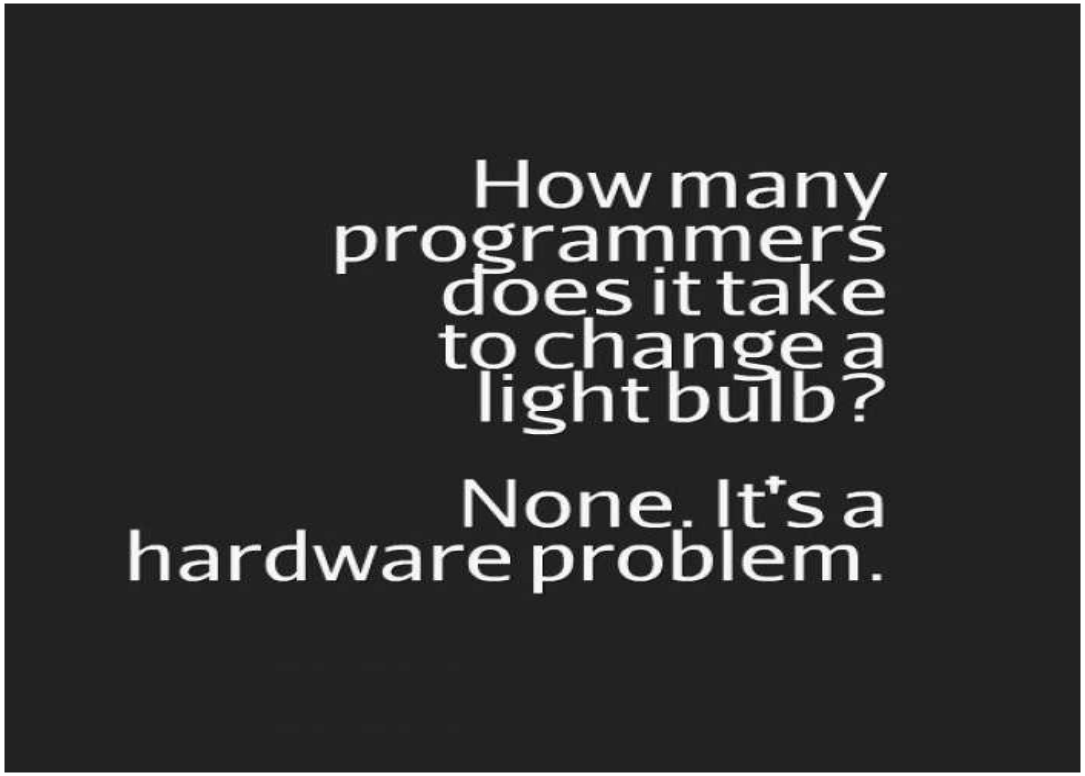
**Keenan Knaur**
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

# Object-Oriented Programming III and Strings Review


How many programmers does it take to change a light bulb?
None. It's a hardware problem.

**CS2012: Introduction to Programming II**

## Immutable Objects and Classes

## Immutable Objects and Classes

- Sometimes you may want an object whose data cannot be changed once it has been created.
  - this is called an ***immutable object*** and its class an ***immutable class***

- The `String` class is such a class
  - once a string is created, its content cannot be changed.
  - changing a string actually involves creating a new string object in memory

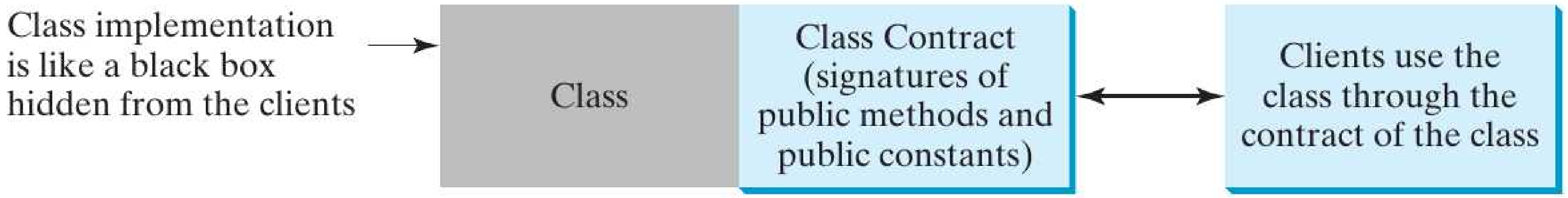
- For a class to be immutable, it must follow these rules:
  - all data fields must be private
  - there can't be any mutator methods for data fields
  - no accessor methods can return a reference to a data field that is mutable.

## Abstraction

## Abstraction

- ***abstraction*** means separating the class implementation (how the class actually works) from how the class is used.
  - When you create an instance of Scanner, or when you use the Math class, you don't know exactly how Math or Scanner work (are implemented) but you still now how to use them.
- ***public interface of a class (class contract)***: the collection of methods and data fields that are accessible from outside of the class (public methods and / or data fields) coupled together with the description of how these members behave.
- A user of a class doesn't (and shouldn't) need to know HOW the class actually works "under the hood".
- We say the details of implementation are ***abstracted*** and hidden from the user of the class.

# Abstraction



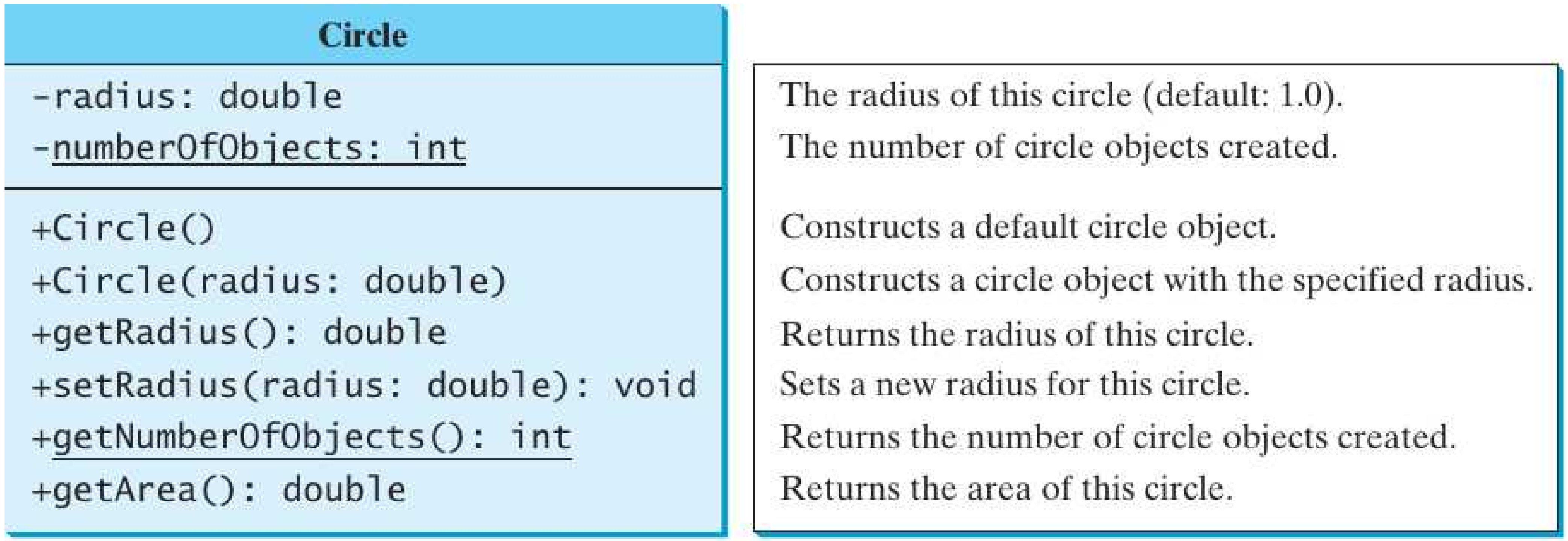
Class abstraction separates class implementation from the use of the class.

