Unit 4: **Using Given Classes, and Standard `String` Class**

Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U4: Using Given Classes, and Standard **`String`** Class

- Basic Class Type and Declaration
- Using Given Classes and Objects
  - Declare
  - Create (& Assign)
- Access Objects: Accessing Fields and Methods
- Class **`String`** in Java
- Standard Classes for Date and Time Handling

Basic Class Type and Declaration

- Java supports **primitive data types** and **reference types** (e.g. array types or user-defined class types)
  - Numeric data types such as `int`, `long`, `float`, and `double` belong to the primitive data types

- However, most of Java programs heavily rely on defining and manipulating objects of specific ***classes*** (in standard APIs, or user-defined types for specific applications)

- In order to make use of certain existing classes (those in standard API libraries such as the class `String`, or programmer-defined ones) we need to know:
  - What they are (e.g. fields, constructors and methods).
  - How to access and interact with them.

Using Given Classes and Objects

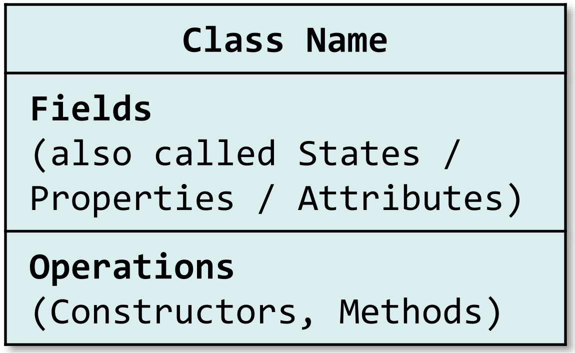
- In general, standard classes are well-documented with **API specification documents** telling:
  - What they are (e.g. fields, constructors and methods)
    - However, we normally do not need to know “how” they are implemented (e.g. details of implementing the methods)
  - How to access and interact with them
    - How we access them (e.g. whether we can access their fields, constructors and methods; and how to)
    - In particular, how we can call their methods properly
- It is important to know how to read these API specification documents that how developers can properly use them
  - E.g. to know the `String` class, read its API document below:

  https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/String.html

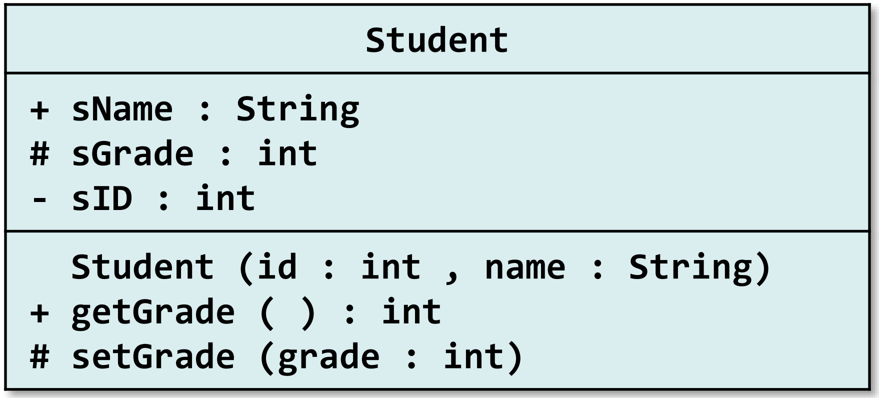
Using Given Classes and Objects

- API specification document of a Java class typical includes:
  - ***General information*** of the class, its package, inheritance hierarchy etc.
  - ***Fields*** and their descriptions: what they are & how to access them
  - ***Constructors*** and their descriptions: what they do & how to access them
  - ***Methods*** and their descriptions: what they do & how to access them
- Apart from API documents, simple UML class diagrams of specific classes may give users a brief reference of how to use them.
  - Below shows the general form (left) of UML class diagram and a sample of UML class diagram representing a class `Student` (right)

** More details of UML class diagrams will be given in later unit.*



```
Class Name
-----------------------------
Fields
(also called States /
Properties / Attributes)
-----------------------------
Operations
(Constructors, Methods)
```



```
Student
-----------------------------------------
+ sName : String
# sGrade : int
- sID : int
-----------------------------------------
  Student (id : int , name : String)
+ getGrade ( ) : int
# setGrade (grade : int)
```

Basic Syntax / Form of ***Defining*** a Java Class

- Common syntax of ***defining a class*** in Java:

```
public class <ClassName> { // class declaration
   <Class Body: may have fields, constructor and methods>
}
```

- E.g. Defining / Declaring Java classes `HelloWorld` and `Circle`

```
public class HelloWorld { // class declaration
   // <Class Body: may have fields, constructor and methods>
}
```

```
public class Circle { // class declaration
   // <Class Body: may have fields, constructor and methods>
}
```

Basic Syntax / Form of ***Defining*** a Java Class

- Common syntax of ***defining fields*** (instance variables) in a Java class:

```
public class <ClassName> { // class
  <DataType> <fieldNameA> ; // a field (instance variable)
  <DataType> <fieldNameB> = <value>; // with default value
}
```

- E.g. Adding data fields to class `Circle`

```
public class Circle { // class declaration
  double radius; // field
  double x=0.0;  // field, with default value (say coordinate)
  double y=0.0;
  // <Class Body: may have fields, constructor and methods>
}
```

Basic Syntax / Form of ***Defining*** a Java Class

- Common Syntax of ***defining constructor and method*** in a Java class:

```
public class <ClassName> { // class
  <ClassName>(<paras>) { <Constructor Body> } // constructor
  <ReturnType> <methodName>(<paras>) { <Method Body> }// method
}
```

- E.g. Adding constructor(s) and method(s) to class `Circle`

```
public class Circle { // class declaration
  double radius; // field
  double x=0.0;  // field, with default value (say coordinate)
  double y=0.0;
  Circle(){ // Constructor
  // Body of Constructor
  }
  double getArea(){ // Method
  // Body of Method
  }
}
```

Basic Syntax / Form of ***Using*** a Defined Java Class

- Basic syntax of ***creating an object / instance*** of a Java class:

```
new <ClassName>(<args>) # obj. creation leads calling constructor
<ClassName> aObj = new <ClassName>(<args>); # often assign to var.
```

```
Circle acircle = new Circle(2.3); # an example
String strObj = new String("A String Object"); # another example
```

- Basic syntax of ***accessing a field*** of a specific object of a class (with Dot-notation):

```
 <varName> = <objectName>.<fieldNameA>; // get field value
 <objectName>.<fieldNameA> = <value>; // set field value
```

```
 aCircle.radius = 12.3; // an example set field value
```

- Basic Syntax of ***accessing / calling a method*** of a specific object of a class (with Dot-notation):

```
<objectName>.<methoName>(<args>); # call a method without return
 <varName> = <objectName>.<methoName>(<args>);# method with return
```

```
 double area = aCircle.getArea(); // an example of method calling
```

