CEN 419
Introduction to Java Programming

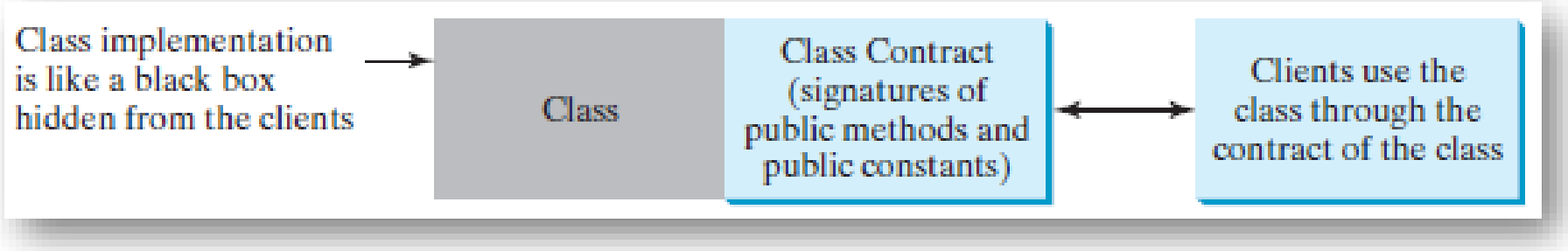
Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

Class Abstraction and Encapsulation

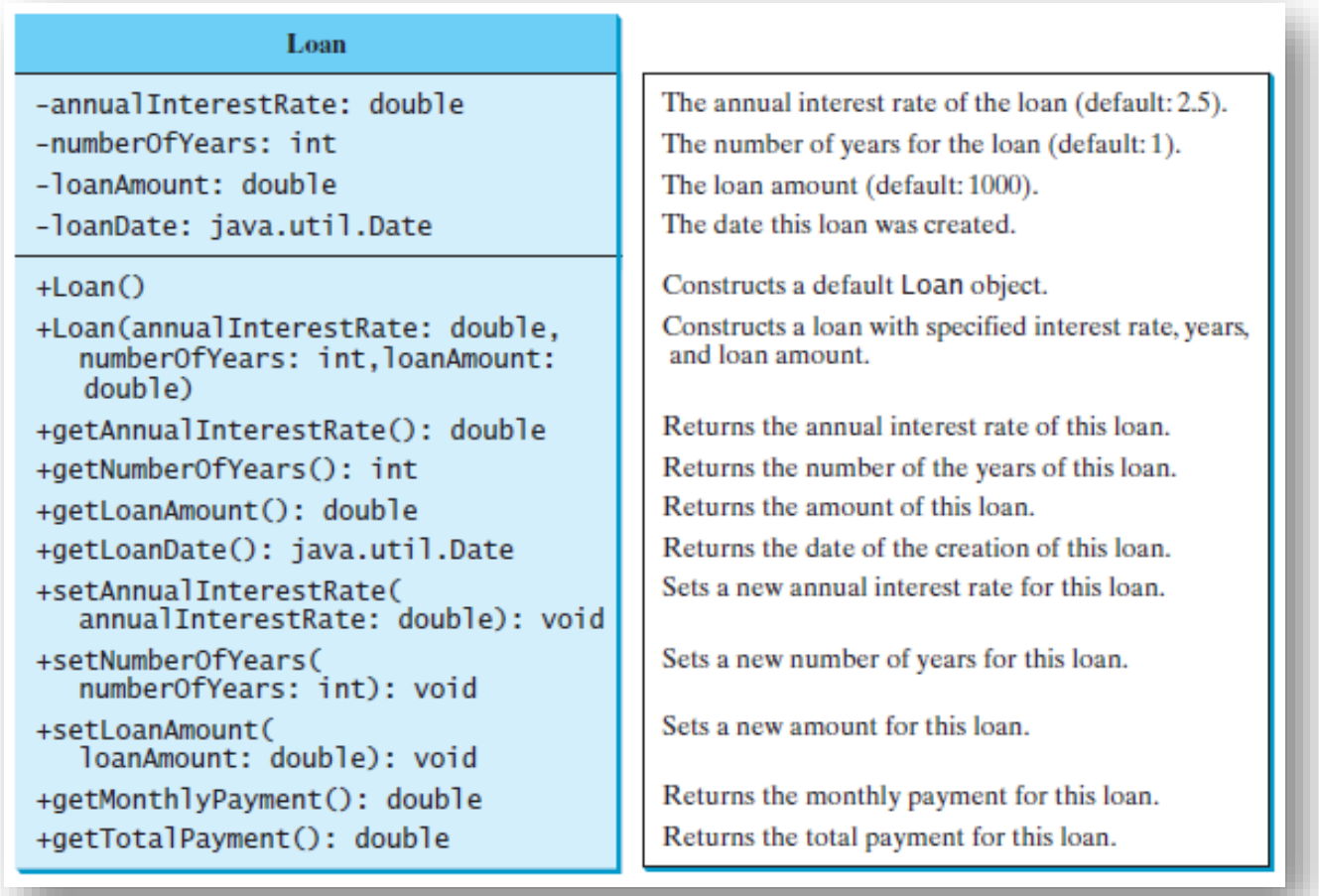
- ***Class abstraction*** means to separate class implementation from the use of the class.
- The creator of the class provides a description of the class and let the user know how the class can be used (***class's contract***). The user of the class does not need to know how the class is implemented.
- The details of implementation are encapsulated and hidden from the user. This is called ***class encapsulation***.