## UML Diagrams

## UML Diagrams

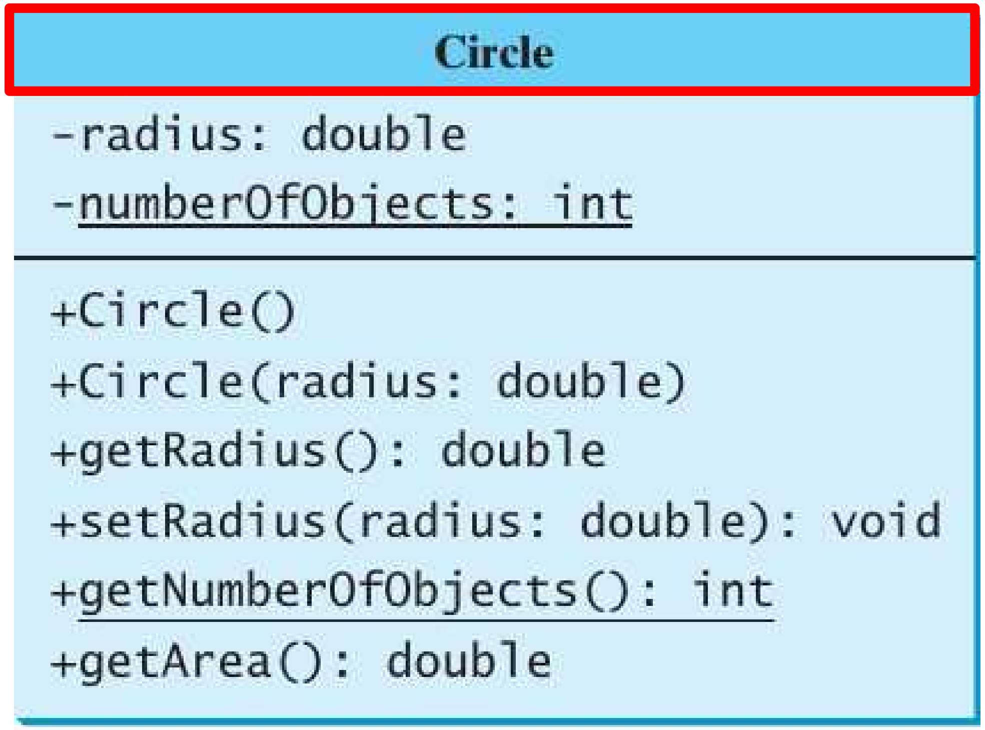
- Classes can be designed using a UML (Unified Modeling Language) Diagram.
  - data fields are denoted as:
    - `dataFieldName: dataFieldType`
  - constructors are denoted as:
    - `ClassName(parameterName: parameterType)`
  - methods are denoted as:
    - `methodName(parameterName: parameterType): returnType`

## UML Diagrams


Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

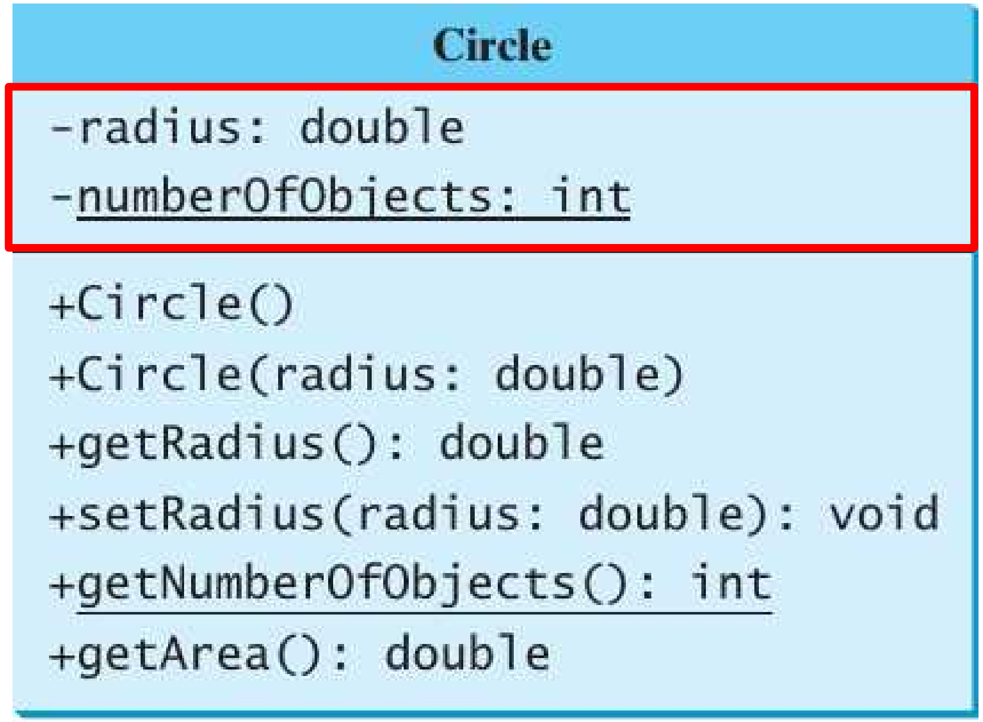
## UML Diagrams

- Class name goes at the top of the UML diagram.
- Class name should be centered.
- Draw a line separating the class name from the data fields.



## UML Diagrams

- Data fields should be listed next.
  - `dataFieldName: datatype`
- Indicate access modifier
  - + for public
  - - for private
  - # for protected
- Underline any static data fields
- Draw a line between the data fields and the methods



# UML Diagrams

- Constructors should be listed first.