Class and Method Declaration

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

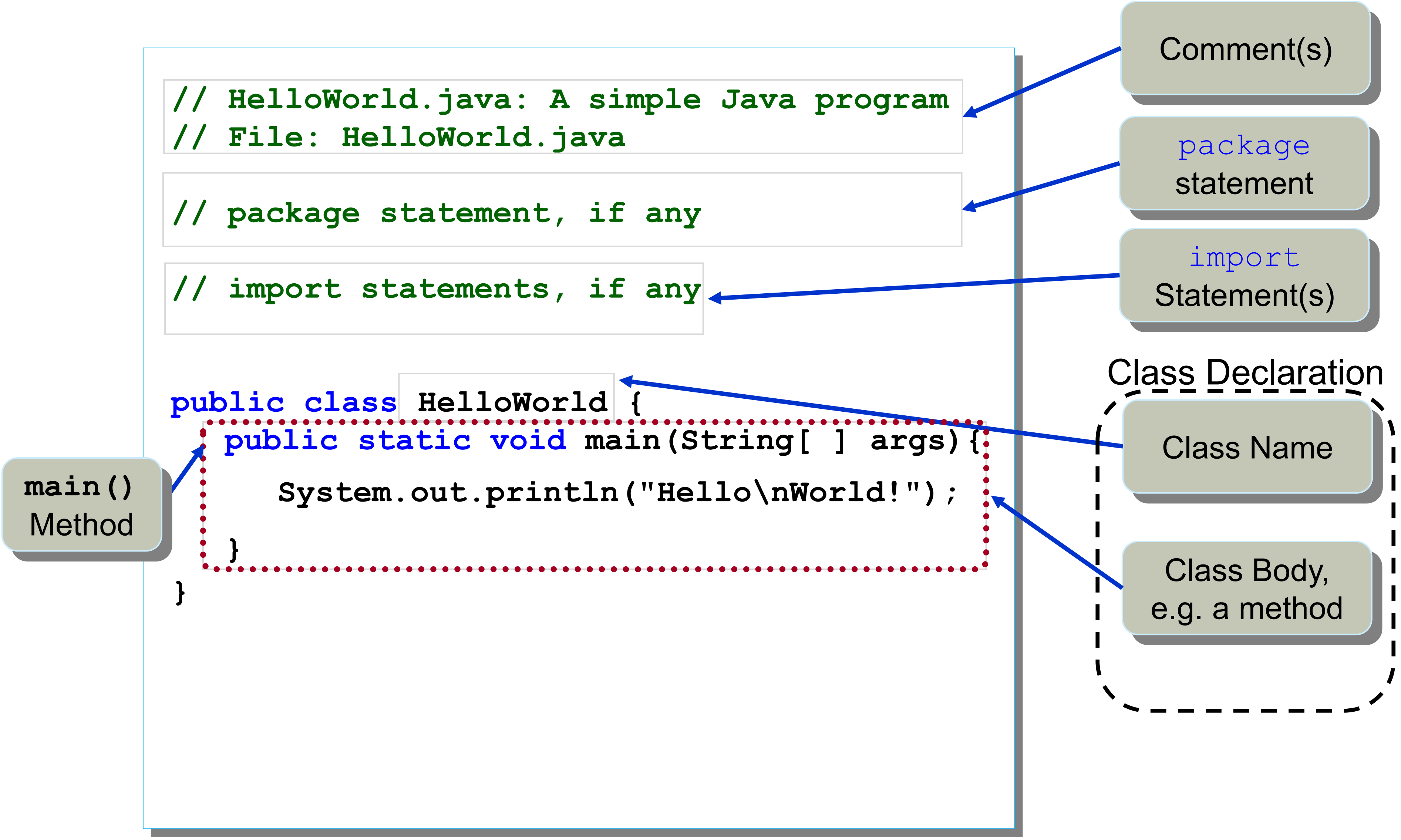
```
<modifier(s)> class <class name> {  <class body>  }
```

- ***Method Declaration*** in a Basic Form:

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

1. `<modifier(s)>` is a sequence of term(s) designating different kinds of methods
2. **`<return type>`** is the type of data value returned by the method. Use keyword `void` if without return value.
3. **`<method name>`** is the name of a method in lowercase by convention
4. `<parameter(s)>` is a sequence of value(s) passed to the method
5. `<method body>` is a sequence of instructions
6. Components 3 and 4 comprise the *method signature*

Class and Method Declaration
(Sample: HelloWorld.java)



```
// HelloWorld.java: A simple Java program
// File: HelloWorld.java

// package statement, if any

// import statements, if any


public class HelloWorld {
   public static void main(String[ ] args){
      System.out.println("Hello\nWorld!");
   }
}
```

Comment(s)

package statement

import Statement(s)

Class Declaration

Class Name

Class Body, e.g. a method

main() Method

Calling and Executing Methods

- **Method** is program “module” containing statements to perform operations for specific tasks.

  (Nature of method in Java is much similar to “function” in C.)*

  - 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
  - Basically, two ways to finish a method after calling:

    1) Method with return value (returns value of specific type)
       - A proper `return` statement is required for returning a value. Method terminates after executing a `return` statement
    2) Method without return value (return type `void`)
       - `return` statement is optional. Without `return` statement, the method will leave after finishing its method body, as the case of the `main()` method in the last sample code

Method `main()` - Entry Point of Java Program

- Every Java program should have at least one class
  - A Java program may include many different classes, in different source `*.java` files

- In order to run a specific class of a program, the class must contain a specific method named **`main`** (case-sensitive, `Main` ≠ `main`):

```
public static void main(String[] args) {
        <main method body>
}
```

- The **`main()` method** is the **execution entry point of Java application**, where the Java program starts running

- Often the only Java class containing this `main()` method in a Java program is often call the main class

Argument for `main()` Method

- The `main` methods get an array of strings as an argument, these are the command line arguments user may pass to the program

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

- To compile and run it with arguments, e.g.:

```
javac MyProgram.java
java MyProgram  arg0  arg1  arg2
```

To Compile

To Run, with arguments

- In the example above, the following string array is passed into the parameter `args`:

```
{ "arg0", "arg1", "arg2" }
```