Designing the Loan Class

The **Loan** class models the properties and behaviors of loans.

Remember that a class user can use the class without knowing how the class is implemented.



Loan Class Intro to Java Programming, Y. Daniel Liang - Loan.java (pearsoncmg.com)

Test Class Intro to Java Programming, Y. Daniel Liang - TestLoanClass.java (pearsoncmg.com)

Thinking in Objects

- A program for computing body mass index.

A BMI Program

Intro to Java Programming, Y. Daniel Liang - ComputeAndInterpretBMI.java (pearsoncmg.com)

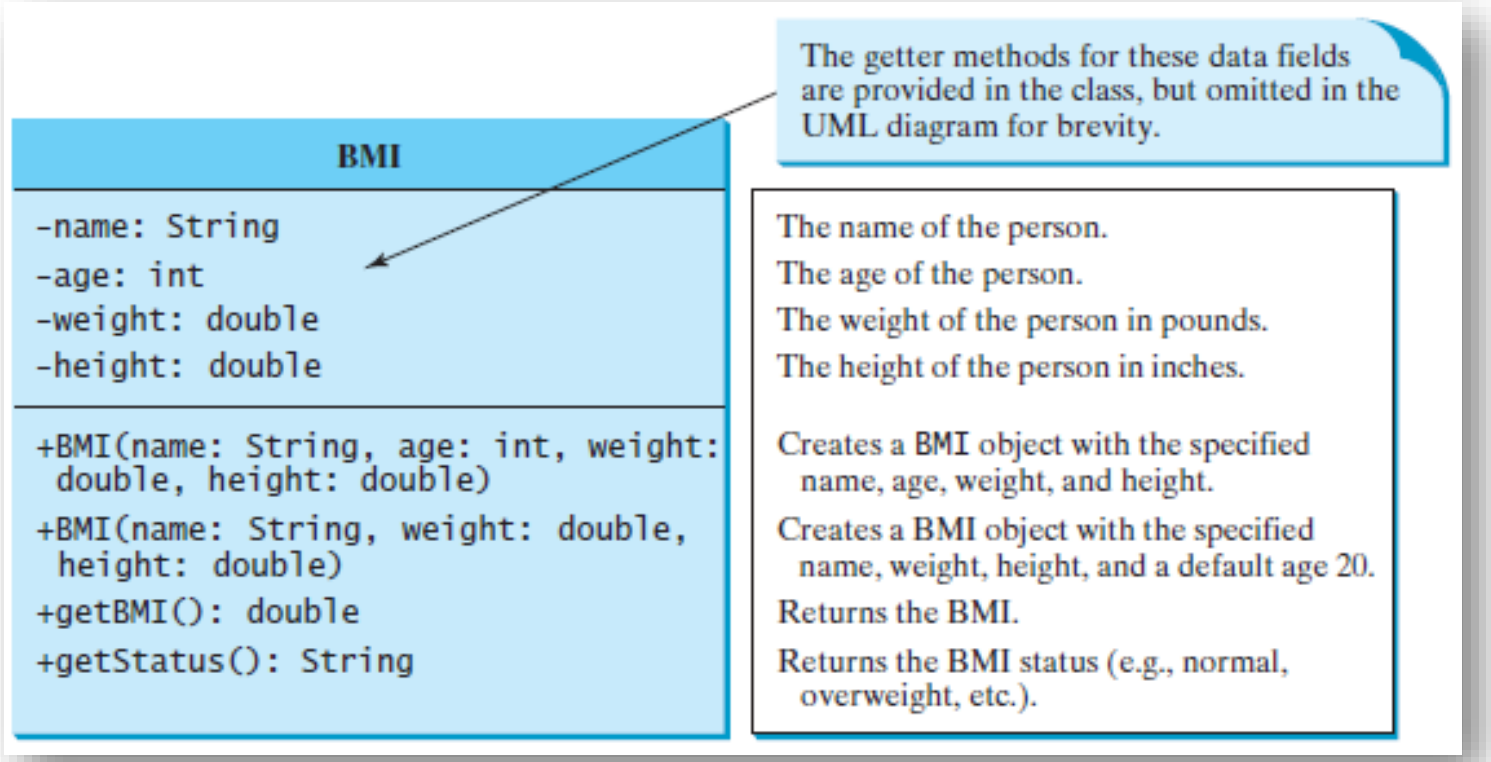
- The code cannot be reused in other programs, because the code is in the **main** method. To make it reusable, define a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

- This method is useful for computing body mass index for a specified weight and height. However, it has limitations.
- Suppose you need to associate the weight and height with a person's name and birth date. You could declare separate variables to store these values, but these values would not be tightly coupled. The ideal way to couple them is to create an object that contains them all.

The BMI Class

Classes provide more flexibility and modularity for building reusable software.