  `ClassName(name1: type1, name2: type2, etc.)`

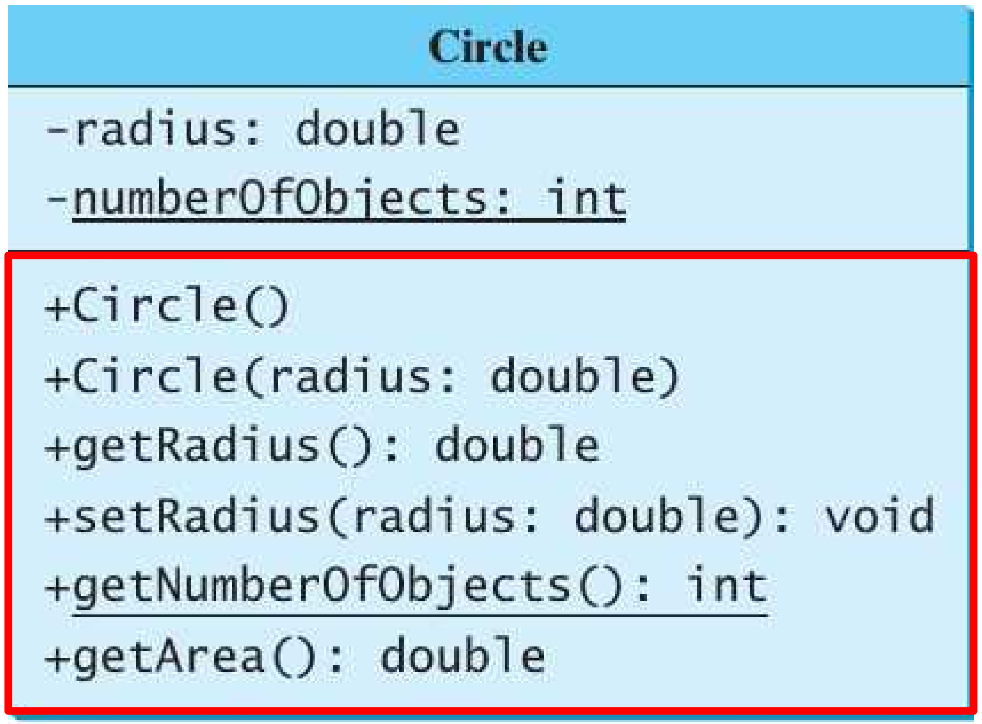
  - list all parameters and their data types
  - indicate access modifier

- Methods come next

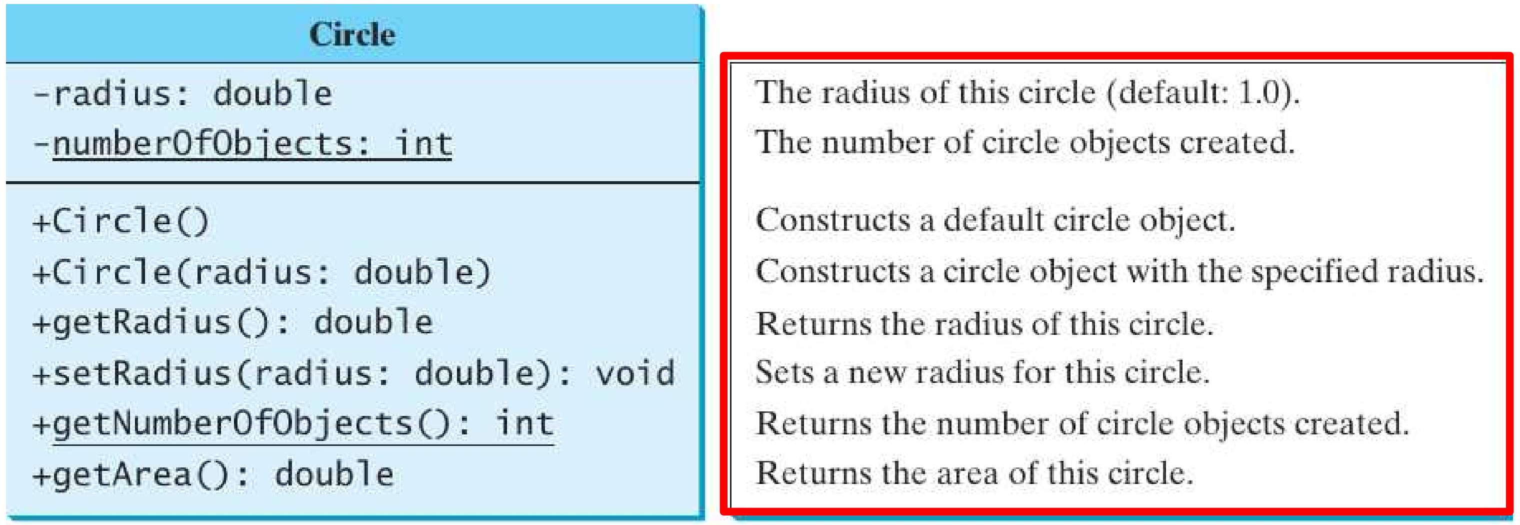
  `methodName(name1: type1, name2: type2, etc.): returnType`

  - list all parameters
  - indicate the return type
  - indicate access modifier

- Underline any static methods



# UML Diagrams



```
-radius: double
-numberOfObjects: int

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
```

The radius of this circle (default: 1.0).
The number of circle objects created.

Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

- Data Fields:
  - Give a one sentence description of the data field.
  - Indicate if the data field has a default value or not
- Constructors:
  - Give a one sentence description of what that constructor does.
- Methods:
  - Give a one sentence description of what that method does.

## BigInteger and BigDecimal

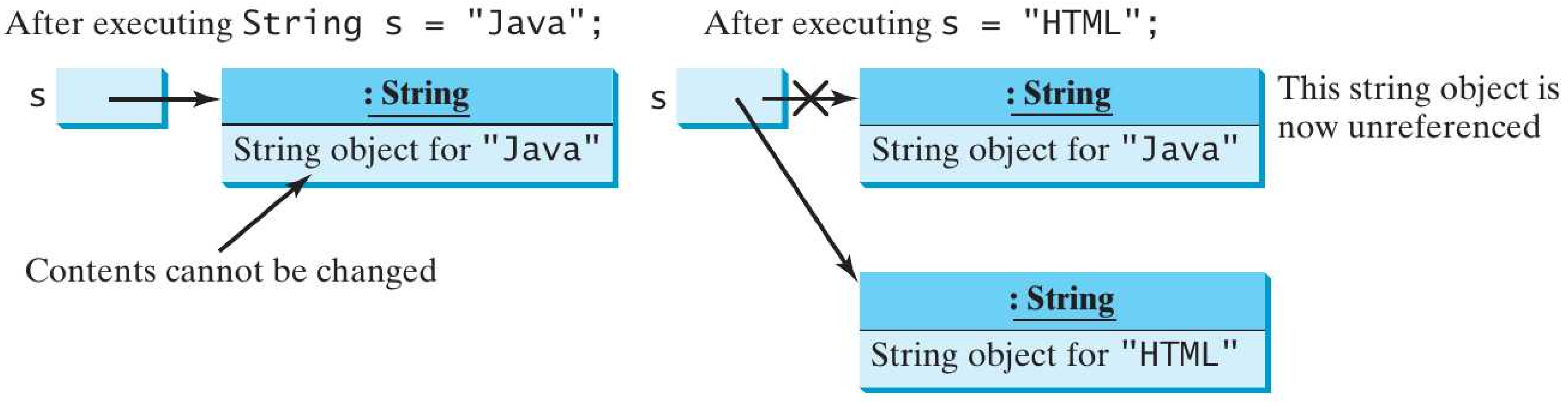
- `BigInteger` and `BigDecimal` can be used to represent integers or decimal numbers of any size or precision.
- You would use these when the long or double types are not enough.
- For more information read about them in the Java API.

## Strings

# Strings

- A ***String*** is an immutable sequence of characters.
  - changing the value of a String means creating a new String in memory, not editing the old value.