Fields and Variables

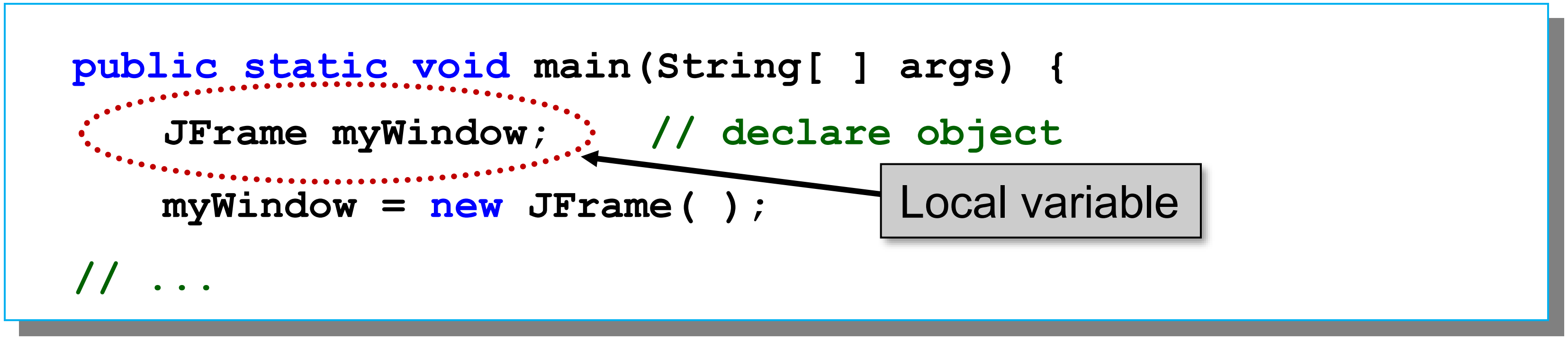
- There are several kinds of “variables” in Java:
  - **Local variables**: variables declared within block of code, such as within a method or a constructor
  - **Parameters**: variables in method/constructor declarations
  - **Fields**: these are “member variables” declared in a class, outside all methods or constructors

```
public class HiWorld {  // class (named HiWorld) declaration
  String inNameStr; // declare a Field
  HiWorld(String inStr) { // a constructor, with Parameter
      inNameStr = inStr;
  }
  public static void main(String[ ] args) { //method & parameter
// a Local Variable below
      String nameStr = JOptionPane.showInputDialog(null,
                    "What is your name?");
// ...
```

** Details of the topic will be further discussed in later unit*

Local Variables

- ***Local variables*** are declared *within a method / constructor* and used for *temporary services*, such as storing intermediate computation results
  - Compiler does not assign a default value to an uninitialized local variable.
    - Accessing uninitialized local variable causes compile-time error
  - Fields declared but not initialized will be set to default values by the compiler: e.g. zero, false or null value



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

Local variable

Using Given Classes and Objects

- It is common to make use of given class types (in predefined standard APIs or self-defined ones) to create more sophisticated software

- To use these classes (and their objects), it includes three basic steps in general:
  - ***Declare*** Object, e.g.

    ```
    String aStr;  // declare object (of String class)
    Student amy;  // declare object (of Student class)
    ```

  - ***Create / Instantiate*** (& ***Assign***) Object, e.g.

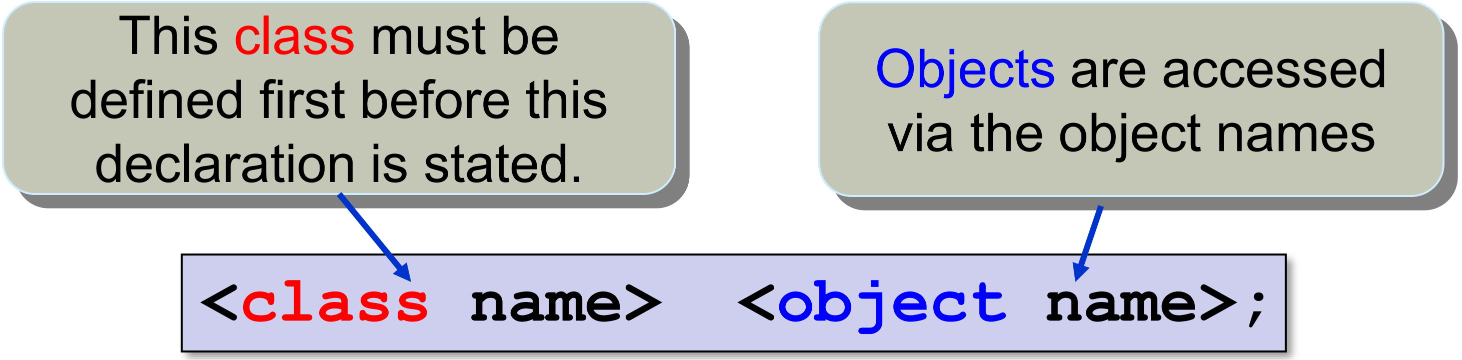
    ```
    new String("OOP") // create object only
    aStr = new String("OOP"); // create & assign object
    amy = new Student(20201234, "CHOW Amy"); // Student obj
    ```

  - ***Access*** Object, e.g.

    ```
    aStr.split(","); // access object’s method
    amy.setGrade(100); // access object’s method
    ```

Object Declaration

- Similar to **declare** a variable of primitive data type, declaring an object has the similar syntax:



This class must be defined first before this declaration is stated.

Objects are accessed via the object names

```
<class name>  <object name>;
```

Examples:

```
JFrame     myWindow;
Customer   customer;
Student    jan, jim, jon;
Vehicle    car1, car2;
```

Object Declaration

```
/*
     Sample Program: Display a Window
     File: Sample1.java
*/
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);
  }
}
```

Object Declaration

Object Creation / Instantiation
(and Assignment)

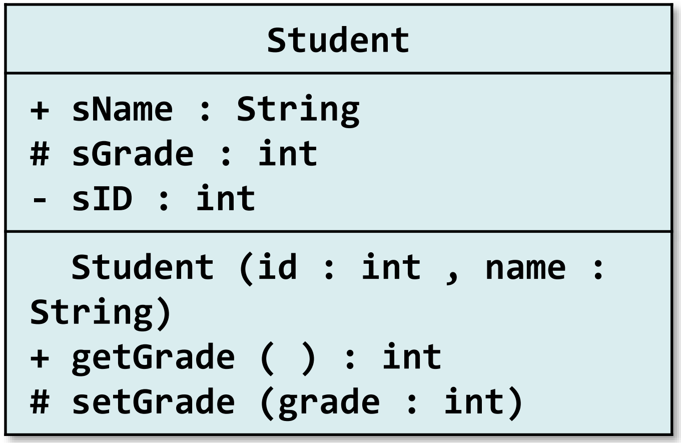
- Objects (or instances) of a specific class type can be created / instantiated in run time, by using the **new** operator with a constructor

- **Object creation / instantiation** has the following syntax:

```
    new <class name>(<arguments>)
```

- Examples (suppose we have a class `Student`):

```
new String("We Love OOP") // String object
new Student(20001234, "CHOW Amy")  // Student object
```



```
Student
+ sName : String
# sGrade : int
- sID : int
  Student (id : int , name : String)
+ getGrade ( ) : int
# setGrade (grade : int)
```

Object Creation / Instantiation
(and Assignment)

- The `new` **operator** allocates memory to store the object of specified class type (also the name of constructor), and then invokes corresponding constructor to initialize the object, which is stored in the memory

- When constructor ends, `new` returns a reference (essentially a memory address) to the object so that it can be accessed elsewhere

- Often declaring and creating object in a line:

```
<class name> <object name> = new <class name>(<arguments>);
```