BMI Class Intro to Java Programming, Y. Daniel Liang - BMI.java (pearsoncmg.com)

Test Class Intro to Java Programming, Y. Daniel Liang - UseBMIClass.java (pearsoncmg.com)

Class Relationships

- To design classes, you need to explore the relationships among classes.
- The common relationships among classes are:
  - ☑ Association
  - ☑ Aggregation
  - ☑ Composition
  - ☑ Inheritance

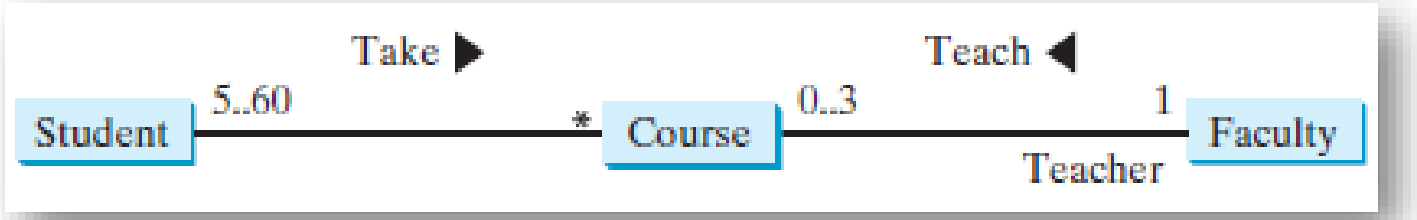
Association

- ***Association*** is a general binary relationship that describes an activity between two classes.
- For example, a student taking a course is an association between the **Student** class and the **Course** class, and a faculty member teaching a course is an association between the **Faculty** class and the **Course** class.

UML Graphical Notation of Association

- An association is illustrated by a *solid line* between two classes with an optional label (e.g. Take, Teach) that describes the relationship.
- Each class involved in an association may specify a *multiplicity*, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship (i.e. * , m..n)

0	No instances (rare)
0..1	No instances, or one instance
1	Exactly one instance
1..1	Exactly one instance
0..*	Zero or more instances
*	Zero or more instances
1..*	One or more instances



Class Representation of Association

- The association relations are implemented using data fields and methods in classes.

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

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

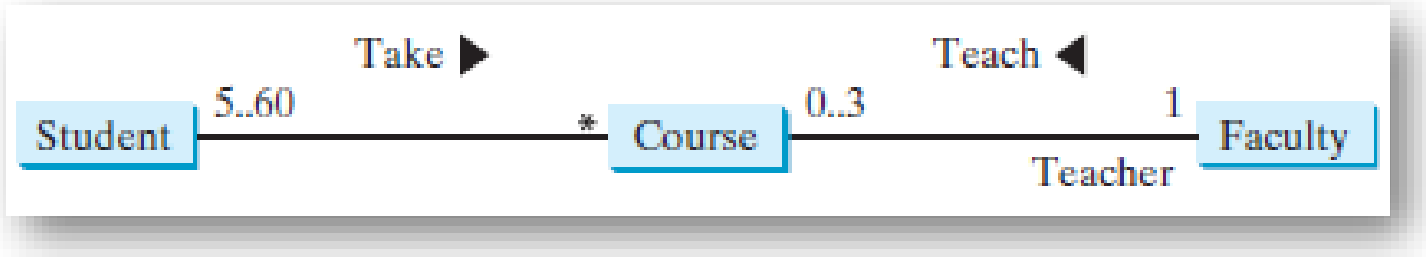
```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  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*** models has-a relationships and represents an ownership relationship between two objects.
- The owner object is called an aggregating object and its class an aggregating class. The subject object is called an aggregated object and its class an aggregated class.
- ***Composition*** is actually a special case of the aggregation relationship.

Aggregation and Composition

- Composition and Aggregation are types of associations. They are very closely related and in terms of programming there does not appear much difference.
  - Aggregation: the object exists outside the other, is created outside, so it is passed as an argument to the constructor. Ex: People – car. The car is create in a different context and than becomes a person property.
  - Composition: the object only exists, or only makes sense inside the other, as a part of the other. Ex: People – heart. You don't create a heart and than pass it to a person.

Code Example

- WebServer is **aggregated** of a HttpListener and a RequestProcessor

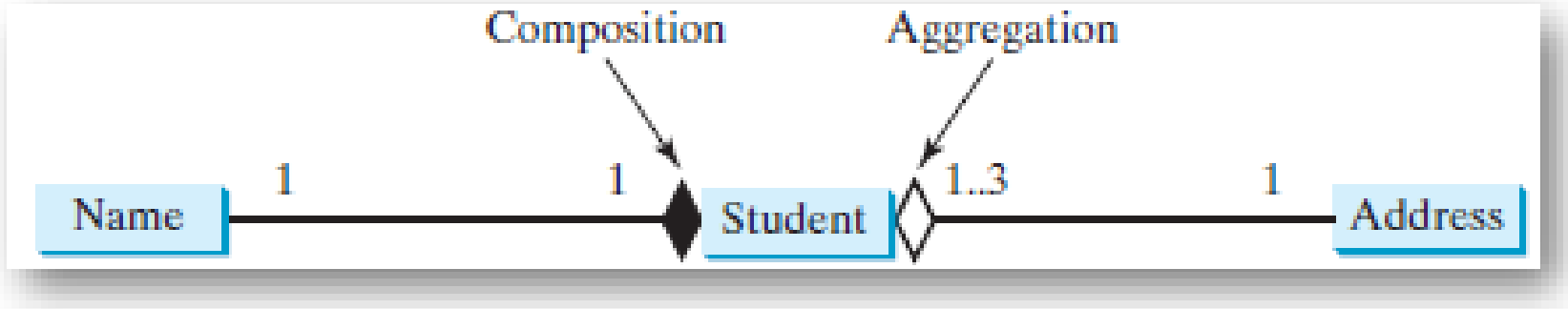
```
public class WebServer {
 private HttpListener listener;
 private RequestProcessor processor;
 public WebServer(HttpListener listener, RequestProcessor processor) {
 this.listener = listener;
 this.processor = processor;
 }
}
```

- WebServer is a **composition** of HttpListener and RequestProcessor and controls their lifecycle