- Example:
  - **`String s1 = "java";`**
  - **`s1 = "html";`**
    - This does not change "java" to "html", the old string becomes garbage and s1 now points to the "html" string.


After executing String s = "Java";
After executing s = "HTML";
s
: String
String object for "Java"
Contents cannot be changed
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Strings are immutable; once created, their contents cannot be changed.

## Constructing a String

- Strings can be created from a string literal or from an array of characters.

- Creating a string from a literal:
  - Syntax:
    - `String newString = new String(stringLiteral);`
    - `String newString = stringLiteral;`
  - Example:
    - `String message = new String("CS-202 is Fun!!!");`
    - `String message = "CS-202 is Fun!!!";`

- Creating a string from an array of characters:
  - Example:
    - `char[] charArray = {'H','e','y',' ','W','o','r','l','d'};`
    - `String message = new String(charArray);`

- A String variable, holds a reference to a String object which stores the String value.

- Strings are object types, not primitive types.

## Interned Strings

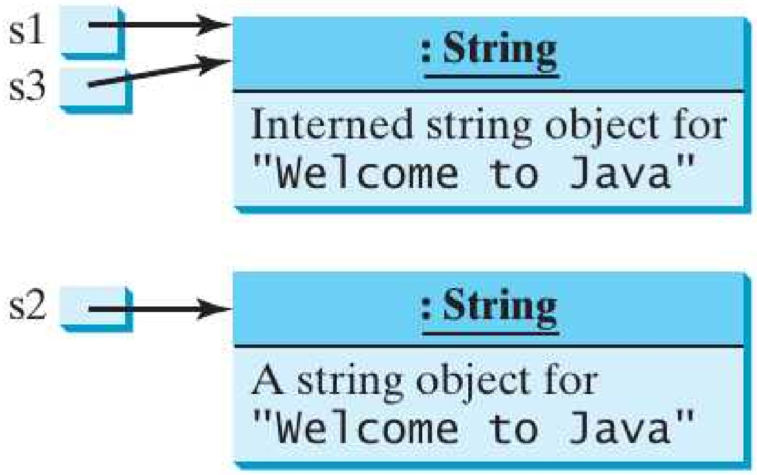
- The JVM uses a ***unique instance*** for string literals with the same character sequence.
  - this is called an ***interned string***
  - imagine there is a "pool" of string literals.  If you make two string objects from the same exact string literal, then both string references would point to the same literal in memory.

```
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));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

display

```
  s1 == s2 is false
  s1 == s3 is true
```

## String Comparisons

- How can you compare two strings?

- Will the following code work?
  - HINT: Strings are objects.

```
if (string1 == string2) {
  System.out.println("string 1 and string2 are the
                                  same object");
}
else {
  System.out.println("string 1 and string2 are
                                  different objects");
}
```

## String Comparisons

- Answer: No, the == operator only checks whether `string1` and `string2` refer (reference) the same object, it does not check whether the contents are the same.

- The correct way to check whether two strings are equal is to use the `equals()` method of the String class.

```
if (string1.equals(string2)) {
   System.out.println("string 1 and string2 have the
                                 same contents");
}
else {
   System.out.println("string 1 and string2 are not
                                 equal");
}
```

## String Comparisons

- The `equals()` method returns true or false.

- Another way to compare strings is to use the `compareTo()` method which returns a numeric value
  - returns 0 if s1 == s2
  - returns a value < 0 if s1 is lexicographically less than s2
  - returns a value > 0 if s1 is lexicographically greater than s2

- `lexicographically` means in terms of ordering based on the Unicode values of the characters.

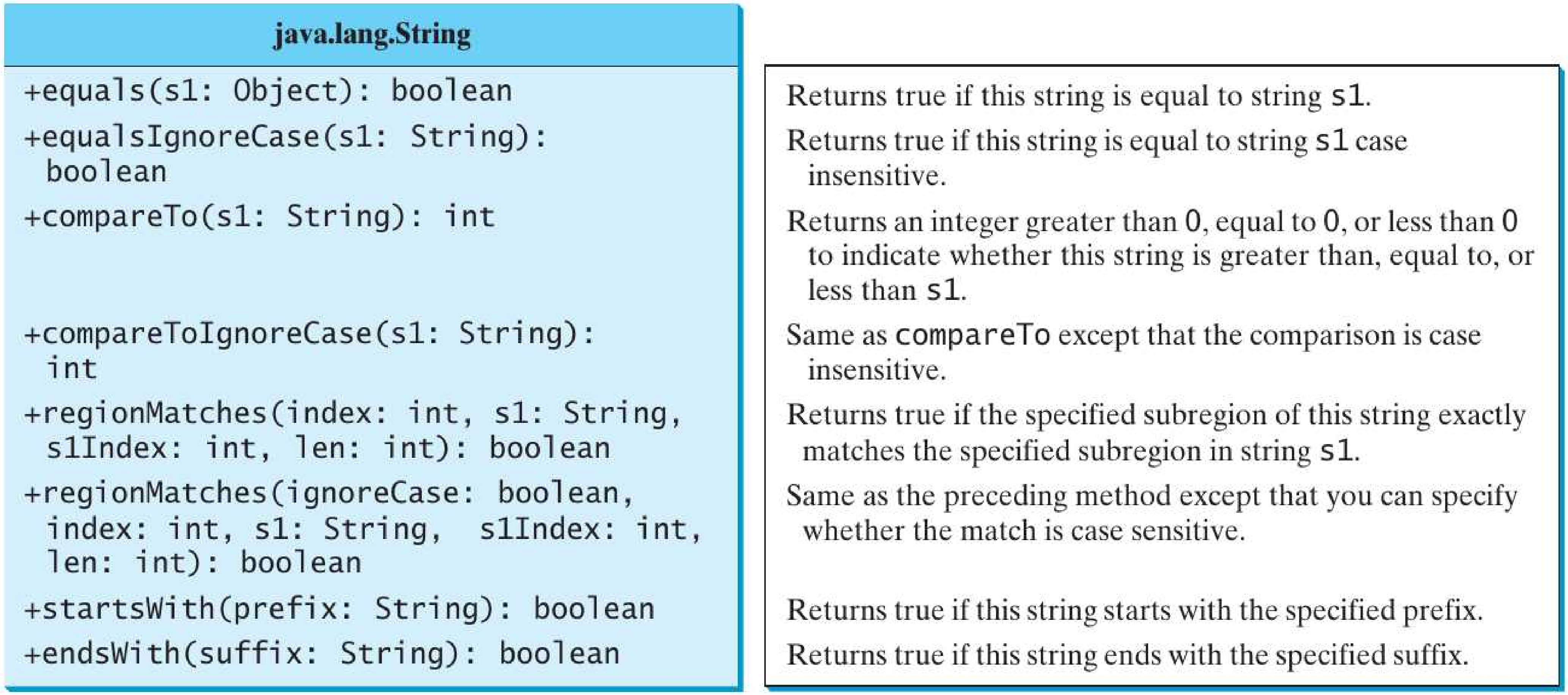
## String Comparisons

- Value returned depends on offset of the first two differing characters in the string from left to right
- 
- Example:

