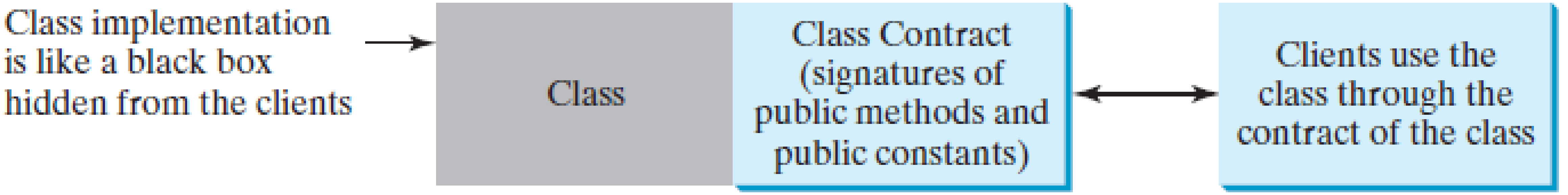
OBJECT-ORIENTED THINKING

Class Abstraction and Encapsulation

- *Class abstraction is the separation of class implementation from the use of a class*
- *details of implementation are encapsulated and hidden from the user, known as class encapsulation*
- user of the class does not need to know how the class is implemented
- class is also known as an *abstract data type* (ADT)

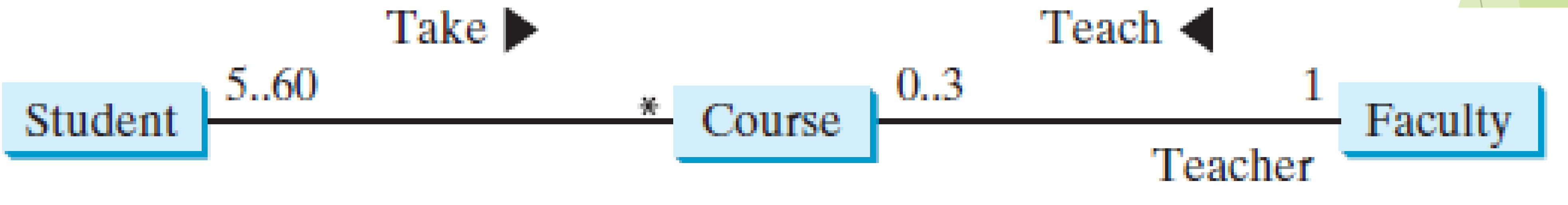

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

Thinking in Objects

- *procedural paradigm focuses on designing methods*
- *object-oriented paradigm couples data and methods together into objects*
- procedural programming, data and operations on the data are separate, and this methodology requires passing data to methods
- Object-oriented programming places data and the operations that pertain to them in an object

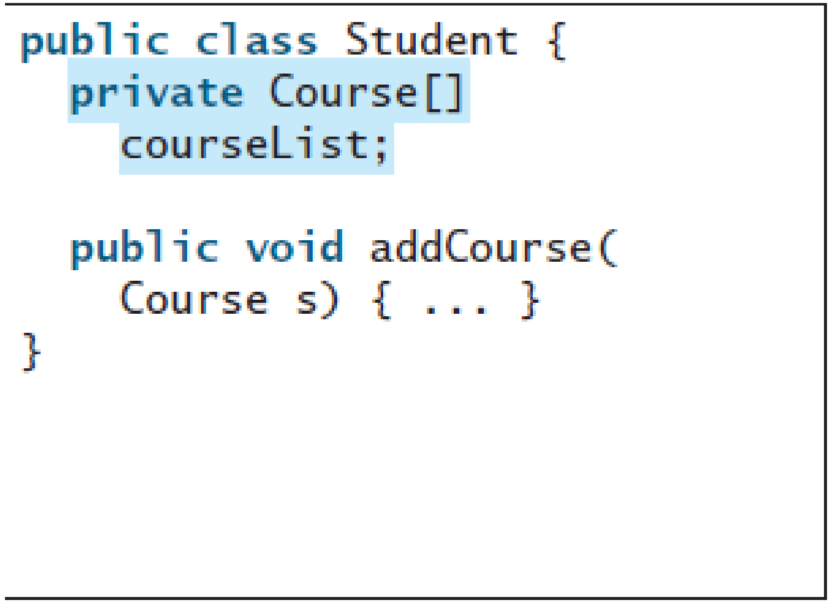
Class Relationships

- *To design classes, you need to explore the relationships among classes*
- *Common relationships among classes are* association, aggregation, composition, *and* inheritance
- *Association* is a general binary relationship that describes an activity between two classes