```
public class WebServer {
 private HttpListener listener;
 private RequestProcessor processor;
 public WebServer() {
 this.listener = new HttpListener(80);
 this.processor = new RequestProcessor("/www/root");
 }
}
```

Class Representation

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

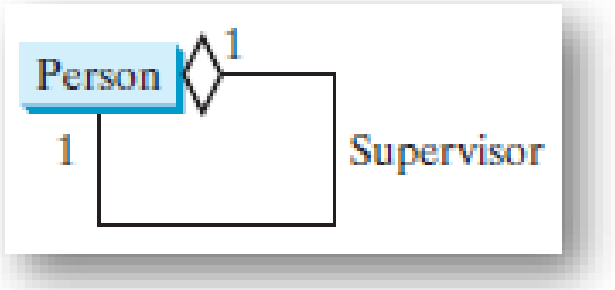
```
public class Address {
  ...
}
```

Aggregated class

Aggregation Between Same Class

Aggregation may exist between objects of the same class.

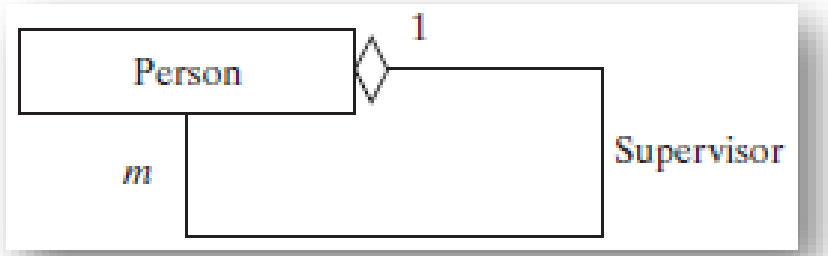
For example, a person may have a supervisor.




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

Aggregation Between Same Class

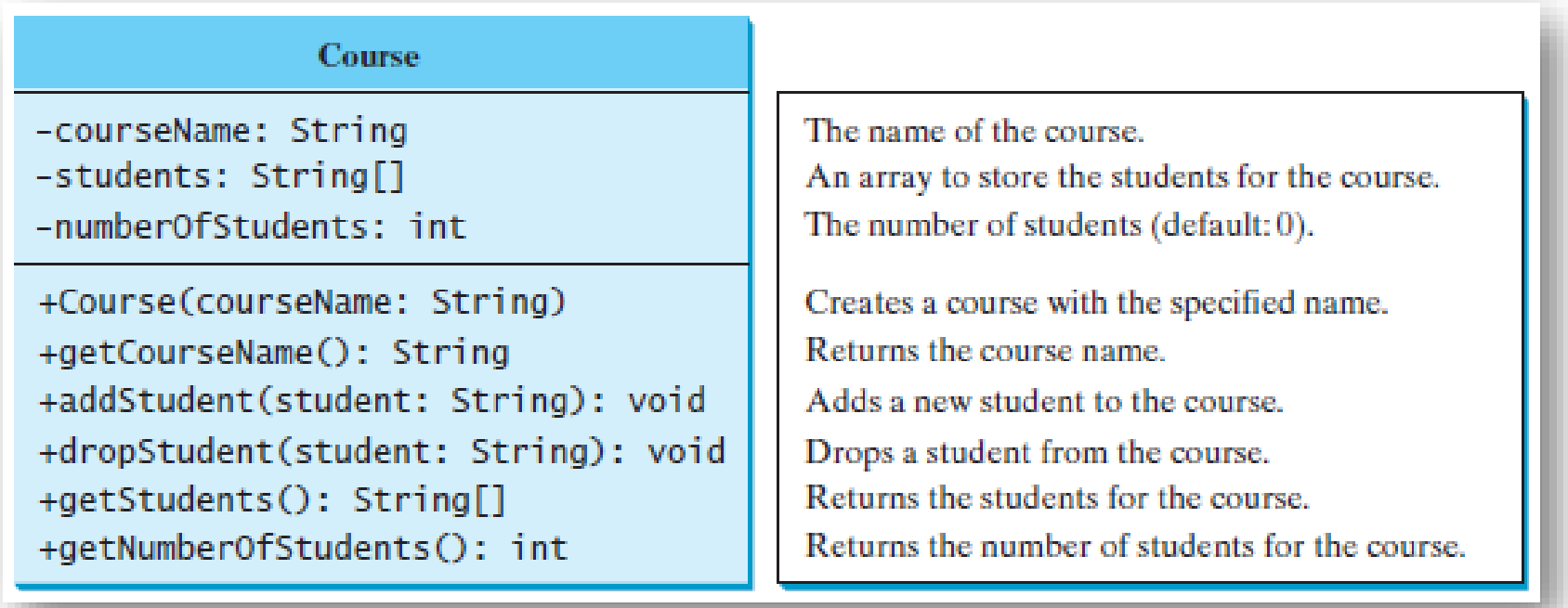
What happens if a person has several supervisors?