```
s1 = "abc"
s2 = "abg"
s1.compareTo(s2) returns -4
```

- How the comparison works:
  - compare 'a' from s1 to 'a' from s2 (they are the same so move on)
  - compare 'b' from s1 to 'b' from s2 (they are the same so move on)
  - compare 'c' from s1 to 'g' from s2 ('c' is 4 Unicode values less than 'g' so -4 is returned)
- If you try to compare strings with >, >=, <, <= you will get a syntax error

# String Comparisons



| java.lang.String | |
|---|---|
| `+equals(s1: Object): boolean` | Returns true if this string is equal to string `s1`. |
| `+equalsIgnoreCase(s1: String): boolean` | Returns true if this string is equal to string `s1` case insensitive. |
| `+compareTo(s1: String): int` | 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`. |
| `+compareToIgnoreCase(s1: String): int` | Same as `compareTo` except that the comparison is case insensitive. |
| `+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean` | Returns true if the specified subregion of this string exactly matches the specified subregion in string `s1`. |
| `+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean` | Same as the preceding method except that you can specify whether the match is case sensitive. |
| `+startsWith(prefix: String): boolean` | Returns true if this string starts with the specified prefix. |
| `+endsWith(suffix: String): boolean` | Returns true if this string ends with the specified suffix. |

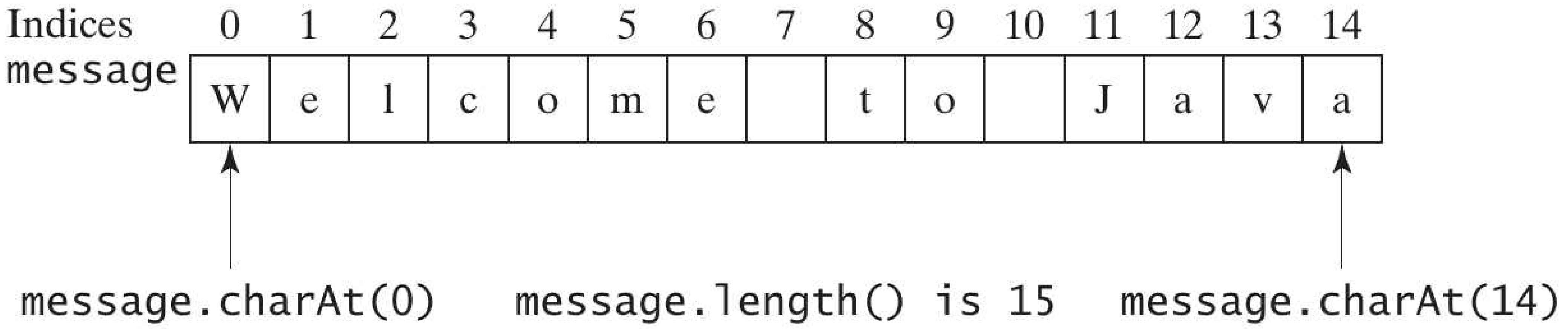
## Length

- The ***size*** or ***length*** of a string is the number of characters in the string.
  - This includes whitespace characters.
- You can get the length of a string using the `.length()` method of the String class.
- NOTE: Don't confuse the .length property of an array with the `.length()` method of a String. One is a property, the other is a method.

# Retrieving a Specific Character

- The `.charAt(index)` method can get a specific character in a string s.
  - index is a value between 0 and s.length() - 1
  - same as the index rules for an array.
  - accessing a character outside the range causes a `StringIndexOutOfBoundsException`

- The String class actually uses an array to store characters internally. The array is private and cannot be accessed outside of the String class. This is a classic example of encapsulation.



The characters in a String object are stored using an array internally.

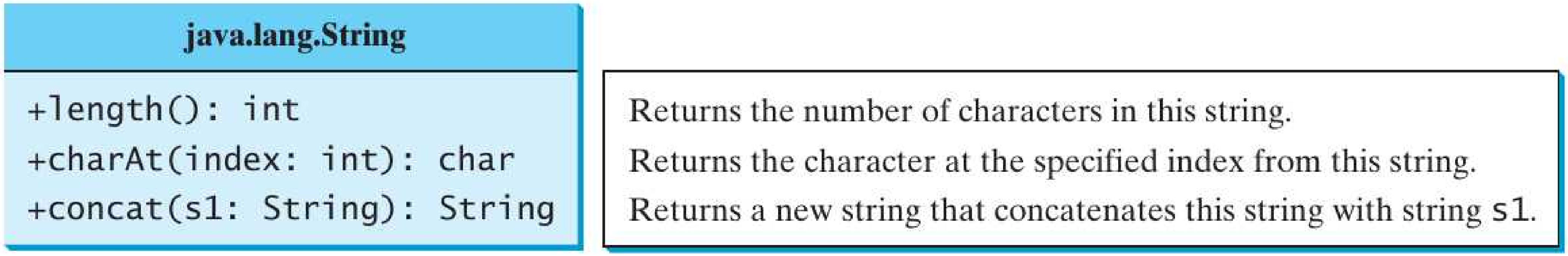
## Concatenation

- The `.concat()` method can be used to combine two strings.
  - Example: `String s3 = s1.concat(s2);`

- Alternatively you can use the + operator to concatenate two strings
  - Example: `String s3 = s1 + s2;`

- Remember this can be done with strings and other types as well, as long as the type can be automatically converted to a String type.

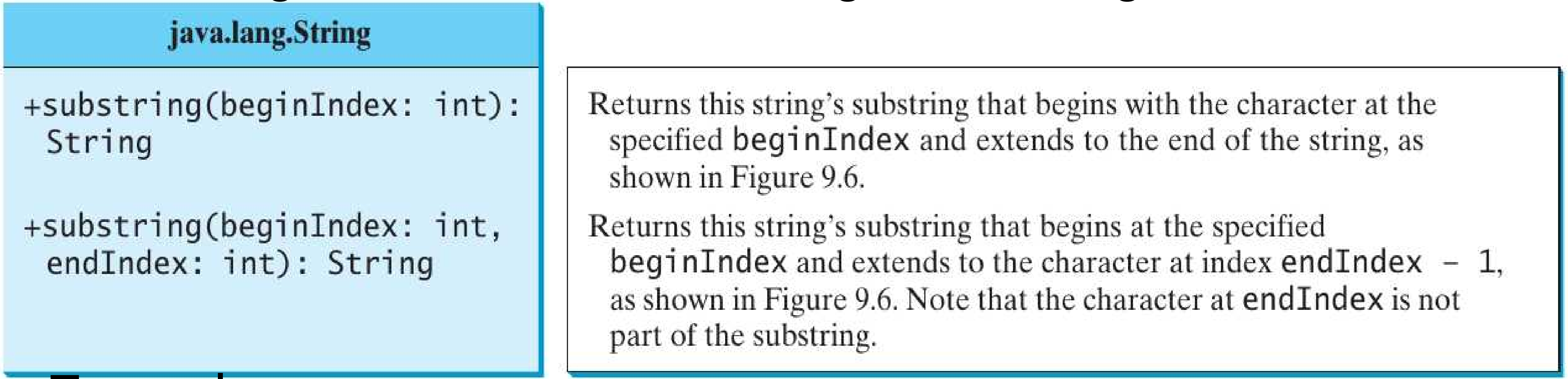
## Length, charAt, and Concat



| java.lang.String | |
|---|---|