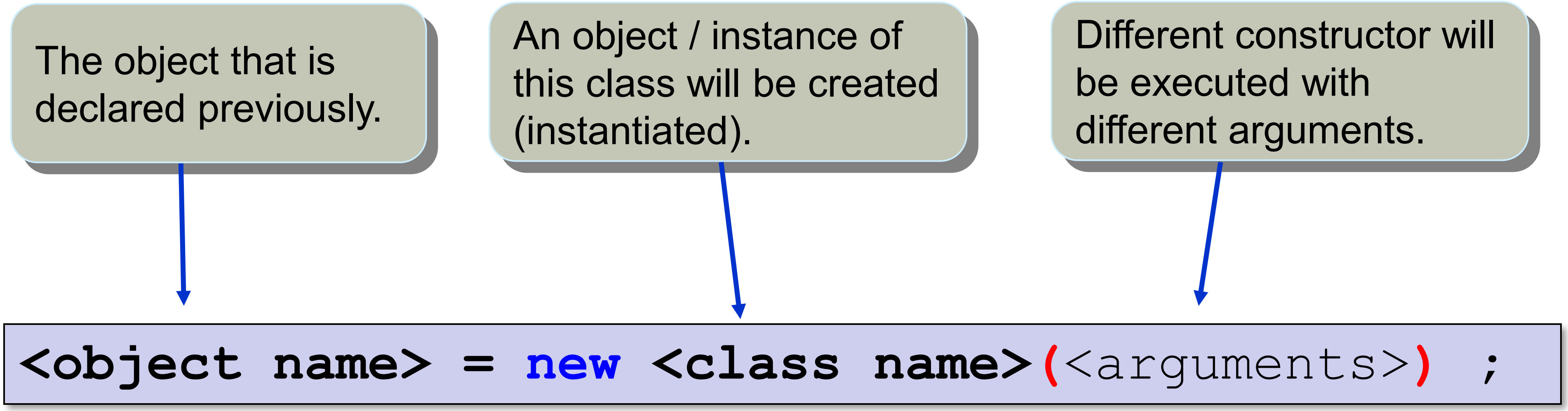
```
// Declare & assign string object below
String oopStr = new String("We Love OOP");

// Declare & assign Student object below
Student amy = new Student(20201234, "CHOW Amy");
```

Object Creation / Instantiation
(and Assignment)

```
/*
     Sample Program: Display a Window
     File: Sample1.java
*/
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);
  }
}
```

Object Creation / Instantiation (and Assignment)

Object Creation / Instantiation
(and Assignment)



The object that is declared previously.

An object / instance of this class will be created (instantiated).

Different constructor will be executed with different arguments.

```
<object name> = new <class name>(<arguments>) ;
```

Examples:

```
myWindow = new JFrame( )  ;
customer = new Customer( );
jon      = new Student("John Java");
car1     = new Vehicle( );
```

** A constructor acts like a special kind of method that is called when an object is instantiated (newly created). Details of the topic will be further discussed in later unit*

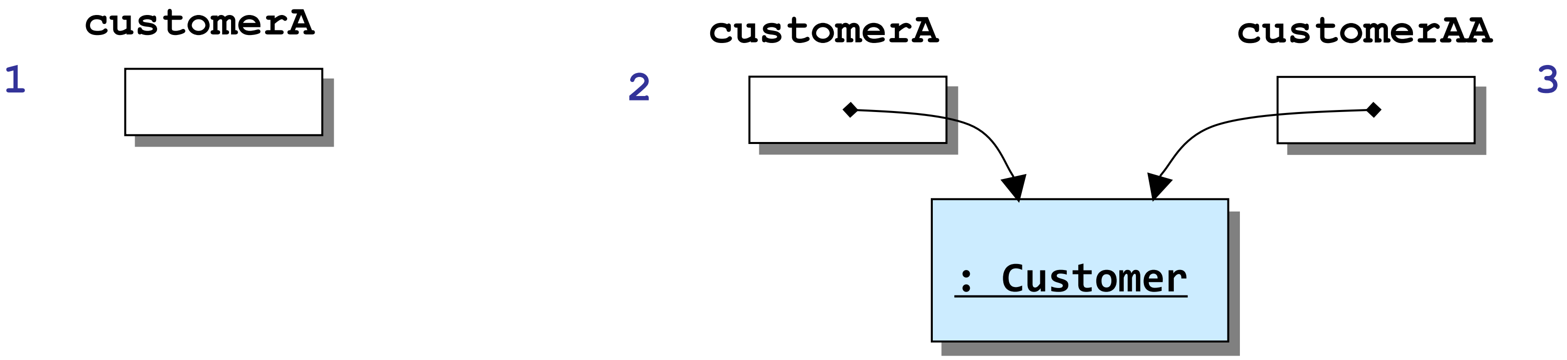
Declaration, Creation (& Assignment)

```
1   Customer customerA;   // object declaration
2   customerA = new Customer(); // create (& assign)
3   Customer customerAA = customerA;   // another variable
```

1. Identifier `customerA` is declared as an object of a `Customer` class, and space is allocated in memory only for referencing an object

2. An object of a `Customer` class is created and the identifier `customerA` is then assigned to refer to it

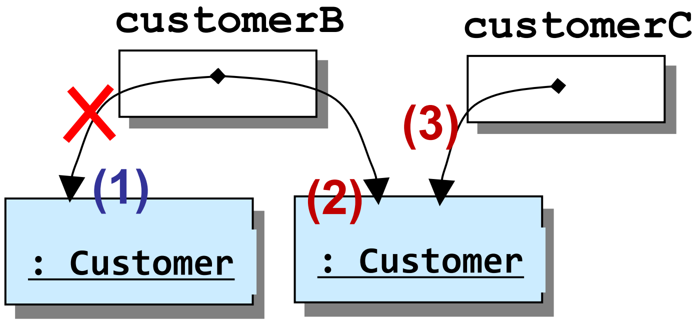
3. Another identifier `customerAA` is also assigned to refer to the same object `customerA` refers to



Re-assigning / Re-referencing an Object

- **Re-assign** another object to the same declared identifier is essentially re-referencing to another object from it's original. For example:

```
    Customer customerB;
(1) customerB = new Customer();
(2) customerB = new Customer();
(3) Customer customerC = customerB;
```



- The object `Customer` previously referenced by `customerB` is no longer referenced
  - It becomes "garbage" in Java, which will be automatically collected by the JVM (Java Virtual Machine)
    - This mechanism of memory management is called **Garbage Collection**.