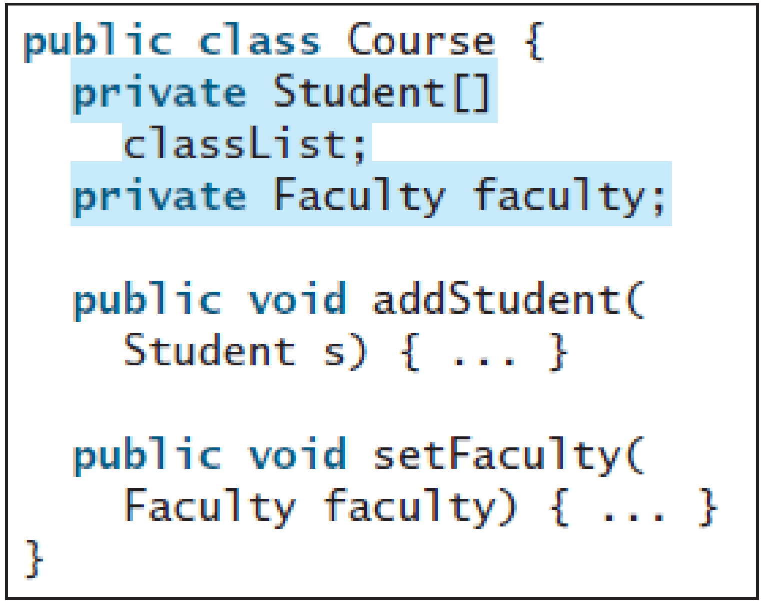


Association

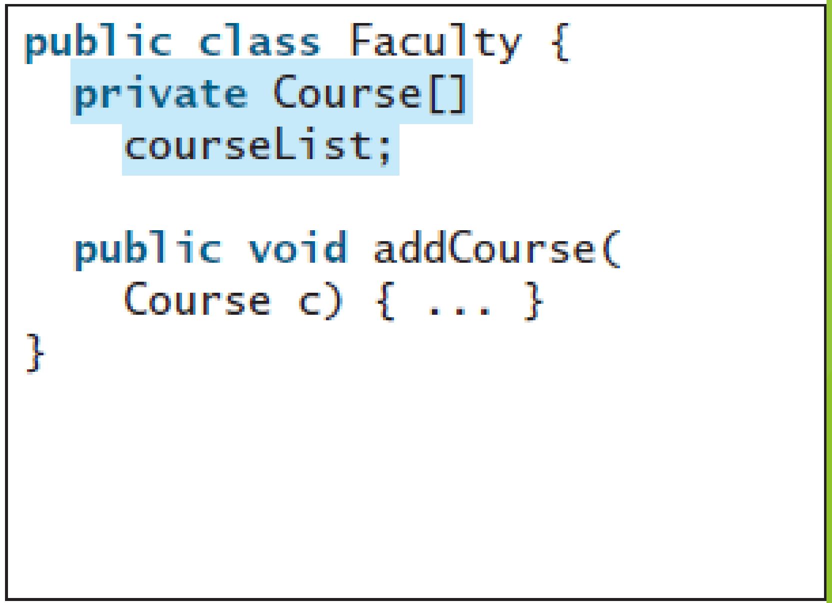
- An association is illustrated by a solid line between two classes
- label is optional that describes the relationship
- Each relationship may have an optional small black triangle that indicates the direction of the relationship
- Each class involved in an association may specify a *multiplicity*
  - to specify how many of the class's objects are involved in the relationship
- multiplicity could be a number or an interval
  - ***** means an unlimited number of objects
  - interval **m..n** indicates that the number of objects is between **m** and **n**
- you can implement associations by using data fields and methods



```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```



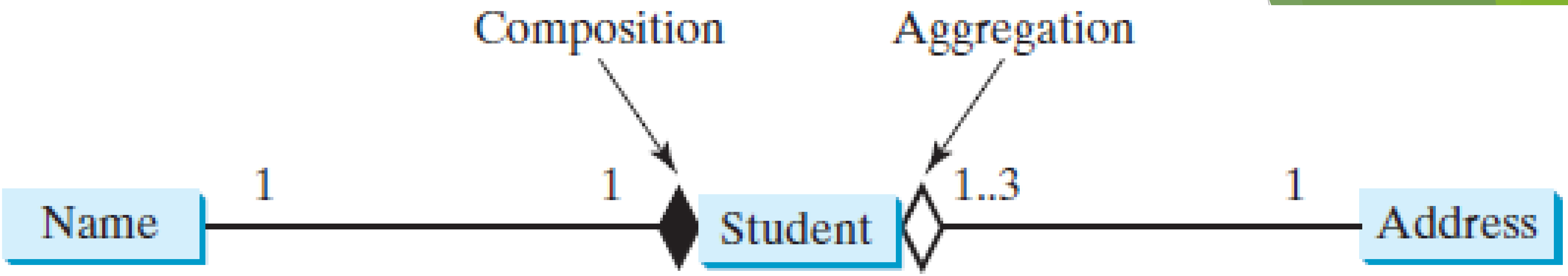
```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

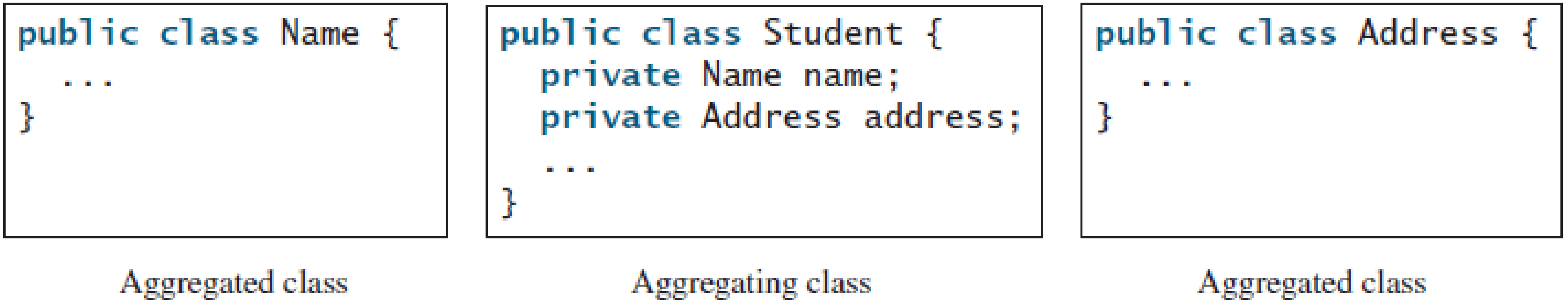


```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

Aggregation and Composition

- *Aggregation* is a special form of association that represents an ownership relationship between two objects
- Aggregation models *has-a* relationships
- owner object is called an *aggregating object*, and its class is called an *aggregating class*
- subject object is called an *aggregated object*, and its class is called an *aggregated class*
- object can be owned by several other aggregating objects
- If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a *composition*.
- An aggregation relationship is usually represented as a data field in the aggregating class





