Lecture 26
Java provides an **automatic conversion feature** when the **+ operator** is used between:

- a *number typed data* and a *string typed data*

Java's **automatic** conversion rule for `number ⇒ string`:

- When the **+ operator** is used between
  - a *number* and
  - a *string*

  Then:
  - the *number* is **automatically converted** to a *string*
  - The **+ operator** is then **applied** on 2 *strings* (i.e., the **+ operator** is a *concatenation*!)

**toString function**
Automatic conversion of variables of a *user-defined* type to String

- Automatic conversion of variables of a *user-defined* type to string
  - Java also provides an automatic conversion feature when the + operator is used between:
    - a *user-defined* type variable (= object) and a string typed data

Java's automatic conversion rule for object → string:

- When the + operator is used between
  - an object reference variable and
  - a string

Then:
- the object referred to by the object reference variable is automatically converted to a string
- The + operator is then applied on 2 strings (i.e., the + operator will be a string concatenation)

- Example:
  ```java
  public class Class10
  { 
    public static void main(String[] args)
    { 
      String x;
      BankAccount stul = new BankAccount( 123, "John", 1000 );
      x = "Test --- " + stul;  // First converts stul to a String
      System.out.println( x );
    }
  }
  
  Output:
  Test --- BankAccount$130019b
  
  The string BankAccount$130019b is the result of the conversion of the object stul into the String type
Java's internal mechanism used to provide automatic object ⇒ String conversion

- The automatic conversion from an object type to the string type is accomplished by
  - A rule that is built-in into the Java compiler!!!

- Translating an expression containing the + operator by the Java compiler:

  The expression:
  ```java
  String + Object
  or
  Object + String
  ```
  is translated into:
  ```java
  String + Object  ⇒  String + Object.toString()
  or
  Object + String  ⇒  Object.toString() + String
  ```

Furthermore:

- Every user-defined type (= object) contains the method
  ```java
  public String toString()
  {
  ....
  }
  ```

- Advanced Java programming note:
  - The toString() method is inherited from the Super class Object
public class Class10 {
    public static void main(String[] args) {
        String x;
        BankAccount stu1 = new BankAccount(123, "John", 1000);
        x = "Test --- " + stu1; // Converted to: "Test --- " + stu1.toString()
        System.out.println(x);
    }
}

- **Explanation:**

  - **The relevant statement is:**
    
    ```java
    x = "Test --- " + stu1;  // String + (BankAccount) Object
    ```

  - **Due to the automatic string conversion rule**, the Java compiler will **translate** this statement into:
    
    ```java
    x = "Test --- " + stu1.toString(); // use "toString()" to convert object to String
    ```

  - **Our** **toString()** **method** defined inside the **BankAccount** **class** is now **invoked**, which returns the **string**:
    
    ```java
    Account number: 123, Name: John, Balance: 1000.0
    ```

    This **returned string** is then **concatenated** with "Test --- "

The String Class

- A String object is immutable: Its content cannot be changed once the string is created.

```java
String newString = new String(stringLiteral);

The argument stringLiteral is a sequence of characters enclosed inside double quotes. The following statement creates a String object message for the string literal "Welcome to Java":

String message = new String("Welcome to Java");

Java treats a string literal as a String object. Thus, the following statement is valid:

String message = "Welcome to Java";

You can also create a string from an array of characters. For example, the following statements create the string "Good Day":

char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```
A **String** object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```java
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a **String** object with the content "Java" and assigns its reference to `s`. The second statement creates a new **String** object with the content "HTML" and assigns its reference to `s`. The first **String** object still exists after the assignment, but it can no longer be accessed, because variable `s` now points to the new object, as shown in Figure 9.1.
Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an \textit{interned string}. For example, the following statements:

