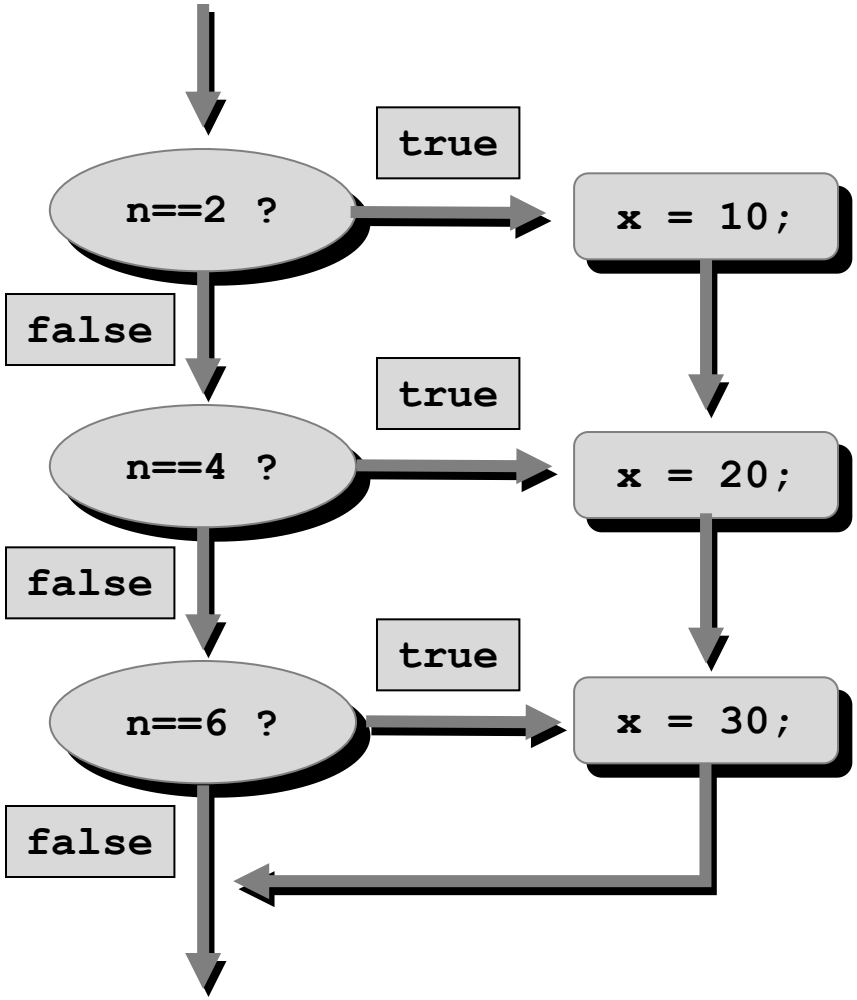
- where the data type of `<expression>` could be `byte`, `short`, `char`, `int` or `String`.