```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

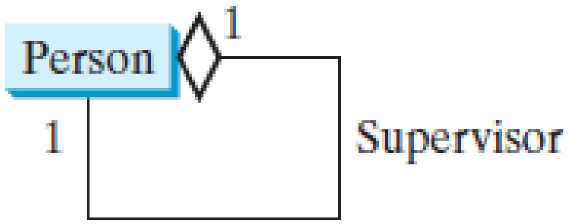
Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

Aggregation and Composition (Contd…)

- Aggregation may exist between objects of the same class



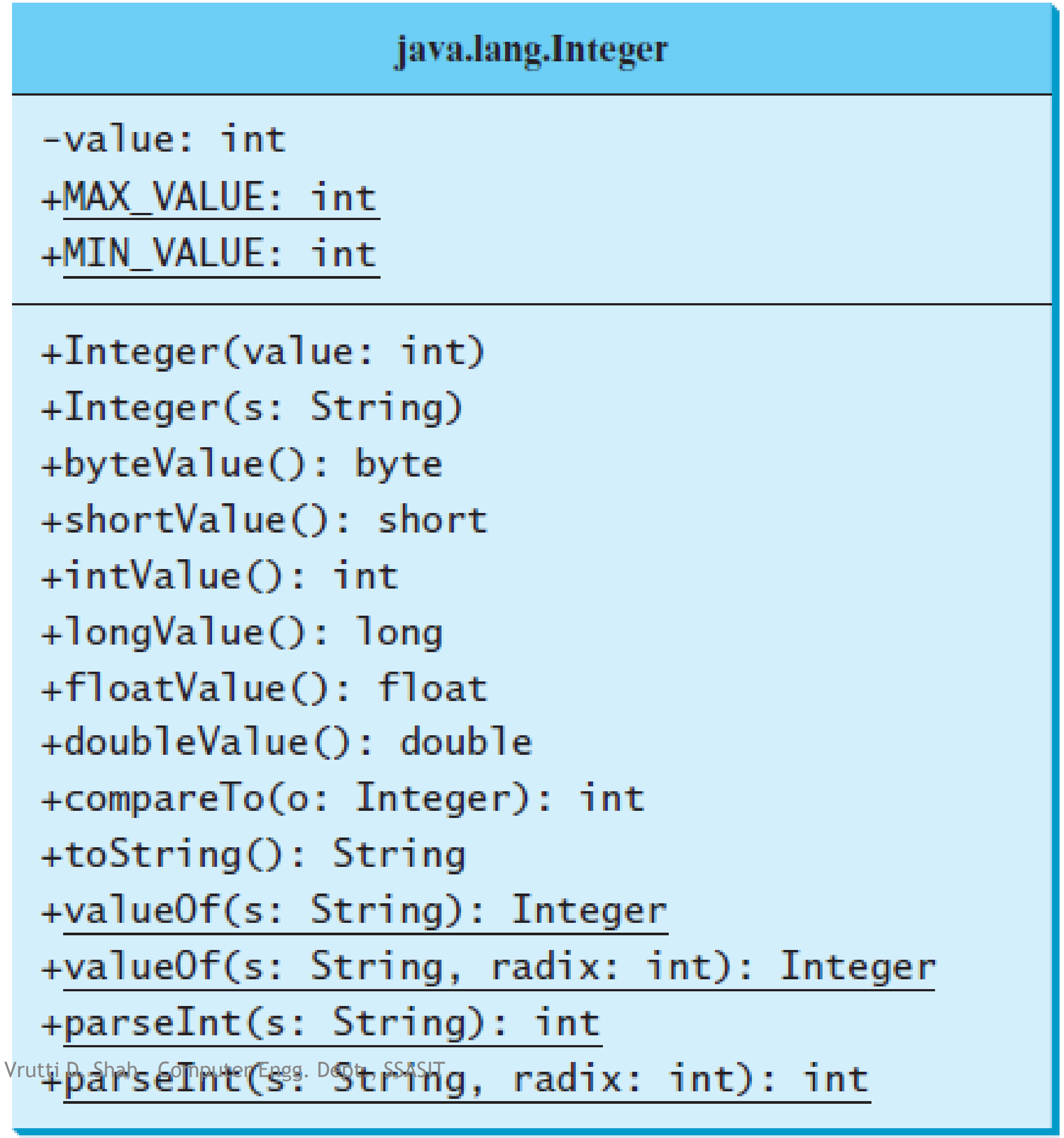
```
public class Person {
// The type for the data is the class itself
private Person supervisor;
...
}
```

Processing Primitive Data Type Values as Objects

- *A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API*
- Java offers a convenient way to incorporate, or wrap, a primitive data type into an object
- By using a wrapper class, you can process primitive data type values as objects
- Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types
- **Boolean** class wraps a Boolean value **true** or **false**

Processing Primitive Data Type Values as Objects (Contd…)