* *Garbage Collection*: Unreferenced space is *automatically collected* by JVM

Object Declaration and Creation

- We often combine the object declaration and creation (with assignment) into one statement as follows:

```
<class name> <object name> = new <class name>(<arguments>);
```

Examples:

```
JFrame  myWindow = new JFrame();
Customer customer = new Customer();
Student jon = new Student("John Java");
Vehicle car1 = new Vehicle();
```

Access Object's Fields and Methods

- Access fields and methods of a specific class / object in two different ways:
  - Access fields and methods ***outside* its own class**
    - Use a **dot-notation form**
    - In general, member to be accessed has to be associated to a specific object, as the form:

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

  - Access fields and methods ***inside* its own class**
    - Dot-notation form is often optional, in most situation
    - In general, keyword `this` is used for dot-notation form

  ** More details will be discussed in later units.*

Access Object's Fields and Methods

- Reference a field (instance variable) in the object (with *dot notation*):

```
<object name>.<field name>
```

- *Examples:* Given `car1` as an object of class `Vehicle` having a (public) field `speed`

```
int currentSpeed = car1.speed;
car1.speed = 10;
```

- Invoke a method on the object (*with dot notation*):

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

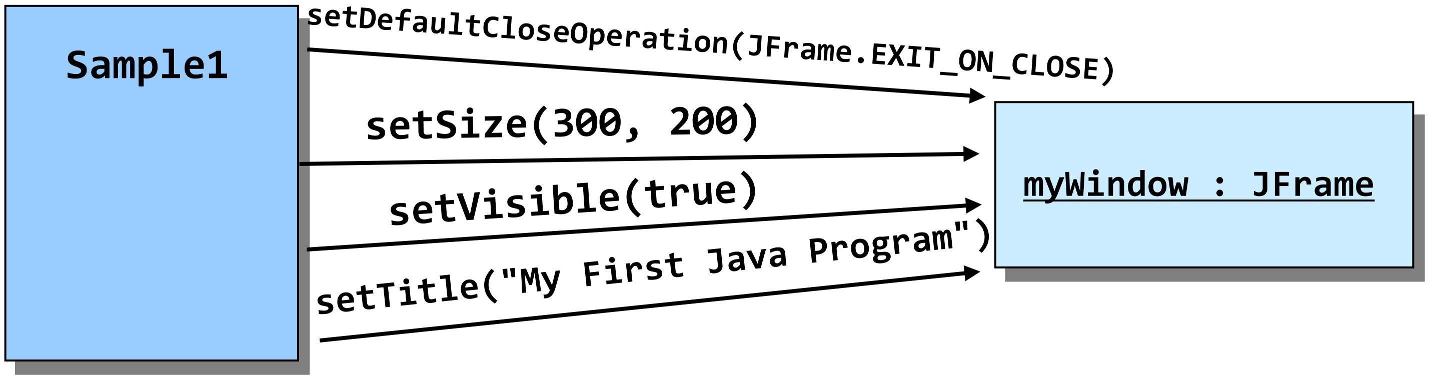
```
car1.setSpeed(2);
myWindow.setSize( 300, 200 );
account.deposit( 200.0 );
```

Invoke Object's Methods: More Examples

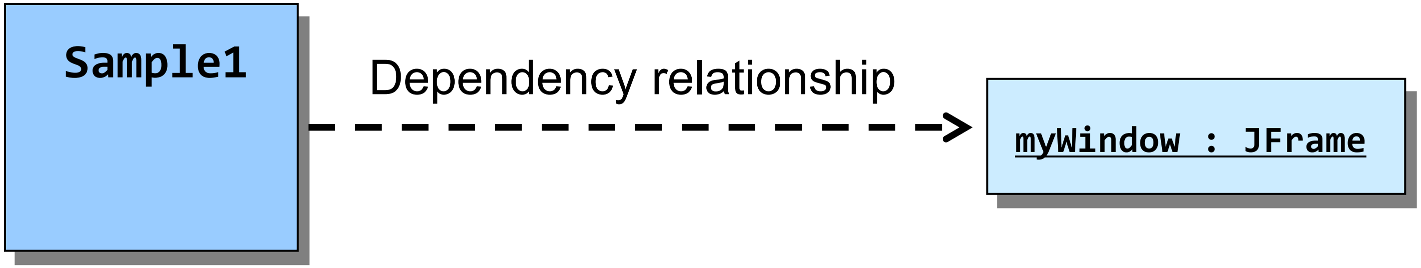
```
/*
     Sample Program: Display a Window
     File: Sample1.java
*/
import javax.swing.*;

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);
   }
}
```

Invoke methods of an object (`myWindow`)

Program Diagram for `Sample1.java`


Sample1
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
setSize(300, 200)
setVisible(true)
setTitle("My First Java Program")
myWindow : JFrame

- Instead of drawing all messages (above), we summarize it by showing only the ***dependency relationship***
- The diagram (below) shows that `Sample1` "depends" on the service provided by `JFrame`



Class `String` in Java

- Among all standard classes in standard APIs, class `String` is a very important and popular one for handling string text.
  - A *string* is a sequence of characters, e.g. `"We all love OOP"`
  - In Java, a single character is represented using the data type `char`, and each is written as a symbol enclosed in single quotes (E.g. `'W'`)

- `String` is a very special Java class that its string object can be constructed in 2 ways:

1) Using typical way ***with* `new` *operator*** (*as other classes*)

```
new String(<String literal>)
```

   – e.g. `String aStr = new String("A new string");`

2) Using a string ***literal directly***:

```
<String literal>
```

   – e.g. `String bStr = "A literal string";`

https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/String.html

Class `String` in Java

- In Java, a **`String` object** is ***immutable*** (i.e. its content cannot be changed once constructed)
- Around many methods defined in the `String` class. E.g.
  - **`length()`, `equals()`, `charAt()`, `split()`, `substring()`, `indexOf()`**
- One of the string operations is called **concatenation** (**+** operator)
- With reference types (e.g. `String` class or array), we have two different ways to compare them. We can check whether:
  1. Two variables ***point to the same object / instance*** (use “equal to” **operator ==** to check if they have the same ***reference***)
  2. Two distinct objects have the ***same content / value*** (we may need to define/override the **`equals()` method** of the root class `Object`, as the class `String`)

* Remark: Note that for primitive types, we use “equal to” operator **==** to compare their values