`switch` without `break` statement

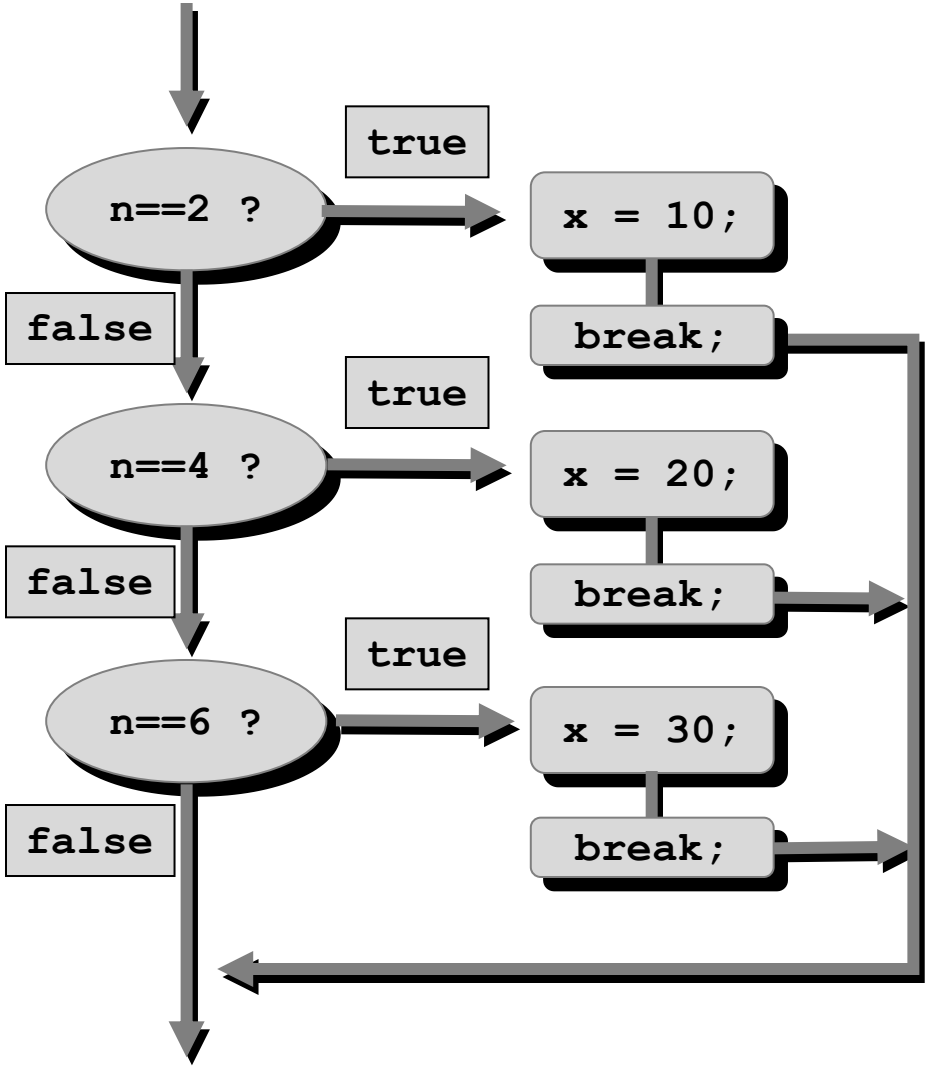
```
switch ( n ) {
    case 2: x = 10;
    case 4: x = 20;
    case 6: x = 30;
}
```

The `switch` statement evaluates its expression first, then executes **ALL** statements that follow the matching case.



`switch` with `break` statements

```
switch ( n ) {
    case 2: x = 10;
            break;
    case 4: x = 20;
            break;
    case 6: x = 30;
            break;
}
```



* Use `break` to terminate the `switch` statement, commonly used for each specific case block.

`switch` with `default` block

- The `default` section handles all values that are not handled by any one of the case sections

```
switch (ranking) {
    case 10:
    case  9:
    case  8:
    case  7: System.out.print("Master");
             System.out.println("-Level");
             break;

    case  6: case  5: case  4:
    case  3: System.out.println("Junior");
             break;

    default: System.out.println("Unknown Level!");
             break;
}
```

Enhanced `switch` Statement (New)

- In newer Java versions, an enhanced **`switch`** statement is introduced
  - This enhanced `switch` statement (in "arrow-form" **`->`**) makes the `switch` statement clearer to read and more concise, when compared to conventional `switch` statement. No `break` is required.
  - The following two examples of enhanced `switch` statements are basically working in the same way as the last example.
    - Left example: execution of statements in case body.
    - Right example: expression yielding a value in case body.

```
switch (ranking) {
  case 10, 9, 8, 7 -> {
    System.out.print("Master");
    System.out.println("-Level");
  }
  case  6, 5, 4, 3 ->
    System.out.println("Junior");
  default ->
    System.out.println("Unknown Level!");
}
```

```
String levelStr = switch (ranking) {
  case 10, 9, 8, 7 -> "Master-Level";
  case  6, 5, 4, 3 -> "Junior";
  default -> "Unknown Level!";
};
System.out.println(levelStr);
```