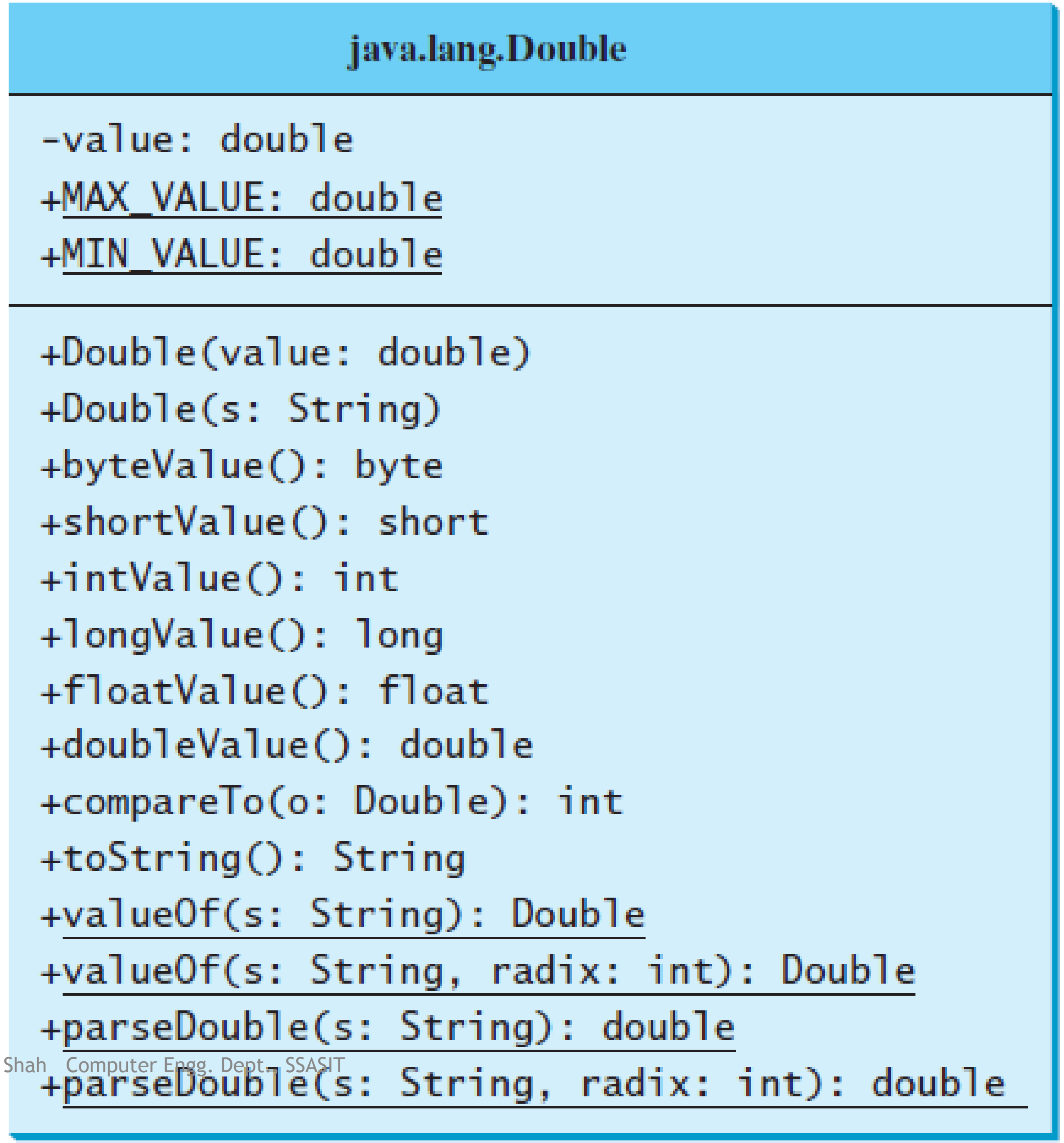
- Numeric wrapper classes are very similar to each other
- Each contains the methods **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**, **shortValue()**, and **byteValue()**
- These methods “convert” objects into primitive type values



java.lang.Integer

```
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
```

```
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int
```



java.lang.Double

```
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
```

```
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double
```

Processing Primitive Data Type Values as Objects (Contd…)

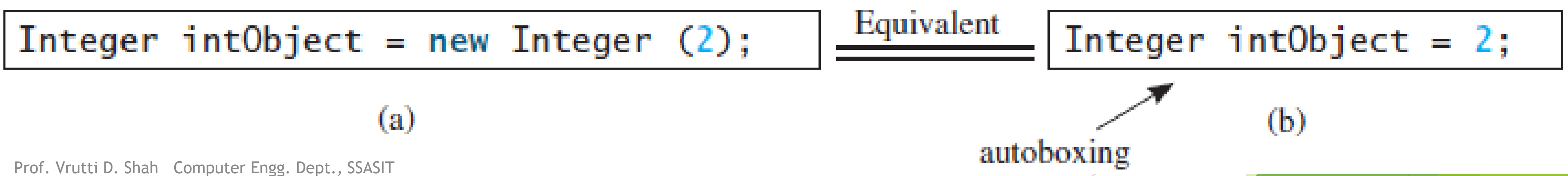
- construct a wrapper object either from a primitive data type value or from a string representing the numeric value
  - **new Double(5.0)**, **new Double("5.0")**,
  - **new Integer(5)**, and **new Integer("5")**
- wrapper classes do not have no-arg constructors
- instances of all wrapper classes are immutable
- numeric wrapper classes contain the **compareTo** method for comparing two numbers
  - **new Double(12.4).compareTo(new Double(12.3))**
  - returns 1, 0, or -1, if this number is greater than, equal to, or less than the other number
- numeric wrapper classes have a static method, **valueOf (String s)** method creates a new object initialized to the value represented by the specified string
  - Double doubleObject = Double.valueOf("**12.4**");

- Each numeric wrapper class has two overloaded parsing methods
  - to parse a numeric string into an appropriate numeric value based on **10** (decimal) or any specified radix (e.g., **2** for binary, **8** for octal, and **16** for hexadecimal)
- **public static byte** parseByte(String s)
- **public static byte** parseByte(String s, **int** radix)
- **public static short** parseShort(String s)
- **public static short** parseShort(String s, **int** radix)
- **public static int** parseInt(String s)
- **public static int** parseInt(String s, **int** radix)

- **public static long** parseLong(String s)
- **public static long** parseLong(String s, **int** radix)
- **public static float** parseFloat(String s)
- **public static float** parseFloat(String s, **int** radix)
- **public static double** parseDouble(String s)
- **public static double** parseDouble(String s, **int** radix)
  - **Integer.parseInt("11", 2)** returns **3**;
  - **Integer.parseInt("12", 8)** returns **10**;
  - **Integer.parseInt("13", 10)** returns **13**;
  - **Integer.parseInt("1A", 16)** returns **26**;

