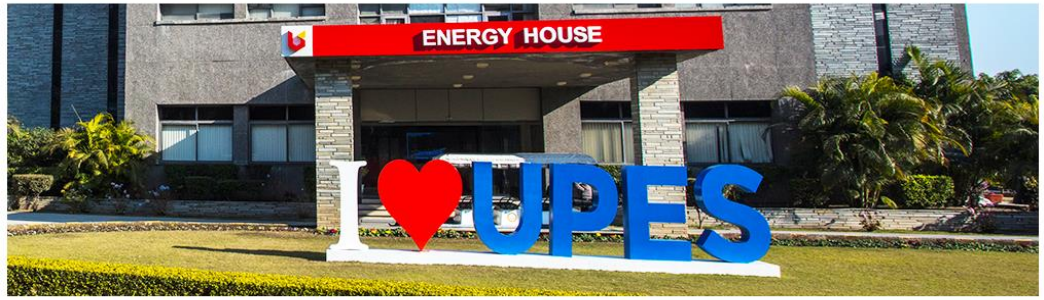

UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

UPES
UNIVERSITY WITH A PURPOSE

String Handling in Java

- Unlike some other languages that implement strings as character arrays, Java implements strings as objects of type `String`.
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- When we create a `String` object, you are creating a string that cannot be changed. That is, once a String object has been created, we cannot change the characters that comprise that string.
- This approach is used because fixed, **immutable** strings can be implemented more efficiently than changeable ones.
- For those cases in which a modifiable string is desired, Java provides two options: `StringBuffer and StringBuilder`. Both hold strings that can be modified after they are created.

String Handling in Java

- The `String, StringBuffer, and StringBuilder` classes are defined in `java.lang`. Thus, they are available to all programs automatically.
- To say that the strings within objects of type `String` are unchangeable means that the contents of the `String` instance cannot be changed after it has been created.
- However, a variable declared as a `String` reference can be changed to point at some other `String` object at any time.

The String Constructors

1. The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,

String s = new String();

2. To create a **String** initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

3. You can specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
//This initializes s with the characters cde.
```

The String Constructors

4. You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

`String(String strObj)`

`Here, strObj is a **String** object. Consider this example:`

```
// Construct one String from another.
class MakeString {
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}

//The output from this program is as follows:
Java
Java
```

The String Constructors

5. String class provides constructors that initialize a string when given a **byte** array. Two forms are shown here:

```
String(byte chrs[])
String(byte chrs[], int startIndex, int numChars)
```

Here, *chrs* specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
public static void main(String args[]) {
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);          System.out.println(s1);
String s2 = new String(ascii, 2, 3);  System.out.println(s2); }  }
This program generates the following output:
ABCDEF
CDE
```

UPES
UNIVERSITY WITH A PURPOSE

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method :

```
int length( )
```

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

UPES
UNIVERSITY WITH A PURPOSE

Special String Operations

- Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the **+** operator, and the conversion of other data types to a string representation.
- There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

Special String Operations: String Literals

- The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator.
- However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```

→ Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object.

→ For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length( )** method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

UPES
TY WITH A PURPOSE

Special String Operations: String Concatenation

- In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations.
- For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);

//This displays the string "He is 9 years old."
```

Special String Operations: String Concatenation with Other Data Types

- You can concatenate strings with other types of data.

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

- In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.
- Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

Special String Operations: String Concatenation with Other Data Types

Exp:

```
String s = "four: " + 2 + 2;
System.out.println(s);
This fragment displays
four: 22

rather than the
four: 4
```

→Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
Now s contains the string "four: 4".
```

JPES
UNIVERSITY WITH A PURPOSE

Special String Operations: String Conversion and toString( )

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**.
- **valueOf( )** is overloaded for all the primitive types and for type **Object**.
- For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called.
- For objects, **valueOf( )** calls the **toString( )** method on the object.
- Every class implements **toString( )** because it is defined by **Object**.
- The **toString( )** method has this general form:

```
String toString( )
```

Special String Operations: String Conversion and toString( )

```
// Override toString() for Box class.
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " + depth + " by "
+ height ;
}
}
```

```
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

//The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

→ As you can see, **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

UPES
UNIVERSITY WITH A PURPOSE

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.

UPES
UNIVERSITY WITH A PURPOSE