Standard (Console) Input / Output

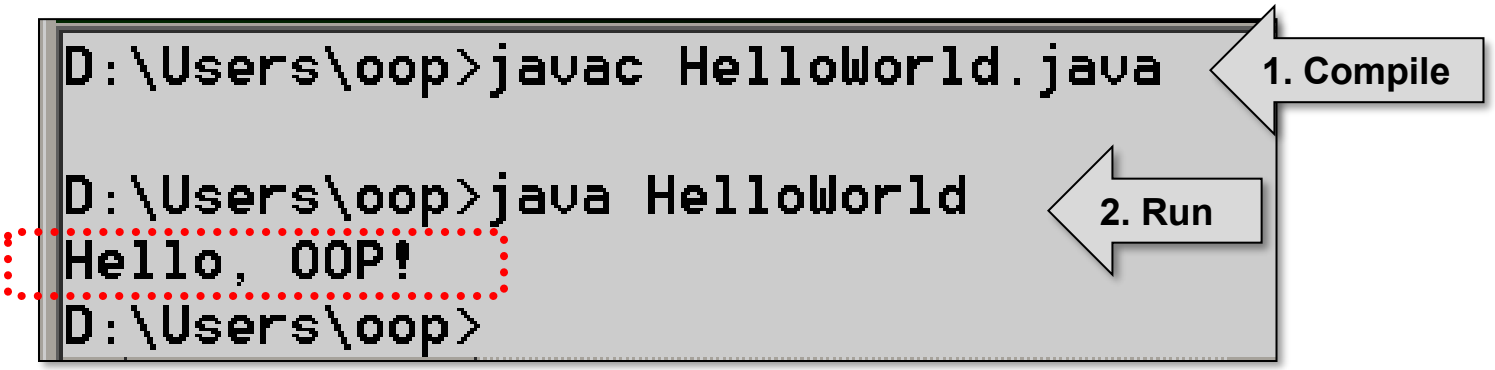
- Two common types of user interface to let user interact with our program:
  - Command-Line Interface (CLI): user-input with typing and output display on console, line-by-line
  - Graphical User Interface (GUI): user interacts with visual graphical components such as graphical buttons and windows
- Beginners often starts learning to write program with CLI, standard input / output window for inputting / displaying text (on console)
- We may use standard Java classes for command line interaction (on console), e.g.
  - **`System`** (in package `java.lang`)
  - **`Scanner`** (in package `java.util`)

Remarks: The **`java.lang`** package is always imported by default.

Standard Output: `print()` Method

- Use the **`print()`** method to output a value to the standard output window, *without adding a new line* after printing
- The **`print()`** method will continue printing, from the end of the currently displayed output
- Example, two **`print()`** method calls will display the second message directly following the first one *without new line*