Automatic Conversion between Primitive Types and Wrapper Class Types

- *A primitive type value can be automatically converted to an object using a wrapper class, and vice versa*
- Converting a primitive value to a wrapper object is called *boxing*, reverse conversion is called *unboxing*
- compiler will automatically box a primitive value that appears in a context requiring an object
- It will unbox an object that appears in a context requiring a primitive value
- *It is called autoboxing* and *autounboxing*



```
Integer intObject = new Integer (2);
```

```
Integer intObject = 2;
```

BigInteger and BigDecimal Classes

- BigInteger *and* BigDecimal *classes can be used to represent integers or decimal numbers of any size and precision*
- If you need to compute with very large integers or high-precision floating-point values, you can use the BigInteger and BigDecimal classes
- Both are *immutable*
- largest integer of the **long** type is Long.MAX_VALUE (i.e., 9223372036854775807)
- instance of BigInteger can represent an integer of any size
- use new BigInteger(String) and new BigDecimal(String) to create an instance of BigInteger and BigDecimal

BigInteger and BigDecimal Classes (Contd…)

- use **add**, **subtract**, **multiply**, **divide**, and **remainder** methods to perform arithmetic operations
- the **compareTo** method to compare two big numbers
-
  ```
  BigInteger a = new BigInteger("9223372036854775807");
    BigInteger b = new BigInteger("2");
    BigInteger c = a.multiply(b);
    System.out.println(c);
  ```

- There is no limit to the precision of a **BigDecimal** object
- **divide** method may throw an **ArithmeticException** if the result cannot be terminated
- can use the overloaded **divide(BigDecimal d, int scale, int roundingMode)** method to specify a scale and a rounding mode to avoid this exception
-
```
BigDecimal a = new BigDecimal(1.0);
 BigDecimal b = new BigDecimal(3);
 BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
 System.out.println(c);
```

The String Class

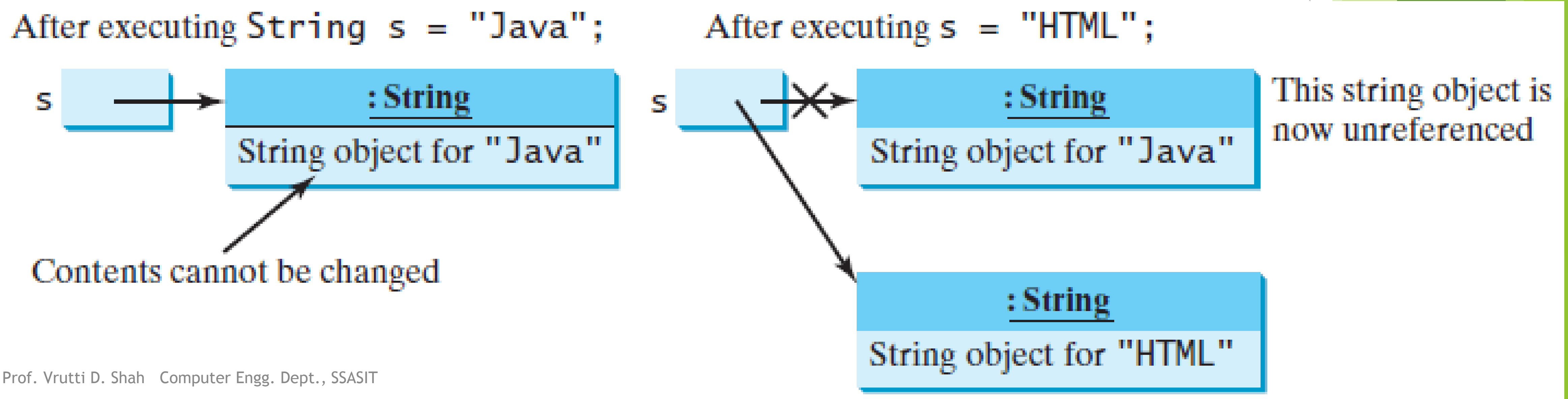
- String *object is immutable: Its content cannot be changed once the string is created.*
- String class has 13 constructors and more than 40 methods for manipulating strings
- can create a string object from a string literal or from an array of characters
  - String newString = new String(stringLiteral);
  - String message = new String("Welcome to Java");
  - String message = "Welcome to Java";
- **char**[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};

  String message = **new** String(charArray);

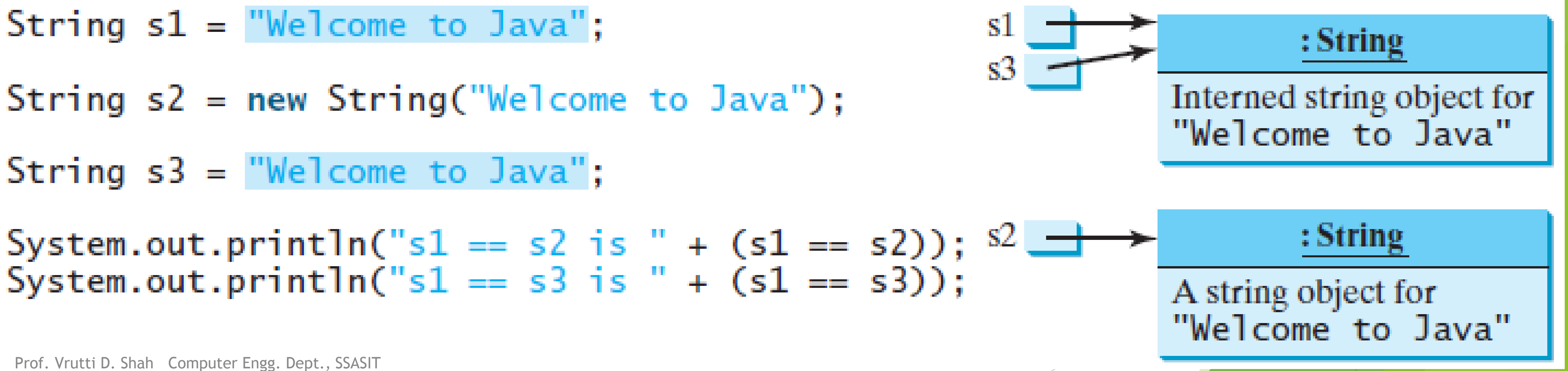
Immutable Strings and Interned Strings

- String object is immutable
- String s = "Java";

  s = "HTML";