```
public class Person {
  ...
  private Person[] supervisors;
}
```

- You may use an array to store supervisors

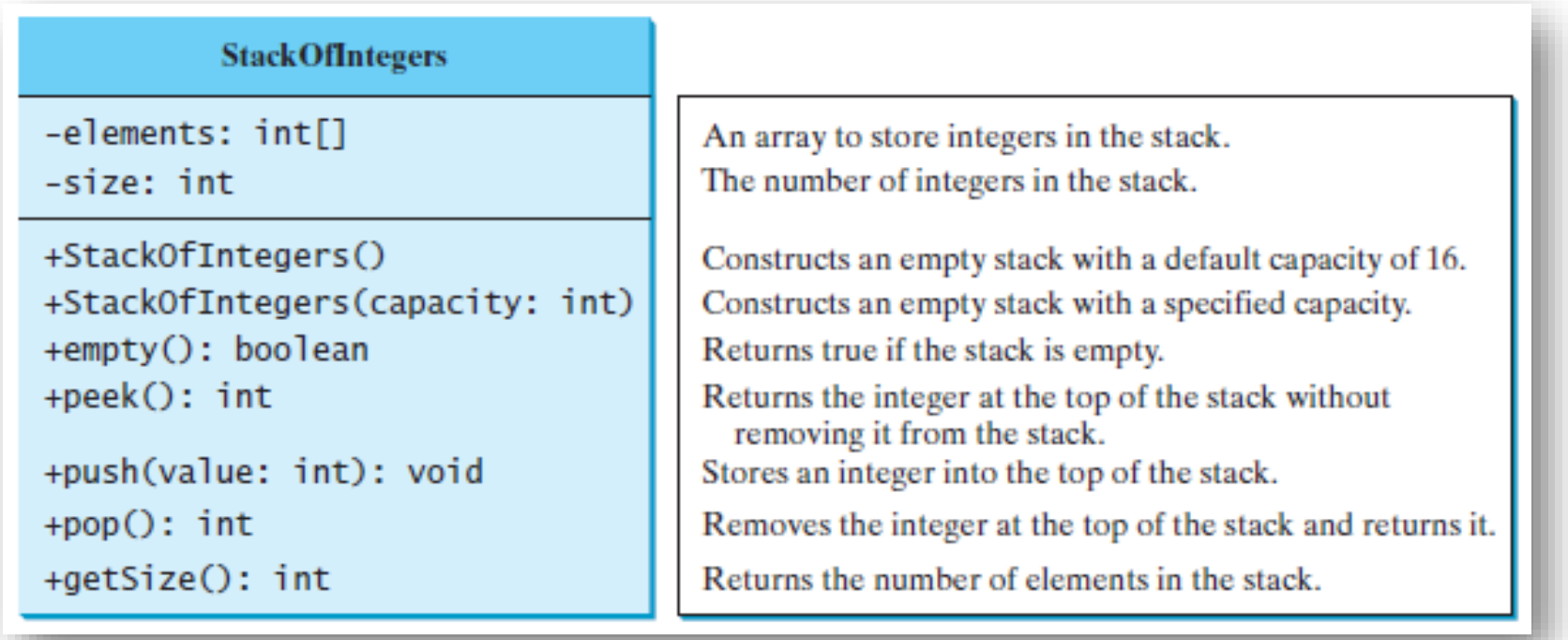
Example: The Course Class



Course Class Intro to Java Programming, Y. Daniel Liang - Course.java (pearsoncmg.com)

Test Class Intro to Java Programming, Y. Daniel Liang - TestCourse.java (pearsoncmg.com)

Example: The `StackOfIntegers` Class

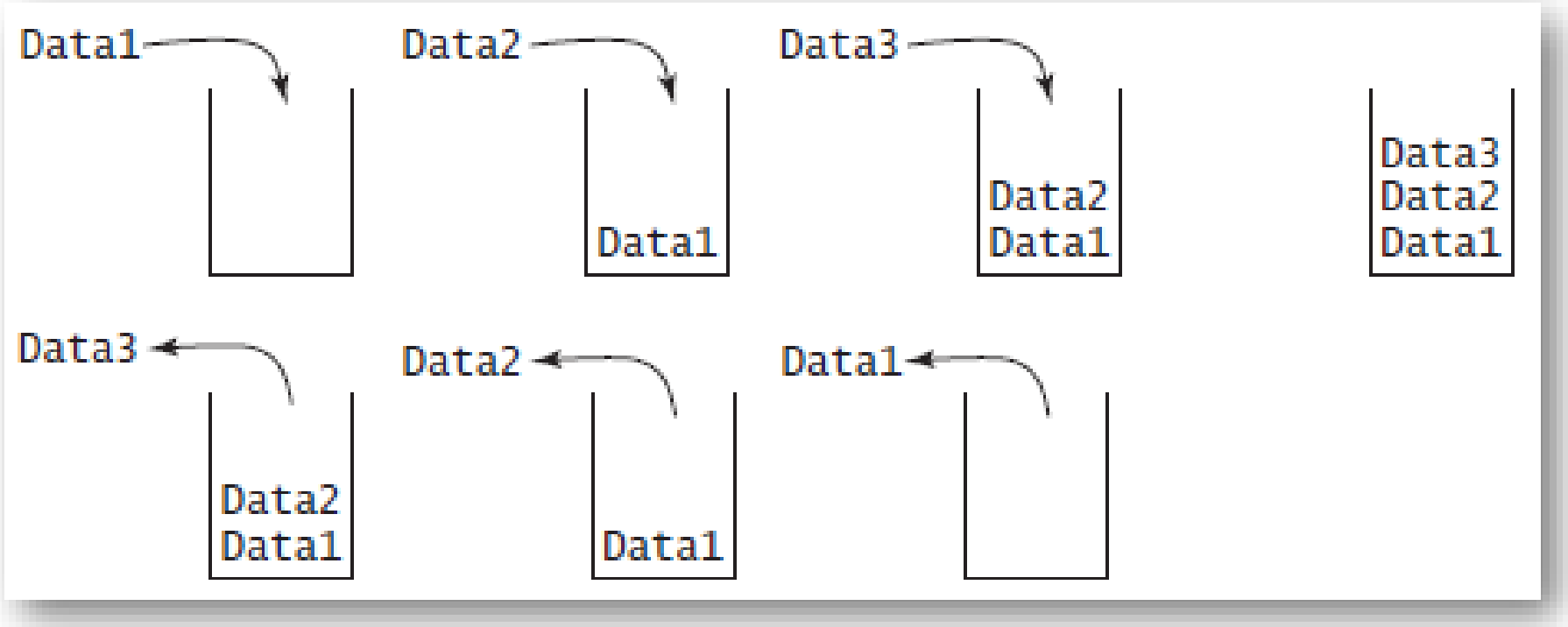


Test Class

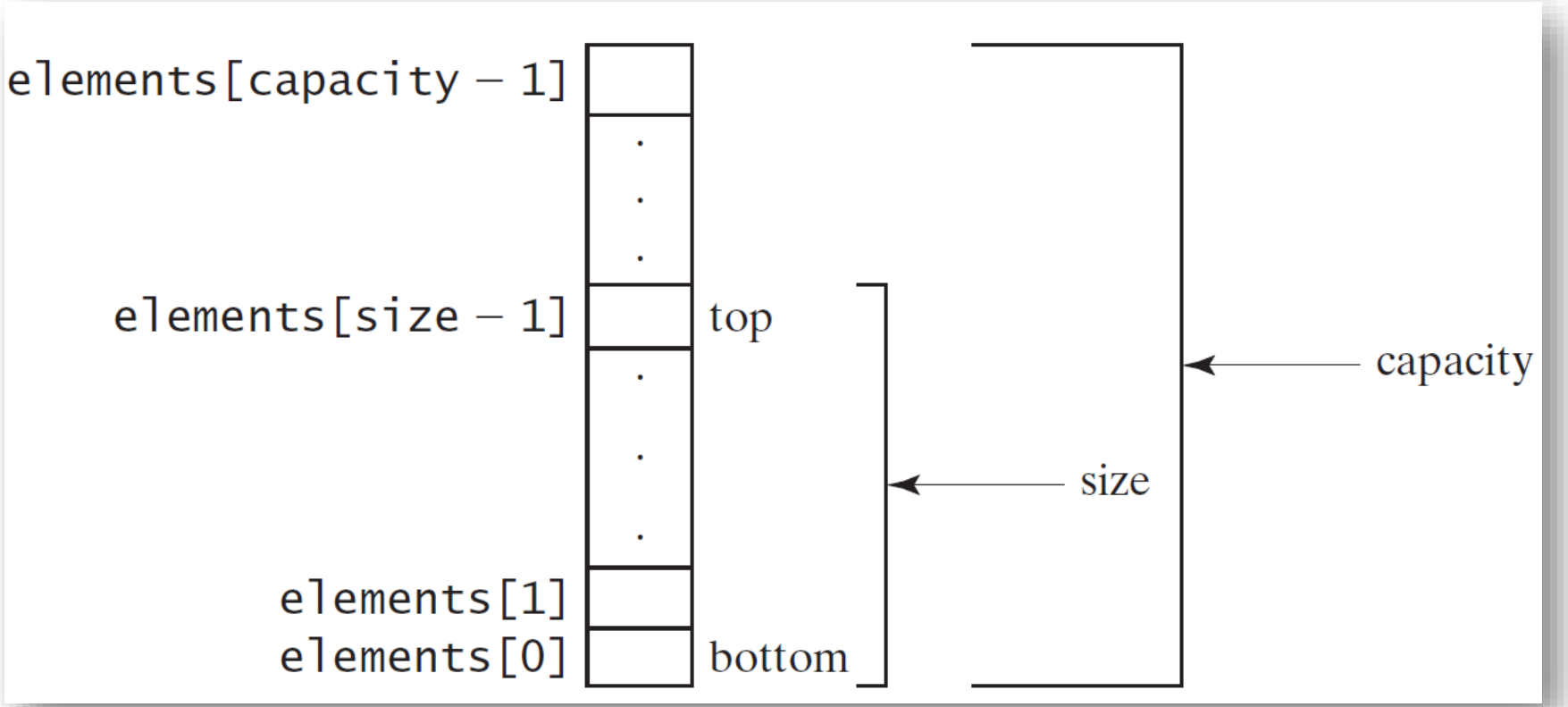
Intro to Java Programming, Y. Daniel Liang - TestStackOfIntegers.java (pearsoncmg.com)

Designing The `StackOfIntegers` Class

Recall that a *stack* is a data structure that holds data in a last-in, first-out fashion,