| `+length(): int` | Returns the number of characters in this string. |
| `+charAt(index: int): char` | Returns the character at the specified index from this string. |
| `+concat(s1: String): String` | Returns a new string that concatenates this string with string `s1`. |

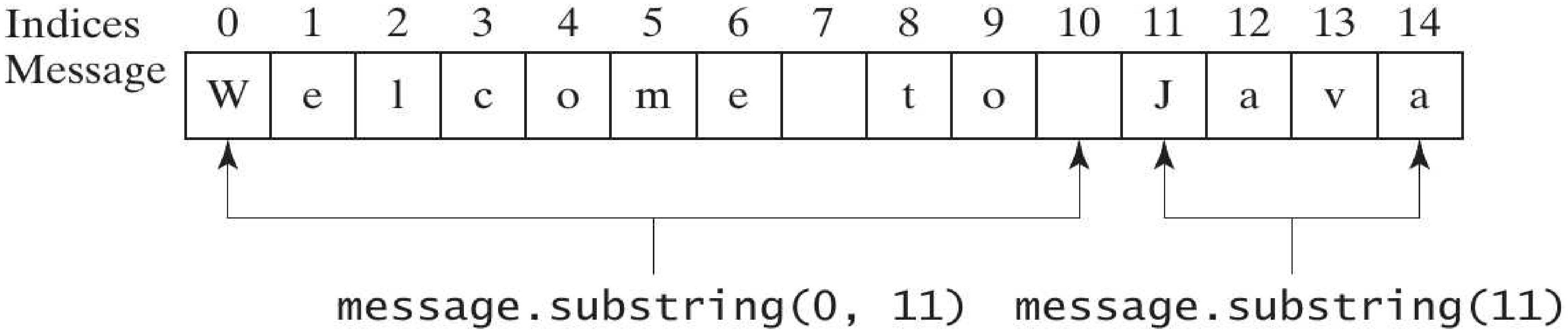
# Substrings

- The String class has methods to get substrings.



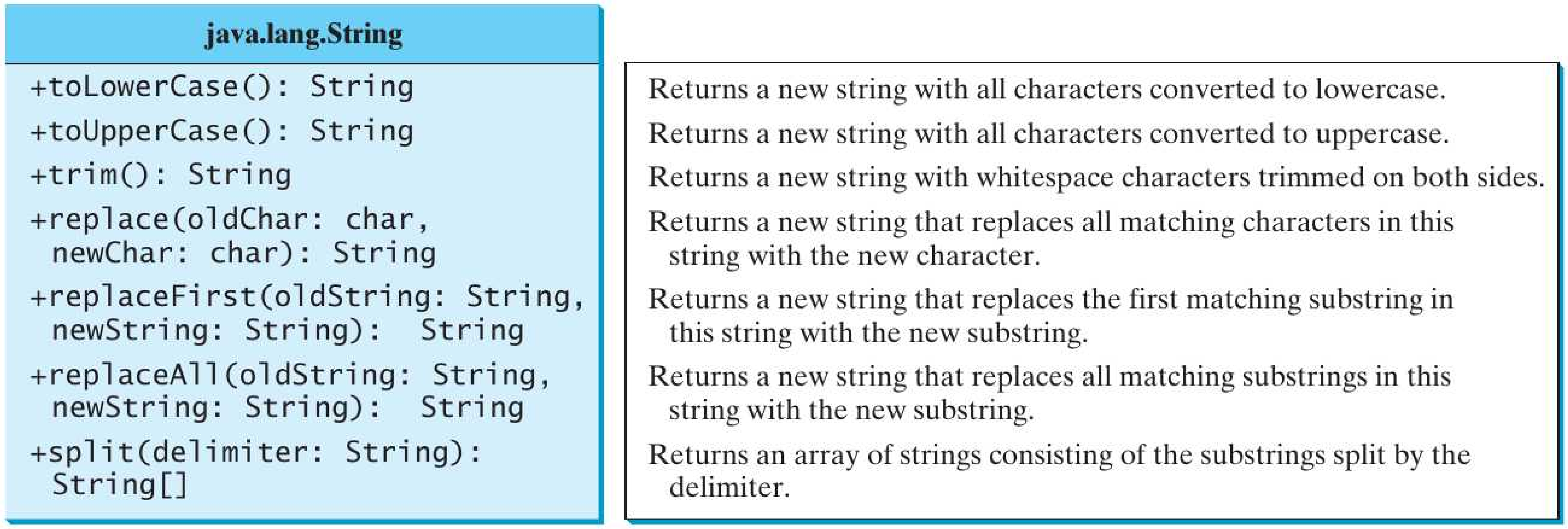
| java.lang.String | |
|---|---|
| `+substring(beginIndex: int): String` | Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 9.6. |
| `+substring(beginIndex: int, endIndex: int): String` | Returns this 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. |

- Example:
  - `String message = "Welcome to Java".substring(0, 11) + "HTML"`
  - `// message is now Welcome to HTML`



## Converting, Replacing, and Splitting Strings

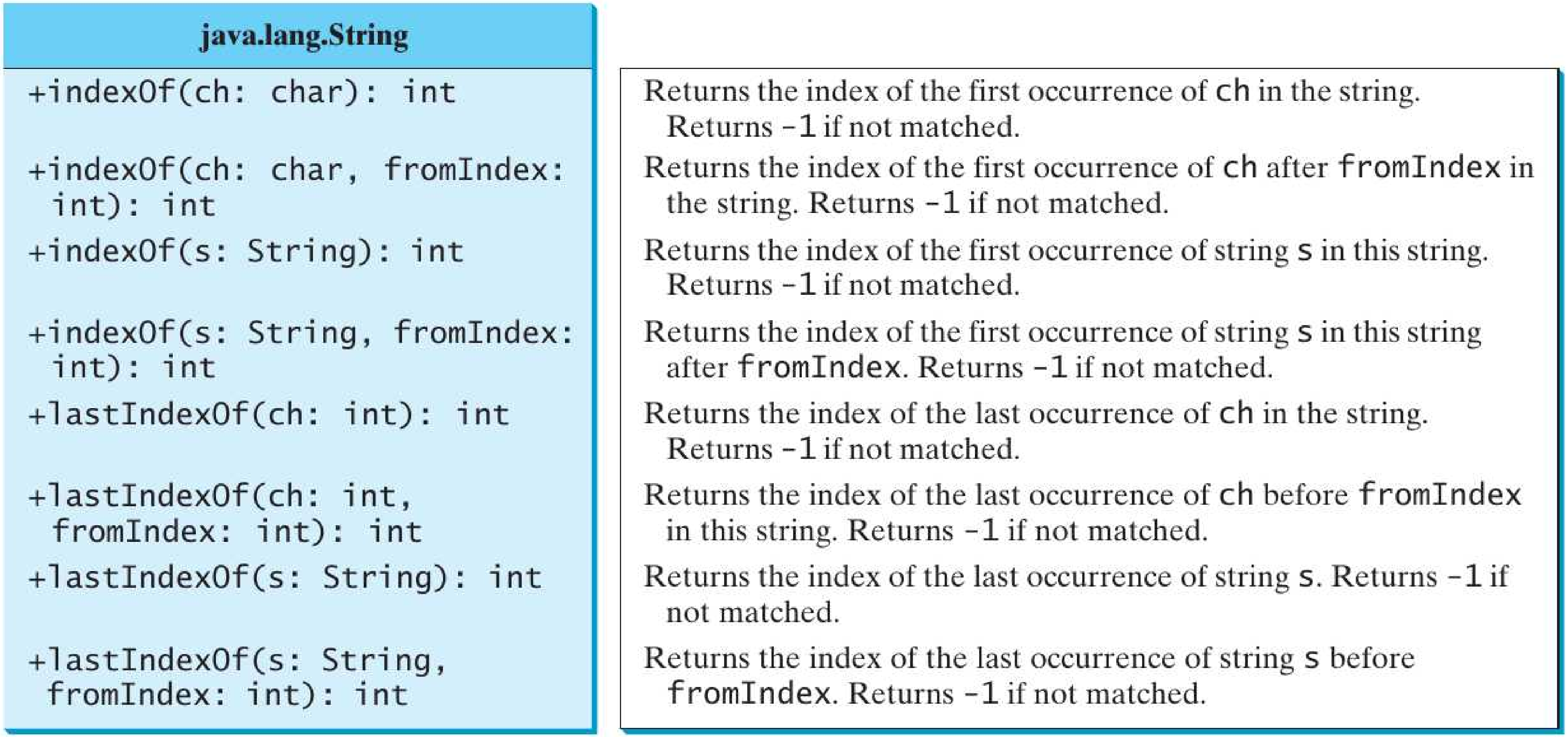
- Some methods are available to convert, replace and split strings.
  - returns a new String, still does not alter the original string



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

# Finding a Character or Substring

- The String class provides several overloaded indexOf() and lastIndexOf() methods to find a character or substring in a string.



| java.lang.String | |
|---|---|
| `+indexOf(ch: char): int` | Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched. |
| `+indexOf(ch: char, fromIndex: int): int` | Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched. |
| `+indexOf(s: String): int` | Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched. |
| `+indexOf(s: String, fromIndex: int): int` | Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched. |
| `+lastIndexOf(ch: int): int` | Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched. |
| `+lastIndexOf(ch: int, fromIndex: int): int` | Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched. |
| `+lastIndexOf(s: String): int` | Returns the index of the last occurrence of string `s`. Returns `-1` if not matched. |
| `+lastIndexOf(s: String, fromIndex: int): int` | Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched. |

## Convert between Strings and Arrays

- Strings are not arrays, but can be converted into an array and vice versa

- Convert a String "Java" to an array of length 4 where each element is one character of the string:
  - `char[] chars = "Java".toCharArray();`

## Convert between Strings and Arrays

- `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  - method to copy a substring of a string into an array
  - `srcBegin` the index of the string where you want to start copying
  - `srcEnd - 1` the ending index of the string where you want to stop
  - `char[] dst` the array to copy into
  - `dstBegin` the index of the dst array where to start copying into

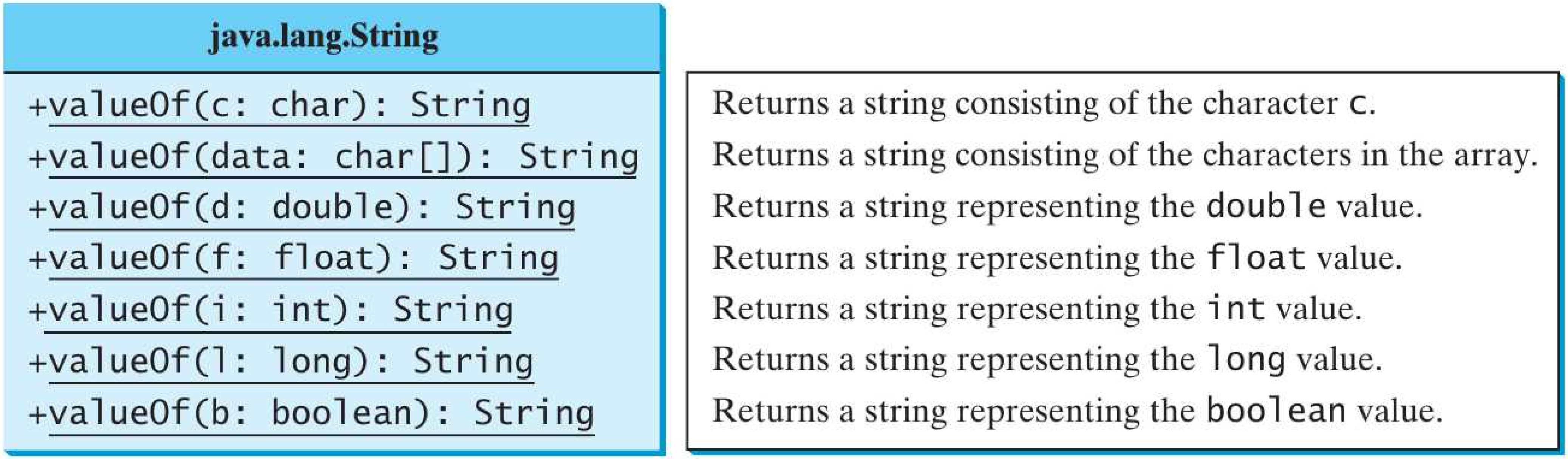
```
char[] dst = {'J','A','V','A','1','3','0','1'}