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

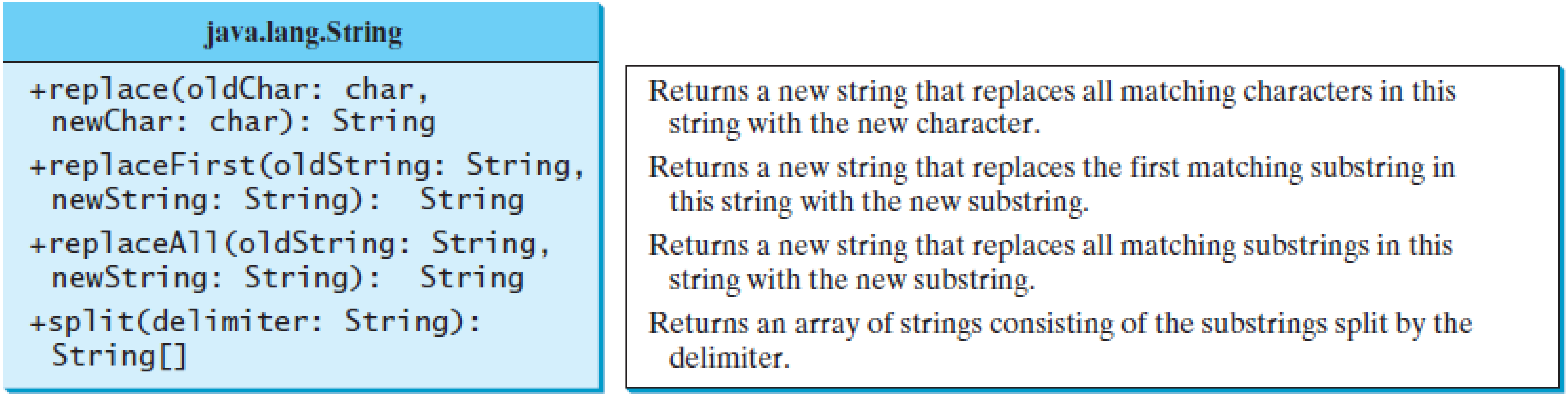
- Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory.
- Such an instance is called an *interned string*



```
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"

Replacing and Splitting Strings



java.lang.String

`+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.

- "Welcome".replace('e', 'A') returns a new string, WAlcomA.
- "Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.
- "Welcome".replace("e", "AB") returns a new string, WABlcomAB.
- "Welcome".replace("el", "AB") returns a new string, WABcome.
- split method can be used to extract tokens from a string with the specified delimiters
- String[] tokens = "Java#HTML#Perl".split("#");