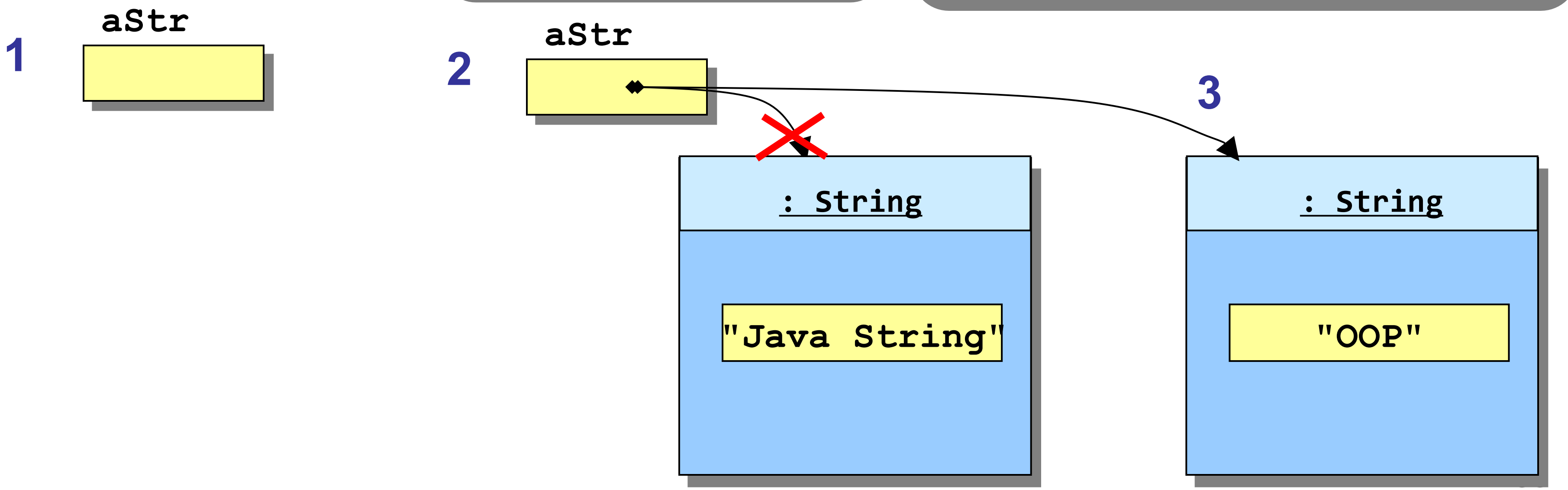
Creating String Objects, with new

```
1   String aStr;
2   aStr = new String("Java String");
3   aStr = new String("OOP");
```

1. The identifier aStr is declared and space is allocated in memory

2. A String variable holds a reference to a String object that stores a string value

Immutable object means:
3. Once created, the object/instance CANNOT be changed
(its content/state cannot change)

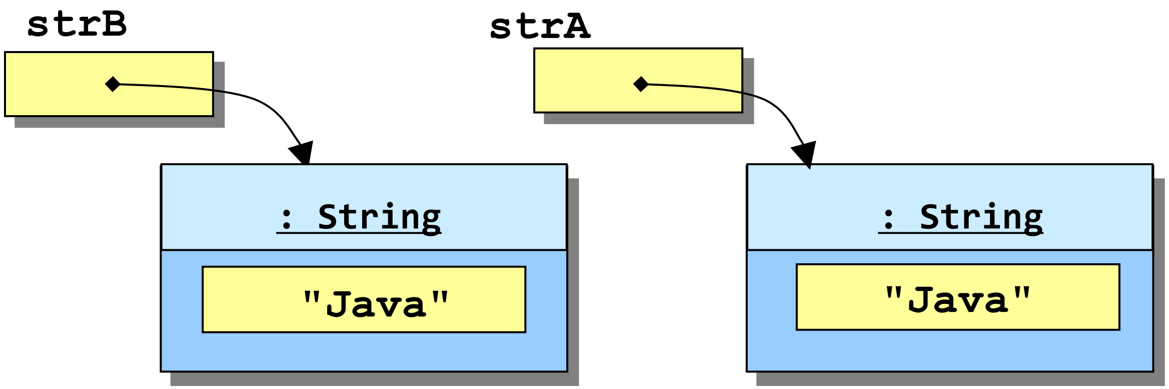


Creating String Objects, with new
(String Comparison with == vs. with Method equals())

```
String strA = new String("Java"); // a new object
String strB = new String("Java"); // another new object

if (strA == strB) {  // "equal to" == operator, compares reference
     System.out.println("They do refer same object");
} else {
     System.out.println("They do NOT refer same object");
}
// equals() method checks if same content
if (strA.equals(strB)) {
     System.out.println("They are equal (content)");
} else {
     System.out.println("They are not equal (content");
}
```

false

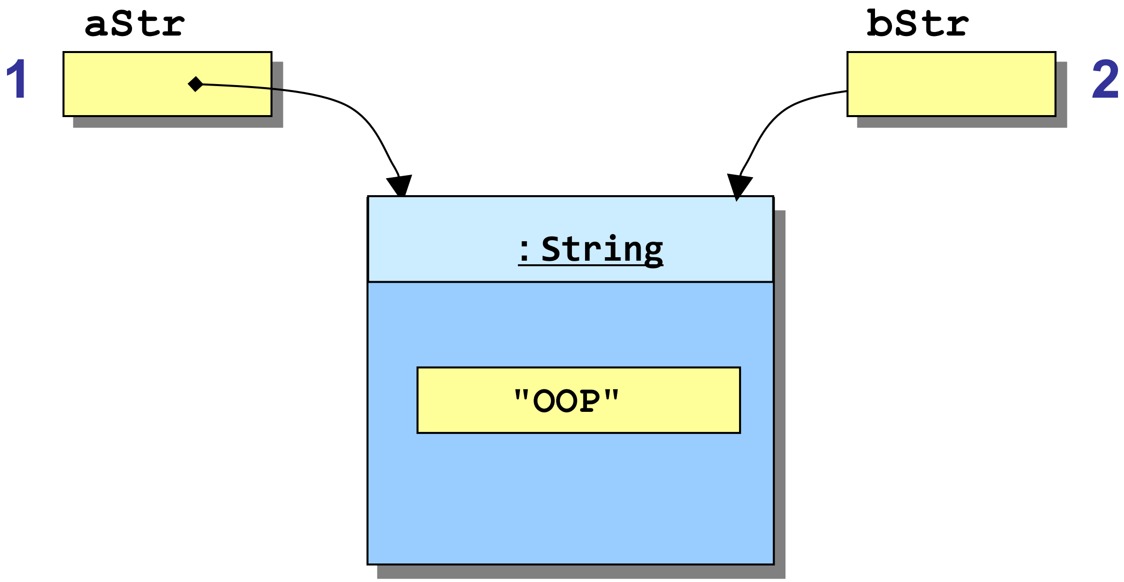
true



Creating `String` Objects, with `String` Literals (`String` is SPECIAL)

```
1   String aStr = "OOP"; // create String with literal
2   String bStr = "OOP";
```

A string that is designated without **new** operator is called String literal. Same contents will share the same storage.