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is \"" + (s1 == s2));
System.out.println("s1 == s3 is \"" + (s1 == s3));
```

display

\begin{align*}
\text{s1 == s2 is false} \\
\text{s1 == s3 is true}
\end{align*}

In the preceding statements, \texttt{s1} and \texttt{s3} refer to the same interned string—\textit{"Welcome to Java"}—so \texttt{s1 == s3} is \texttt{true}. However, \texttt{s1 == s2} is \texttt{false}, because \texttt{s1} and \texttt{s2} are two different string objects, even though they have the same contents.
String Comparisons

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+equals(s1: Object): boolean</code></td>
<td>Returns true if this string is equal to string s1.</td>
</tr>
<tr>
<td><code>+equalsIgnoreCase(s1: String): boolean</code></td>
<td>Returns true if this string is equal to string s1 case insensitive.</td>
</tr>
<tr>
<td><code>+compareTo(s1: String): int</code></td>
<td>Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.</td>
</tr>
<tr>
<td><code>+compareToIgnoreCase(s1: String): int</code></td>
<td>Same as <code>compareTo</code> except that the comparison is case insensitive.</td>
</tr>
<tr>
<td><code>+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean</code></td>
<td>Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.</td>
</tr>
<tr>
<td><code>+regionMatches( ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean</code></td>
<td>Same as the preceding method except that you can specify whether the match is case sensitive.</td>
</tr>
<tr>
<td><code>+startsWith(prefix: String): boolean</code></td>
<td>Returns true if this string starts with the specified prefix.</td>
</tr>
<tr>
<td><code>+endsWith(suffix: String): boolean</code></td>
<td>Returns true if this string ends with the specified suffix.</td>
</tr>
</tbody>
</table>

The `compareTo` method can also be used to compare two strings. For example, consider the following code:

```java
s1.compareTo(s2)
```

The method returns the value 0 if `s1` is equal to `s2`, a value less than 0 if `s1` is lexicographically (i.e., in terms of Unicode ordering) less than `s2`, and a value greater than 0 if `s1` is lexicographically greater than `s2`.

The actual value returned from the `compareTo` method depends on the offset of the first two distinct characters in `s1` and `s2` from left to right. For example, suppose `s1` is `abc` and `s2` is `abg`, and `s1.compareTo(s2)` returns -4. The first two characters (a vs. a) from `s1` and `s2` are compared. Because they are equal, the second two characters (b vs. b) are compared. Because they are also equal, the third two characters (c vs. g) are compared. Since the character c is 4 less than g, the comparison returns -4.

**Caution**

Syntax errors will occur if you compare strings by using comparison operators `>`, `>=`, `<`, or `<=`. Instead, you have to use `s1.compareTo(s2)`.

**Note**

The `equals` method returns `true` if two strings are equal and `false` if they are not. The `compareTo` method returns 0, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.
Getting String Length and Characters, and Combining Strings

**Caution**

`length` is a method in the `String` class but is a property of an array object. Therefore, you have to use `s.length()` to get the number of characters in string `s`, and `a.length` to get the number of elements in array `a`.

The `s.charAt(index)` method can be used to retrieve a specific character in a string `s`, where the index is between 0 and `s.length()-1`. For example, `message.charAt(0)` returns the character `W`, as shown in Figure 9.4.

**Note**

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, "Welcome to Java".charAt(0) is correct and returns `W`.

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>W</td>
<td>e</td>
<td>l</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>e</td>
<td>t</td>
<td>o</td>
<td>J</td>
<td>a</td>
<td>v</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`message.charAt(0)`

`message.length()` is 15

`message.charAt(14)`
Obtaining Substrings

```java
java.lang.String
+substring(beginIndex: int): String
+substring(beginIndex: int, endIndex: int): String
```

Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 9.6.

Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.

Indices
---
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Message: `Welcome to Java`

- `message.substring(0, 11)`
- `message.substring(11)`
Converting, Replacing, and Splitting Strings