```
 for (int i = 0; i < tokens.length; i++)
 System.out.print(tokens[i] + " ");
```

Matching, Replacing and Splitting by Patterns

- *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings
- can match, replace, or split a string by specifying a pattern
- **matches** method is very similar to the **equals** method
- **"Java"**.matches("**Java**");

  "Java".equals("**Java**");
- **matches** method can match fixed stringand a set of strings that follow a pattern
- **"Java is fun"**.matches**("Java.*")**
- **"Java is cool"**.matches**("Java.*")**
- **"Java is powerful"**.matches**("Java.*")**
- **"440-02-4534"**.matches**("\\d{3}-\\d{2}-\\d{4}")**
  - **\\d** represents a single digit, and **\\d{3}** represents three digits

- replaceAll, replaceFirst, and split methods can be used with a regular expression
- String s = "a+b$#c".replaceAll("[$+#]", "NNN");

  System.out.println(s);

- String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

Conversion between Strings and Arrays

- Strings are not arrays, but a string can be converted into an array, and vice versa
- To convert a string into an array of characters, use the **toCharArray** method
  - **char**[] chars = "Java".toCharArray();
- **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** method to copy a substring of the string
  - **char**[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
  - "CS3720".getChars(2, 6, dst, 4);
  - dst becomes {'J', 'A', 'V', 'A', '3', '7', '2', '0'}.

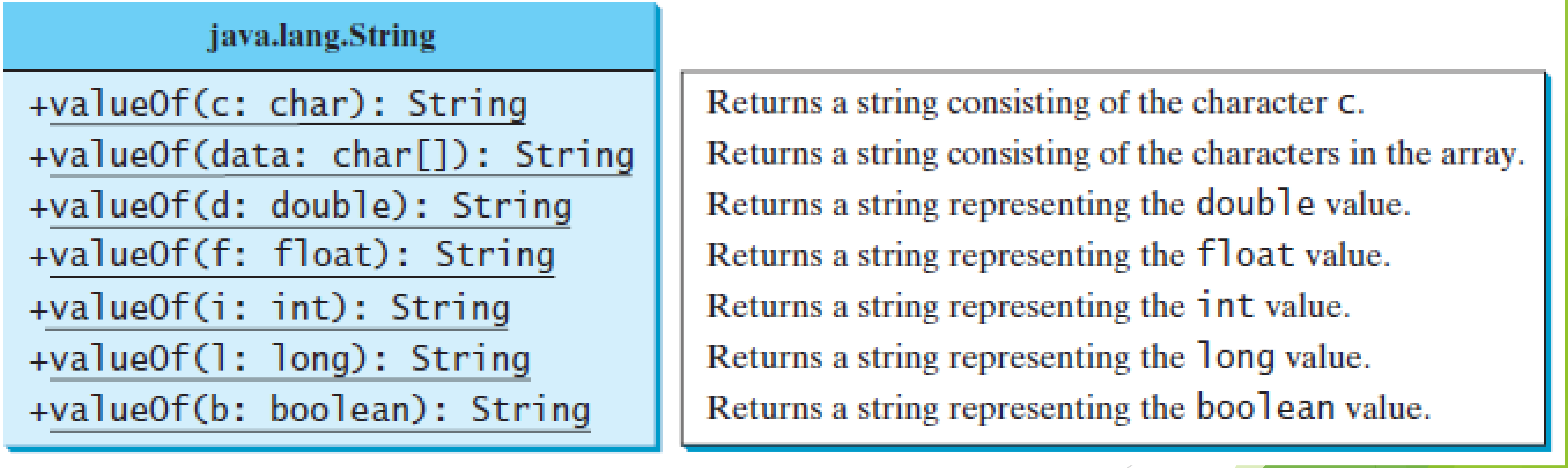
Conversion between Strings and Arrays (Contd…)

- To convert an array of characters into a string, use the **String(char[])** constructor or the **valueOf(char[])** method
- constructs a string from an array using the **String** constructor
  - String str = new String(new char[]{'J', 'a', 'v', 'a'});
- constructs a string from an array using the **valueOf** method
  - String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});

Converting Characters and Numeric Values to Strings

- you can use **Double.parseDouble(str)** or **Integer.parseInt(str)** to convert a string to a **double** value or an **int** value
- you can convert a character or a number into a string by using the string concatenating operator
- Another way of converting a number into a string is to use the overloaded static **valueOf** method

Converting Characters and Numeric Values to Strings (Contd…)



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

- **String** class contains the static **format** method to return a formatted string
  - String.format(format, item1, item2, ..., item*k*)
- method is similar to the **printf** method, except that the **format** method returns a formatted string, whereas the **printf** method displays a formatted string