Implementing The `StackOfIntegers` Class



StackOfIntegers Class

Intro to Java Programming, Y. Daniel Liang - StackOfIntegers.java (pearsoncmg.com)

Wrapper Classes

A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API. The below wrapper classes are in the java.lang package.

- Boolean
- Byte
- Float
- Character
- Integer
- Double
- Short
- Long

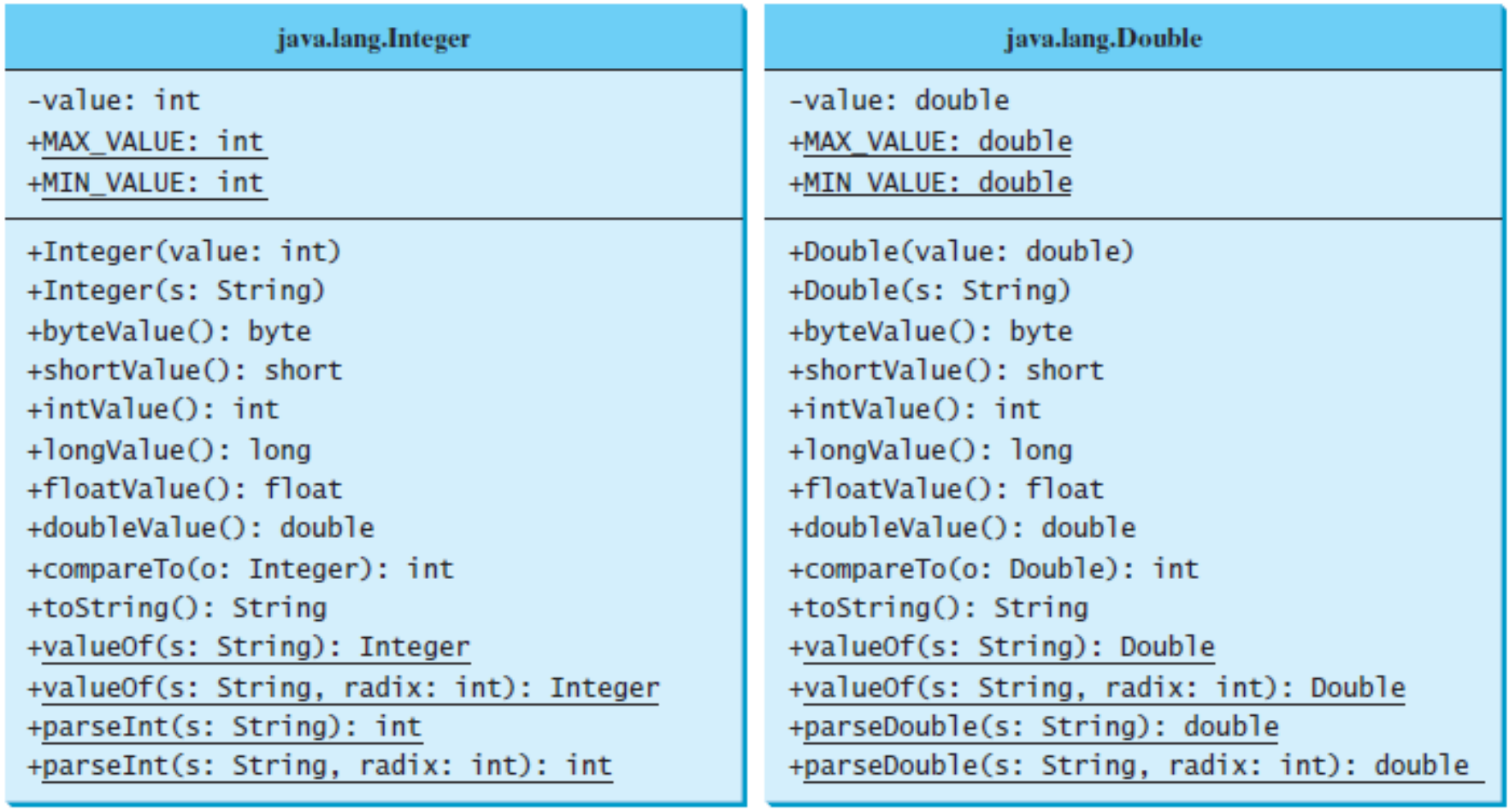
NOTE:

(1) Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are Integer and Character.

(2) The wrapper classes do not have no-arg constructors.

(3) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

The Integer and Double Classes



The `Integer` Class and the `Double` Class

- Constructors
- Class Constants:
  - ✓ `MAX_VALUE`
  - ✓ `MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class **Constructors**

You can construct a wrapper object either *from a primitive data type value* or *from a string representing the numeric value*. The constructors for Integer and Double are:

Constructor	Usage
public Integer(int value)	new Integer(5)
public Integer(String s)	new Integer("5")
public Double(double value)	new Double(5.0)
public Double(String s)	new Double("5.0")

Numeric Wrapper Class **Constants**

- Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE.
- MAX_VALUE represents the maximum value of the corresponding primitive data type.
- For Byte, Short, Integer, and Long MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values.