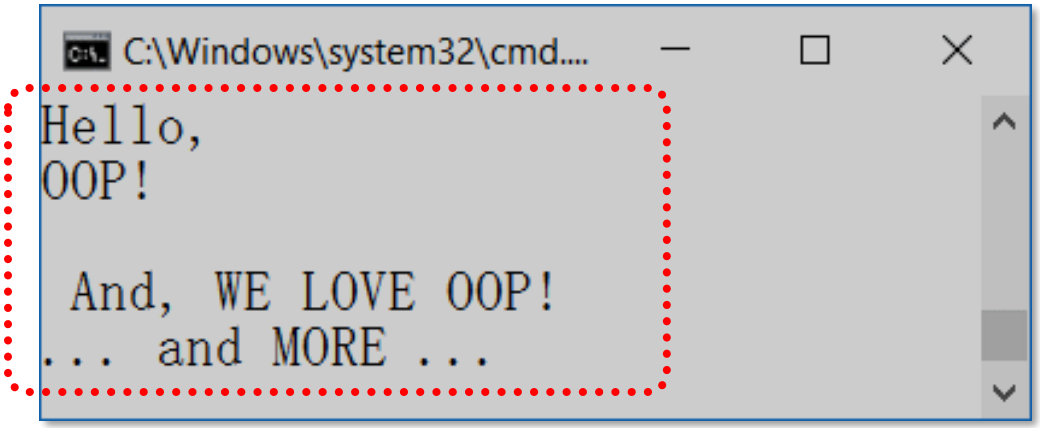
```
System.out.print("Hello, ");
System.out.print("OOP!");
```



Standard Output: `println()` Method

- Use `println()` instead of `print()` to terminate the current line (jump to a new line), after printing the text message
  - Thus, a new line `'\n'` is added after printing

```
System.out.println("Hello, ");
System.out.println("OOP!");
System.out.println("\n And, WE LOVE OOP!");
System.out.println("... and MORE ...");
```



Standard Output: `printf()` Method

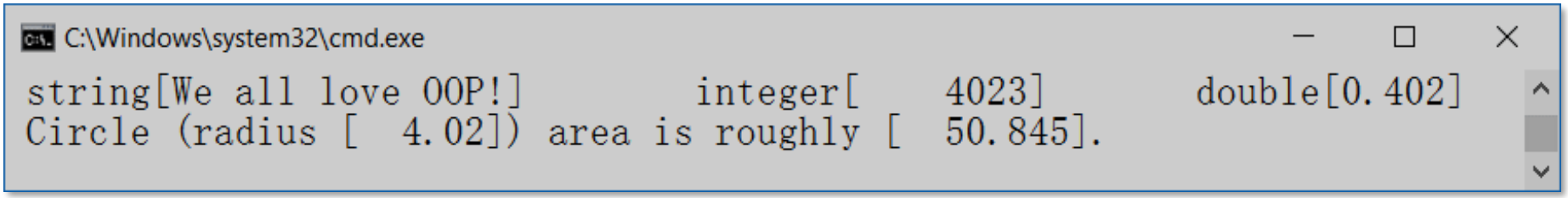
- Use `printf` method ("formatted") displays *formatted data*, similar to the one in C language.
  - Some simple formatting conversion specifiers below:

`%d` for integer; **`%f`** for floating point; **`%s`** for string

`%6d`, **`6`** is minimum character width;

`%6.2f`, **`6`** is minimum character width, with **`2`** decimal places

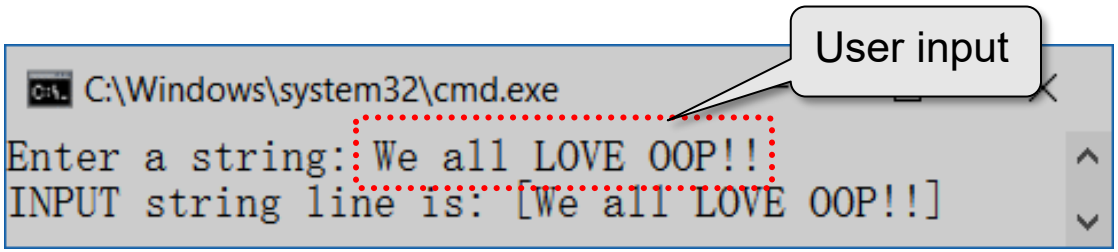
```
System.out.printf( " string[%s] \t integer[%8d]\t double[%.3f]\n",
                          "We all love OOP!", 4023, 0.4023);
double radius = 4.023;
System.out.printf( " Area of circle (radius [%6.2f]) is [%8.3f].\n",
                          radius, Math.PI*radius*radius);
```