- String s = String.format(**"%7.2f%6d%-4s", 45.556, 14, "AB"**);

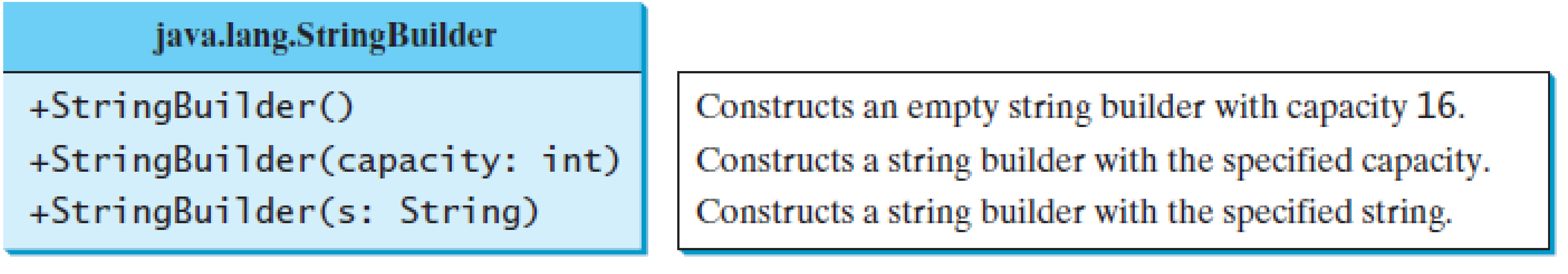
```
System.out.println(s);
```

StringBuilder and StringBuffer Classes

- StringBuilder *and* StringBuffer *classes are similar to the* **String** *class except that the* **String** *class is immutable*
- StringBuilder and StringBuffer classes can be used wherever a string is used
- StringBuilder and StringBuffer are more flexible than String
- You can add, insert, or append new contents into StringBuilder and StringBuffer objects
- value of a String object is fixed once the string is created
- StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are *synchronized*
  - which means that only one task is allowed to execute the methods

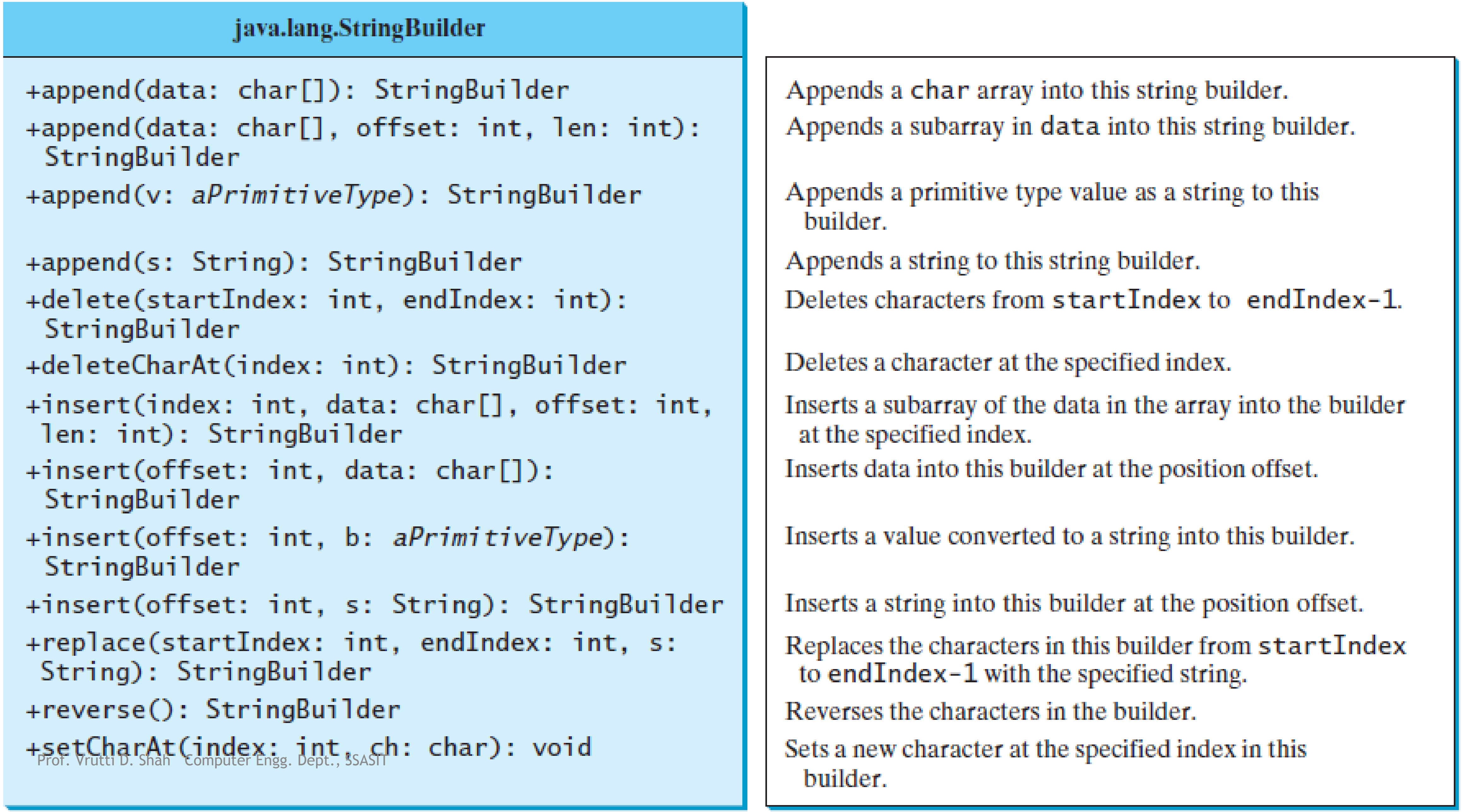
StringBuilder and StringBuffer Classes (Contd…)

- Use StringBuffer if the class might be accessed by multiple tasks concurrently
- Using StringBuilder is more efficient if it is accessed by just a single task
- constructors and methods in StringBuffer and StringBuilder are almost the same
- StringBuilder class has three constructors and more than 30 methods
- You can create an empty string builder or a string builder from a string using the constructors



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.

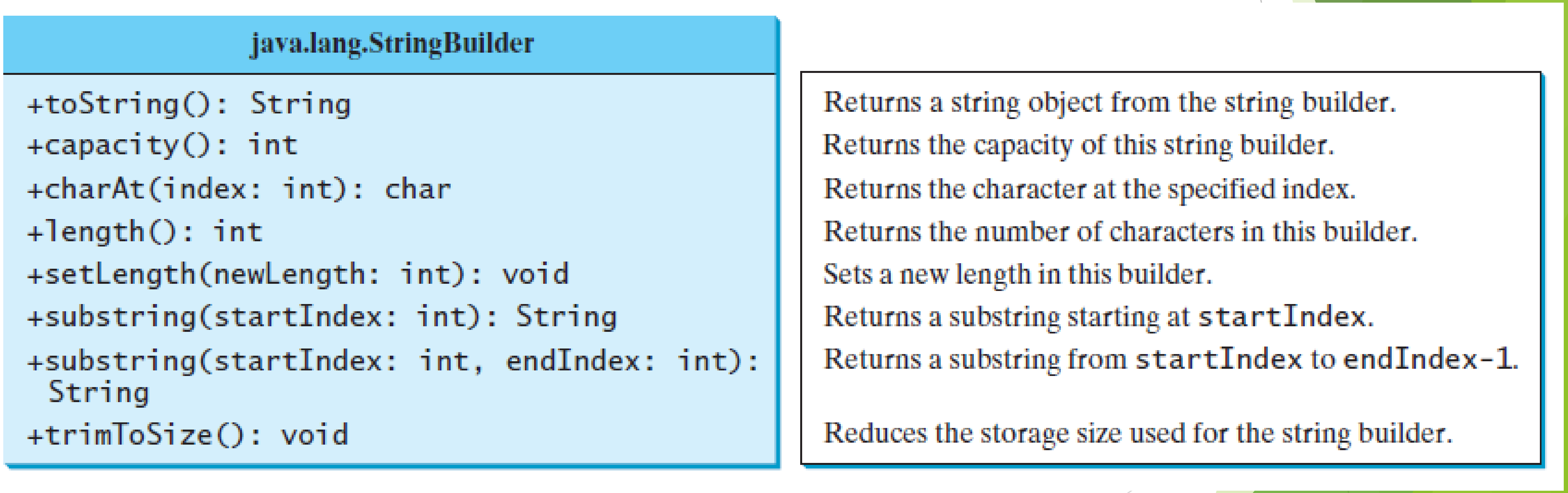


java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a 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.

StringBuilder and StringBuffer Classes (Contd…)

- StringBuilder class provides several overloaded methods to append boolean, char, char[], double, float, int, long, and String into a string builder
- StringBuilder class also contains overloaded methods to insert boolean, char, char array, double, float, int, long, and String into a string builder
  - stringBuilder.insert(**11**, **"HTML and "**);
- You can also delete characters from a string in the builder using the two delete methods
- reverse the string using the reverse method
- replace characters using the replace method
- set a new character in a string using the setCharAt method

toString, capacity, length, setLength, and charAt Methods



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.

toString, capacity, length, setLength, and charAt Methods (Contd…)

- capacity() method returns the current capacity of the string builder
- length() method returns the number of characters actually stored in the stringbuilder
- setLength(newLength) method sets the length of the string builder
  - newLength argument must be greater than or equal to **0**
- charAt(index) method returns the character at a specific index in the stringbuilder
  - index argument must be greater than or equal to **0 and** less than the length of the string builder