"CS3720".getChars(2, 6, dst, 4);
//dst is now {'J','A','V','A','3','7','2','0'}
```

## Conversion Between Strings and Arrays

- To convert an array to a string there are two ways
- use the String(char[]) constructor:
  - String str = new String(new char[]{'J','a','v','a'});
- use the .valueOf() method:
  - String str = String.valueOf(new char[]{'J','a','v','a'});

## Converting Characters and Numeric Values to Strings

- shown previously, the static .valueOf() method can be used to covert an array of characters to a string.
- this method is overloaded to work with different parameter types.
- NOTE: this can also be done by concatenating any of these types with a string.



| java.lang.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. |

## Formatting Strings

- The static `.format()` method can be used to format a string
  - Syntax: `String.format(format, item1, item2,..., itemk)`

- Similar to the `printf()` method except format() returns a formatted string, `printf()` displays the formatted string.

- Example:

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

//displays □□45.56□□□□14AB□□ □ = space

- NOTE: You may need to review what a format specifier is and how they are used.

## StringBuilder and String Buffer

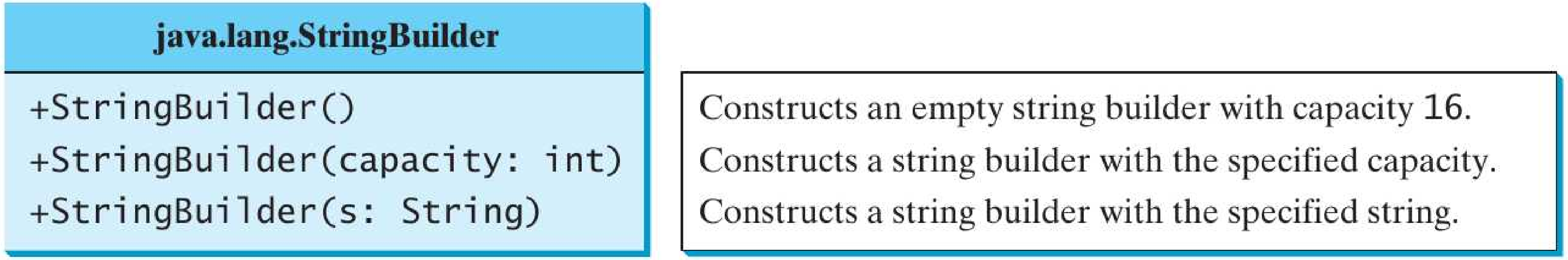
## StringBuilder and StringBuffer

- These can be used wherever a String is used, but are generally more flexible than strings

- Can add, insert, or append new contents into StringBuilder and StringBuffer objects whereas String objects are immutable.

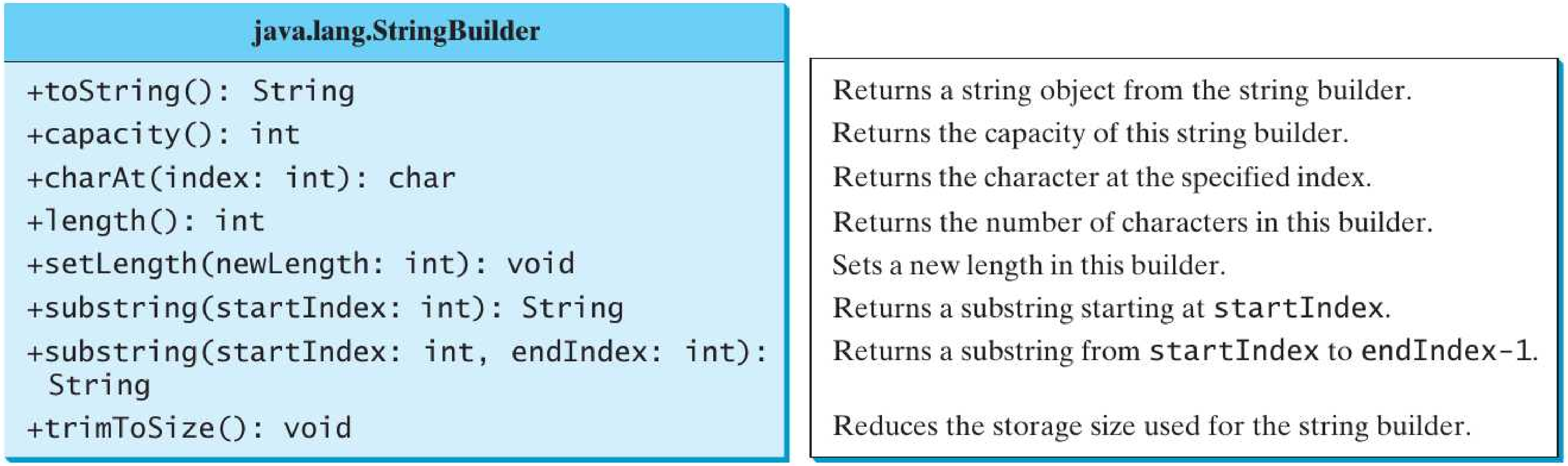
- StringBuilder and StringBuffer are similar except:
  - all methods in StringBuffer that modify the buffer are synchronized (only one task is allowed to execute the methods at a time when using multithreading)

- When to use:
  - StringBuffer: if you are using multithreading
  - StringBuilder: if it will only be accessed by a single task at a time (no multithreading)
  - These can be interchanged as needed.



| java.lang.StringBuilder | |
|---|---|
| `+StringBuilder()` | Constructs an empty string builder with capacity 16. |
| `+StringBuilder(capacity: int)` | Constructs a string builder with the specified capacity. |
| `+StringBuilder(s: String)` | Constructs a string builder with the specified string. |

| java.lang.StringBuilder | |
|---|---|
| `+append(data: char[]): StringBuilder` | Appends a `char` array into this string builder. |
| `+append(data: char[], offset: int, len: int): StringBuilder` | Appends a subarray in `data` into this string builder. |
| `+append(v: `*`aPrimitiveType`*`): StringBuilder` | Appends a primitive type value as a string to this builder. |
| `+append(s: String): StringBuilder` | Appends a string to this string builder. |
| `+delete(startIndex: int, endIndex: int): StringBuilder` | Deletes characters from `startIndex` to `endIndex-1`. |