```
java.lang.String
+toLowerCase(): String
+toUpperCase(): String
+trim(): String
+replace(oldChar: char, newChar: char): String
+replaceFirst(oldString: String, newString: String): String
+replaceAll(oldString: String, newString: String): String
+split(delimiter: String): String[]
```

- `toLowerCase()`: Returns a new string with all characters converted to lowercase.
- `toUpperCase()`: Returns a new string with all characters converted to uppercase.
- `trim()`: Returns a new string with whitespace characters trimmed on both sides.
- `replace(oldChar, newChar)`: Returns a new string that replaces all matching characters in this string with the new character.
- `replaceFirst(oldString, newString)`: Returns a new string that replaces the first matching substring in this string with the new substring.
- `replaceAll(oldString, newString)`: Returns a new string that replaces all matching substrings in this string with the new substring.
- `split(delimiter)`: Returns an array of strings consisting of the substrings split by the delimiter.

```
"Welcome". toLowerCase() returns a new string, welcome.
"Welcome". toUpperCase() returns a new string, WELCOME.
"\t Good Night \n". trim() returns a new string, Good Night.
"Welcome". replace('e', 'A') returns a new string, WALcomA.
"Welcome". replaceFirst("e", "AB") returns a new string, WAB1come.
"Welcome". replace("e", "AB") returns a new string, WAB1comAB.
"Welcome". replace("el", "AB") returns a new string, WABcome.
```

The `split` method can be used to extract tokens from a string with the specified delimiters. For example, the following code:

```java
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
    System.out.print(tokens[i] + " ");
```

displays

Java HTML Perl
Finding a Character or a Substring in a String

```java
java.lang.String

+indexOf(ch: char): int
+indexOf(ch: char, fromIndex: int): int
+indexOf(s: String): int
+indexOf(s: String, fromIndex: int): int
+lastIndexOf(ch: int): int
+lastIndexOf(ch: int, fromIndex: int): int
+lastIndexOf(s: String): int
+lastIndexOf(s: String, fromIndex: int): int

Returns the index of the first occurrence of ch in the string.
  Returns -1 if not matched.
Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
Returns the index of the first occurrence of string s in this string.
  Returns -1 if not matched.
Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
Returns the index of the last occurrence of ch in the string.
  Returns -1 if not matched.
Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
Returns the index of the last occurrence of string s in this string. Returns -1 if not matched.
Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.
```

```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('o') returns 4.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.

"Welcome to Java".lastIndexOf('W') returns 0.
"Welcome to Java".lastIndexOf('o') returns 9.
"Welcome to Java".lastIndexOf('o', 5) returns 4.
"Welcome to Java".lastIndexOf("come") returns 3.
"Welcome to Java".lastIndexOf("Java", 5) returns -1.
"Welcome to Java".lastIndexOf("Java") returns 11.
```
Conversion between Strings and Arrays

```java
char[] chars = "Java".toCharArray();

char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);

Thus, dst becomes {'J', 'A', 'V', 'A', '3', '7', '2', '0'}.

String str = new String(new char[]{'J', 'a', 'v', 'a'});

String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```
Converting Characters and Numeric Values to Strings

```
java.lang.String
+valueOf(c: char): String
+valueOf(data: char[]): String
+valueOf(d: double): String
+valueOf(f: float): String
+valueOf(i: int): String
+valueOf(l: long): String
+valueOf(b: boolean): String
```

- `valueOf(c: char): String`: Returns a string consisting of the character `c`.
- `valueOf(data: char[]): String`: Returns a string consisting of the characters in the array.
- `valueOf(d: double): String`: Returns a string representing the `double` value.
- `valueOf(f: float): String`: Returns a string representing the `float` value.
- `valueOf(i: int): String`: Returns a string representing the `int` value.
- `valueOf(l: long): String`: Returns a string representing the `long` value.
- `valueOf(b: boolean): String`: Returns a string representing the `boolean` value.
The `String` class contains the static `format` method to create a formatted string. The syntax to invoke this method is:

```java
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```java
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
45.56□□14AB□□
```

Note that

```java
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```java
System.out.printf(
    String.format(format, item1, item2, ..., itemk));
```

where the square box (□) denotes a blank space.