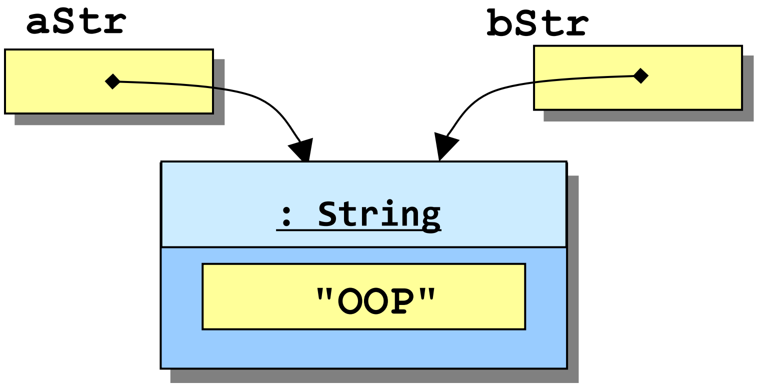


Creating `String` Objects, with `String` Literals
(`String` Comparison with == vs. with Method **`equals()`**)

```
String aStr = "OOP"; // create String with literal
String bStr = "OOP";

if (aStr == bStr) {  // "equal to" == operator, compares reference
    System.out.println("They do refer same object");
} else {
    System.out.println("They do NOT refer same object");
}
// equals() method checks if same content
if (aStr.equals(bStr)) {
    System.out.println("They are equal (content)");
} else {
    System.out.println("They are not equal (content");
}
```

true

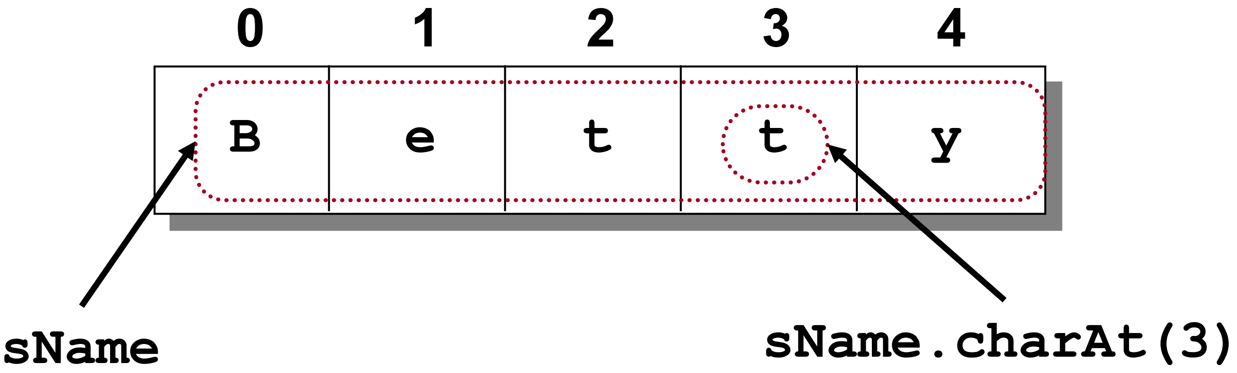
true



Class: String
Method: charAt()

- **Individual characters** in a **String** accessed with the **charAt()** method. Index number starts from 0.

```
String sName = "Betty";
```



This variable refers to the whole string

The method returns the character **'t'** at position # 3

public char charAt(int index)

Returns the **char** value at the specified index.

Example: Counting Vowels

```
char      letter;
String    name  =  JOptionPane.showInputDialog(null,"Your name:");
int       numberOfCharacters = name.length();
int       vowelCount = 0;

for (int i = 0; i < numberOfCharacters; i++) {
    letter = name.charAt(i);
    if (   letter == 'a' || letter == 'A' ||
           letter == 'e' || letter == 'E' ||
           letter == 'i' || letter == 'I' ||
           letter == 'o' || letter == 'O' ||
           letter == 'u' || letter == 'U`         ) {
        vowelCount++;
    }
}
System.out.print(name + ", your name has " + vowelCount + " vowels");
```

Here’s the code to count the number of vowels in the input string

Class: `String`
Method: `length()`

- Assume `text` is a `String` object, and properly initialized to a string
- Method **`text.length()`** will return the number of characters in `text`
- Assume the value of `text` is `"programming"`, then `text.length()` will return 11 because there are 11 characters in the value of `text`

```
String str1, str2, str3, str4;
str1 = "Hello" ;
str2 = "Java" ;
str3 = "" ; //empty string
str4 = " " ; //one space
```

```
str1.length( )  →  5
str2.length( )  →  4
str3.length( )  →  0
str4.length( )  →  1
```

public int length()

Returns the length of this string.

Class: `String`
Method: `split()`

- Method `split()` returns a string array that is created by splitting the string around a matching string token
- The string array returned by this `split()` method contains each substring of this string that
  - is terminated by another substring that matches the given expression, or
  - is terminated by the end of the string

```
String str = "I Love Java and Java loves me.";
String sInfo = "Chan Tai Man,12345678,M,Programming";
String [] sArr1 = str.split(" ");
String [] sArr2 = str.split("and");
String [] sArr3 = sInfo.split(",");
```

```
sArr1 -> {"I", "Love", "Java", "and",
               "Java", "loves", "me."}
sArr2 -> {     "I Love Java ",
               " Java loves me."      }
sArr3 -> {"Chan Tai Man","12345678","M","Programming"}
```

public String[] split(String regex)

Splits this string around matches of the given regular expression.

Class: String
Method: format()

- String provides a method to return a formatted string (*similar* to C's printf() function):

```
String.format(<strFormat>, <item1>, <item2>, ...)
```

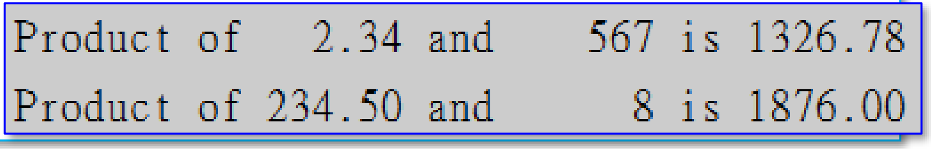
  - Some simple formatting conversion specifiers:

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

%6d, **6** is min. char width

%6.2f, **6** is min. char width, with **2** decimal places precision

```
// Examples
String sF = "%s of %6.2f and %6d is %6.2f"; // string format specifiers
float f1=2.34f, f2=234.5f;
int n1=567, n2=8;
String fStr1 = String.format(sF, "Product", f1, n1, f1*n1);
String fStr2 = String.format(sF, "Product", f2, n2, f2*n2);
System.out.println(fStr1); // ->
System.out.println(fStr2); // ->
```



```
Product of   2.34 and    567 is 1326.78
Product of 234.50 and      8 is 1876.00
```

public static String format(String format, Object... args)

Returns a formatted string using the specified format string and arguments.

Class: `String`

Method: `valueOf()`

- Returns the string representation of argument of other types.
  - Often these are primitive types e.g. `boolean, int, double`

```
String str1, str2, str3;
str1 = String.valueOf(true); // boolean to String
str2 = String.valueOf(4023); // int to String
str3 = String.valueOf(40.23); // double to String
```

```
str1 → "true"
str2 → "4023"
str3 → "40.23"
```

`static String valueOf(double d)`

Returns the string representation of the `double` argument.