Standard Input (with **`Scanner`**)

- To input a simple string line, we may use method `nextLine()` of class `Scanner` in package `java.util`
  - `import` statement required: `import java.util.Scanner;`

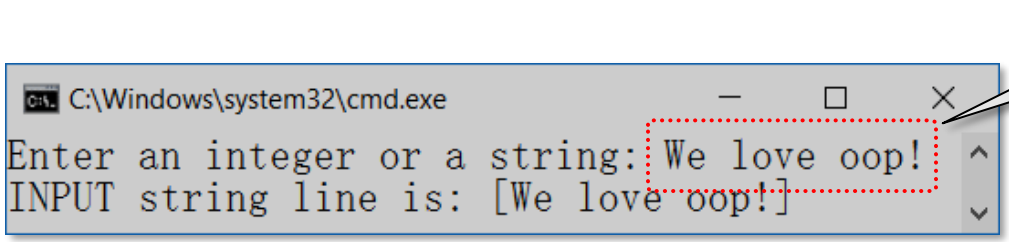
```
System.out.print("Enter a string: ");

// Below: Set Scanner scanning from standard input
Scanner scanner = new Scanner(System.in);

// Below: Scan the whole line as a string
String str = scanner.nextLine();

System.out.println("INPUT string line is: [" + str + "]");
```



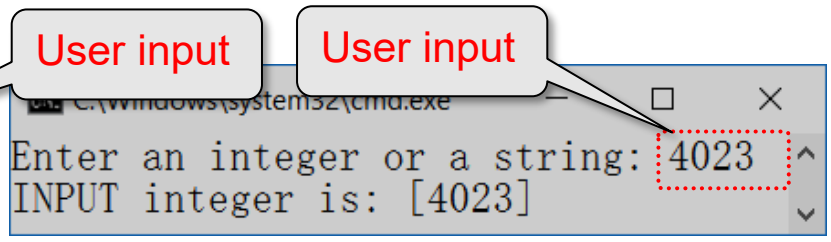
Standard Input (with **`Scanner`**)

- To input primitive data values, we may use other methods such as `nextInt()` for integer, or `nextFloat()` for floating point number

```
System.out.print("Enter an integer or a string: ");
Scanner scanner = new Scanner(System.in);
if (scanner.hasNextInt()){ // check if interpreted as an int value
// Below: scan the next as an integer (int)
   int num = scanner.nextInt();
   System.out.println("INPUT integer is: [" + num + "]");
} else {
// Below: scan the whole line as a string
   String str = scanner.nextLine();
   System.out.println("INPUT string line is: [" + str + "]");
}
scanner.close();  // *** BE CAREFUL
// ** CAUTION: closing scanner also closes System.in,
//             which may NOT be reopened again later.
```



User input



User input

Common `Scanner` Methods:

Reference Only

Methods	Examples
`nextLine()`	`String strLine = scanner.nextLine();`
`nextBoolean()`	`boolean bo = scanner.nextBoolean();`
`nextByte()`	`byte by = scanner.nextByte();`
`nextDouble()`	`double d = scanner.nextDouble();`
`nextFloat()`	`float f = scanner.nextFloat();`
`nextInt()`	`int i = scanner.nextInt();`
`nextLong()`	`long l = scanner.nextLong();`
`nextShort()`	`short s = scanner.nextShort();`
`next()`	`String str = scanner.next();`

** A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.*

https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Scanner.html

Common Problems in Java Programming

- **Use a variable WITHOUT declaration first**

```
int abc = 123;
```

- **Duplication** of declaring the same variable within the same scope

```
double n1, n2, n3;
// some codes here
double n3; // ERR: Duplicated variable declared!
```

- **Mix** up **=** (*assignment operator*) with **==** (*relational operator, equal-to*)

```
if (testScore = 60) // ERR: should use == to compare
```

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/