| `+deleteCharAt(index: int): StringBuilder` | Deletes a character at the specified index. |
| `+insert(index: int, data: char[], offset: int, len: int): StringBuilder` | Inserts a subarray of the data in the array into the builder at the specified index. |
| `+insert(offset: int, data: char[]): StringBuilder` | Inserts data into this builder at the position offset. |
| `+insert(offset: int, b: `*`aPrimitiveType`*`): StringBuilder` | Inserts a value converted to a string into this builder. |
| `+insert(offset: int, s: String): StringBuilder` | Inserts a string into this builder at the position offset. |
| `+replace(startIndex: int, endIndex: int, s: String): StringBuilder` | Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string. |
| `+reverse(): StringBuilder` | Reverses the characters in the builder. |
| `+setCharAt(index: int, ch: char): void` | Sets a new character at the specified index in this builder. |



| java.lang.StringBuilder | |
|---|---|
| `+toString(): String` | Returns a string object from the string builder. |
| `+capacity(): int` | Returns the capacity of this string builder. |
| `+charAt(index: int): char` | Returns the character at the specified index. |
| `+length(): int` | Returns the number of characters in this builder. |
| `+setLength(newLength: int): void` | Sets a new length in this builder. |
| `+substring(startIndex: int): String` | Returns a substring starting at `startIndex`. |
| `+substring(startIndex: int, endIndex: int): String` | Returns a substring from `startIndex` to `endIndex-1`. |
| `+trimToSize(): void` | Reduces the storage size used for the string builder. |

## Command-Line Arguments

# Command-Line Arguments

- The main method can receive String arguments from the command line.
- Recall the header for main:
- `  public static void main(String[] args)`
- Now you should be able to understand the parameter of the main method as an array of `String` objects, but what are these objects for?
- You can pass "command-line arguments" to a program when you compile it from the command line
- these arguments are stored in the args array and can be accessed in your program.

- How does one pass command-line arguments?
- Example: Compiling from the command-line

```
java TestMan arg0 arg1 arg2


java TestMain "First num" alpha 53
```

- The program takes three string inputs "First num", "alpha" and 53.
- when main is invoked you can access these arguments through the args string array.
- NOTE: if you don't pass any arguments the array is created with new String[0] (an empty array with length 0 and args references the empty array (not NULL))

- Why use these?
  - it harkens back to the days before IDEs when everything was done on the command-line.
  - You could pass options which would determine how your program would initially execute.

## References

- Liang, Chapter 09: Classes and Objects
- Liang, Chapter 10: Thinking in Objects