Some Useful `String` Methods

Reference Only

Method	Meaning
`compareTo`	Compares the two strings, and return 0 if they are equal. `str1.compareTo( str2 )`
`substring`	Extracts the a substring from a string from the beginning index (inclusive) to the ending index (exclusive). `str1.substring( 1, 4 )`
`trim`	Removes the leading and trailing spaces. `str1.trim( )`
`valueOf`	Converts a given primitive data value to a string. `String.valueOf( 123.4565 )`
`startsWith`	Returns true if a string starts with a specified prefix string. `str1.startsWith( str2 )`
`endsWith`	Returns true if a string ends with a specified suffix string. `str1.endsWith( str2 )`

Remark: refer to the `String` class API documentation for more details:
https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/String.html

Handling Data Type Mismatch
(Number vs. String)

- While we may use `String` method `valueOf()` to convert primitive type to type `String`, often we need to convert string to primitive type. E.g. Suppose we want to input an age.

```
int age; //declare int variable for information of age
// ...
age = JOptionPane.showInputDialog( // which returns a String
         null, "Your age"); // attention: dialog to ask age info.
// similar to
age = "18"; // assign a "18" to int directly, OR input below
```

```
error: incompatible types: String cannot be converted to int
age = JOptionPane.showInputDialog( // which returns a String
```

- ***Compilation Error!!*** Because of Type Mismatch!

- `String` value (user input into a string type) cannot be assigned *directly* to an `int` variable (`age`)
  - Proper conversion should be done before assignment

- We use *Wrapper classes* to convert strings to different primitive types

Handling Data Type Conversion
(with Wrapper Class)

- **Wrapper classes** are special Java classes could be used for performing type conversions, e.g.
  - Using the wrapper class `Integer` to convert a `String` into an `int` numeric value as sample below
  - All the methods of the wrapper classes are `static`

    ** The usage of `static` will be further discussed in later unit*

```
int     age;                   // variable of primitive data
String  inputStr;              // variable of String Class
inputStr = JOptionPane.showInputDialog(
          null, "Your age");      // dialog box to get string
age = Integer.parseInt(inputStr); // convert String to int
System.out.println ("Your age is " + age);
```

Primitive Type and Wrapper Class

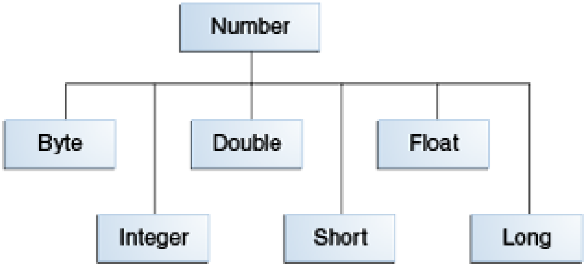
- When working with numbers, we often use the *primitive* data types in our program, e.g.:

```
int abc = 369;
float myF = 96.3f;
```

- However, there are situations we need to manipulate numbers using objects
  - such as creating a `Collection` of objects holding numbers
- Java provides a *wrapper* class for each primitive data type

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Wrapper Class	Method	Example	Result
Integer	parseInt()	Integer.parseInt("123")	123
Float	parseFloat()	Float.parseFloat("12.3")	12.3f
Double	parseDouble()	Double.parseDouble("12.3")	12.3



Standard Classes for Date and Time Handling

- There are many useful classes pre-defined in standard API libraries
  - Besides the classes of `String`, `JOptionPane`, `Scanner`, `Math`, there are other useful standard classes. E.g. to handle time and date, we may use classes **`Date`** and **`Calendar`** (in `java.util` package)

- Classes `Date` and `Calendar` from `java.util` package are used to represent a date (and time)
  - Class `Date` is suitable for a simple current timestamp
  - Class `Calendar` provides better fields (e.g. `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR`, `MINUTE`, `SECOND`), for specific handling

```
Date curDT = new Date(); // represent current date & time
Calendar c = Calendar.getInstance(); // current date & time as Calendar
Date date =  c.getTime(); // return a Date object of this calendar's time
int year = c.get(Calendar.YEAR); // return the year of the calendar
int hour = c.get(Calendar.HOUR_OF_DAY); // return the hour: 0~23
// etc.
```

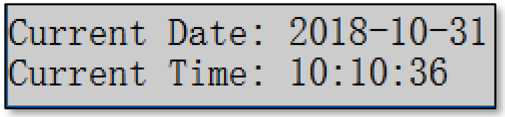
The Calendar Class
(In java.util package)

Reference Only

- Class Calendar fields (e.g. YEAR, MONTH, DAY_OF_MONTH, HOUR, MINUTE, SECOND)

```
Calendar c = Calendar.getInstance(); //get current time as Calendar
Date date =  c.getTime(); //return a Date object of Calendar's time
int year = c.get(Calendar.YEAR);
int month = 1 + c.get(Calendar.MONTH); // * RANGE: 0~11, for Jan~Dec
int day = c.get(Calendar.DAY_OF_MONTH);// 1~31
int hour = c.get(Calendar.HOUR_OF_DAY);// 0~23
int minute = c.get(Calendar.MINUTE);  // 0~59
int second = c.get(Calendar.SECOND);  // 0~59
System.out.println("Current Date: "+ year+ "-"+ month+ "-"+ day);
System.out.println("Current Time: "+ hour+ ":"+ minute+ ":"+ second);

long mSec = c.getTimeInMillis(); //return Calendar's time in millisec
c.setTimeInMillis(mSec); // set Calendar's time from a long value (ms)
c.setTime(date); // set Calendar's time from Date object
```

Sample output:



```
Current Date: 2018-10-31
Current Time: 10:10:36
```

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

An Application: Estimating the Execution Time

Reference Only

- To evaluate and compare program efficiency, we may measure how long it took to execute the program (sample code below)
  - Execution time can be measured easily by using the `Date` class

```
Date startTime = new Date(); // time BEFORE execution
//... code you want to measure the execution time
//... E.g. call a method to do certain work
Date endTime = new Date(); // time AFTER execution
long sTimeMS = startTime.getTime(); // Date's time in millisec
long eTimeMS = endTime.getTime();
long elapsedTimeInMilliSec = eTimeMS - sTimeMS;
```

  - May also get the `Date` object back from the millisecond value (e.g. `sTimeMS` above): **`Date orgDate = new Date(sTimeMS);`**

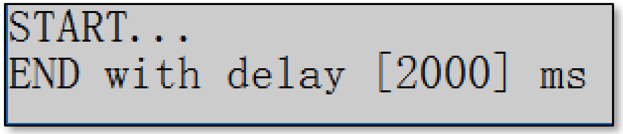
- For convenience of storing and handling the date and time in one value, we may use `System.currentTimeMillis()` to get the current time in milliseconds

Simple Delay, Using `Thread` Class
(In `java.lang` package)

Reference Only

- A thread (Java class **`Thread`**) is in particular useful concept for concurrent programming.
  - The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
  - Its Java method **`sleep()`** causes the currently executing thread to sleep (temporarily cease execution) for the specified number of *milliseconds*

```
System.out.println("START...");
Date startTime = new Date();
try {  Thread.sleep(2000); //Delay for about 2 seconds, with Thread
} catch (InterruptedException ie) {}
Date endTime = new Date();
long elapsedTimeInMilliSec = endTime.getTime() - startTime.getTime();
System.out.println("END with delay ["+ elapsedTimeInMilliSec + "] ms");
```



```
START...
END with delay [2000] ms
```

https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/Thread.html

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/