```
Integer.MAX_VALUE -> 2,147,483,647
Float.MIN_VALUE   -> 1.4E–45
Double.MAX_VALUE  -> 1.79769313486231570e + 308d
```

Numeric Wrapper Class **Conversion Methods**

- Each numeric wrapper class contains the methods: ***doubleValue, floatValue, intValue, longValue,*** and ***shortValue***.

- These methods **"convert" objects** into **primitive type values**.

```
new Double(12.4).intValue() returns 12;
new Integer(12).doubleValue() returns 12.0;
```

Numeric Wrapper Class **compareTo Method**

- Recall that the **String** class contains the **compareTo** method for comparing two strings.
- The numeric wrapper classes contain the **compareTo** method for comparing two numbers and returns **1**, **0**, or **-1**, if this number is greater than, equal to, or less than the other number.

```
new Double(12.4).compareTo(new Double(12.3)) returns 1;
new Double(12.3).compareTo(new Double(12.3)) returns 0;
new Double(12.3).compareTo(new Double(12.51)) returns -1;
```

Numeric Wrapper Class **The Static valueOf Methods**

- The numeric wrapper classes have a useful class method, `valueOf(String s)`.
- This method creates a new object initialized to the value represented by the specified string.
- For example:

```
Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

- You have used the `parseInt` method in the Integer class to **parse a numeric string** into an **int value** and the `parseDouble` method in the Double class to **parse a numeric string** into a **double value**.
- Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

The Methods for Parsing Strings into Numbers

- For example these two methods are in the Integer class:

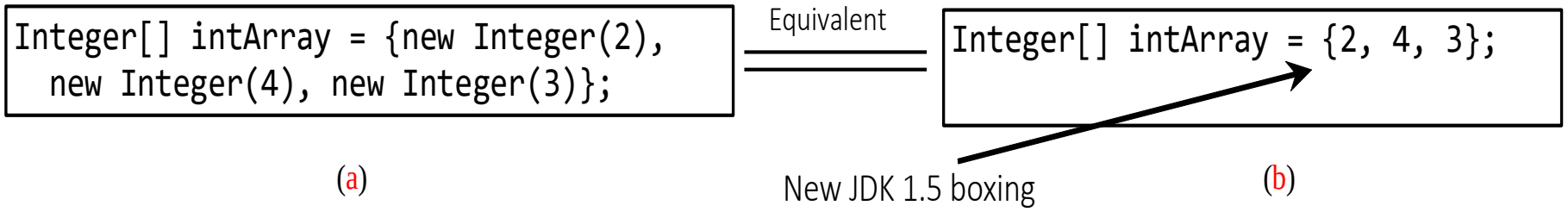
```
public static int parseInt(String s)
public static int parseInt(String s, int radix)
```

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

- `Integer.parseInt("12", 2)` would raise a runtime exception because **12** is not a binary number.

Automatic Conversion Between Primitive Types and Wrapper Class Types

- JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):




```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

BigInteger and BigDecimal

- If you need to compute with very large integers or high precision floating-point values, you can use the **BigInteger** and **BigDecimal** classes in the java.math package.

```
BigInteger a = new BigInteger("92233720368547758 07");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 92233720368547758 07 * 2
System.out.println(c);
```

The `String` Class

- A **String** object is immutable: Its content cannot be changed once the string is created.
- Before we have used :
  - **`charAt(index)`** method to obtain a character at the specified index from a string,
  - The **`length()`** method to return the size of a string,
  - The **`substring`** method to return a substring in a string,
  - The **`indexOf`** and **`lastIndexOf`** methods to return the first or last index of a matching character or a substring.

Constructing Strings

- You can create a string object from a string literal or from an array of characters.

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

- The argument **`stringLiteral`** is a sequence of characters enclosed inside double quotes:

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

- String literals are treated as **String** objects. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

Constructing Strings

- 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);
```

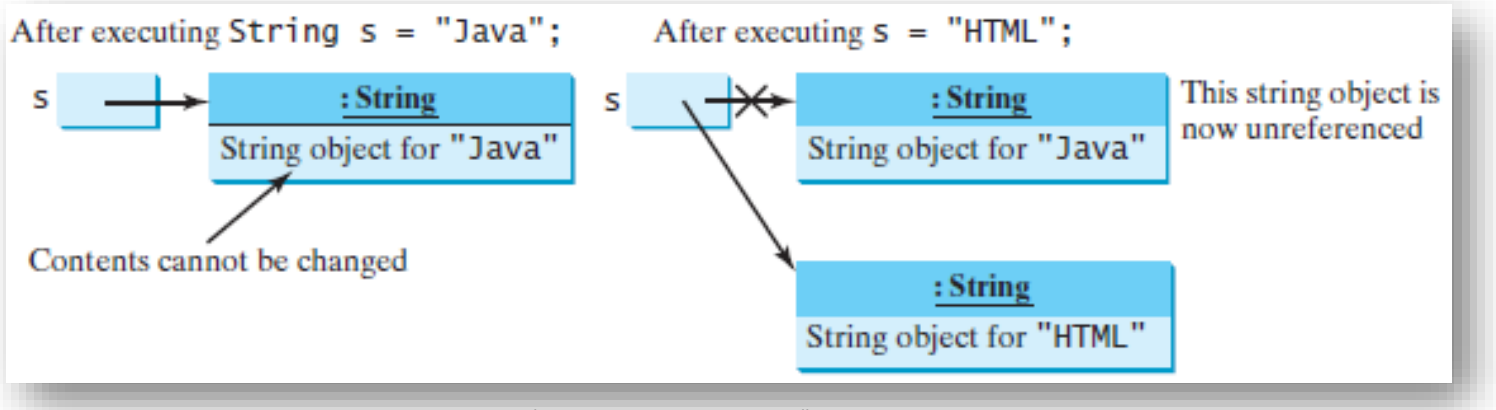
Strings Are Immutable

- A String object is immutable; its contents cannot be changed.

Does the following code change the contents of the string?

```
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:



Interned Strings

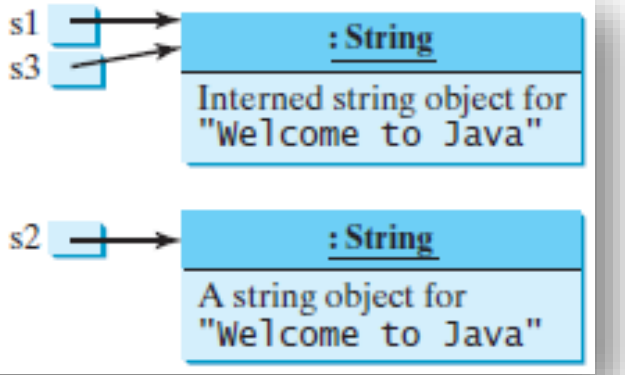
- Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called *an interned string*. For example, the following statements:

```
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));
```

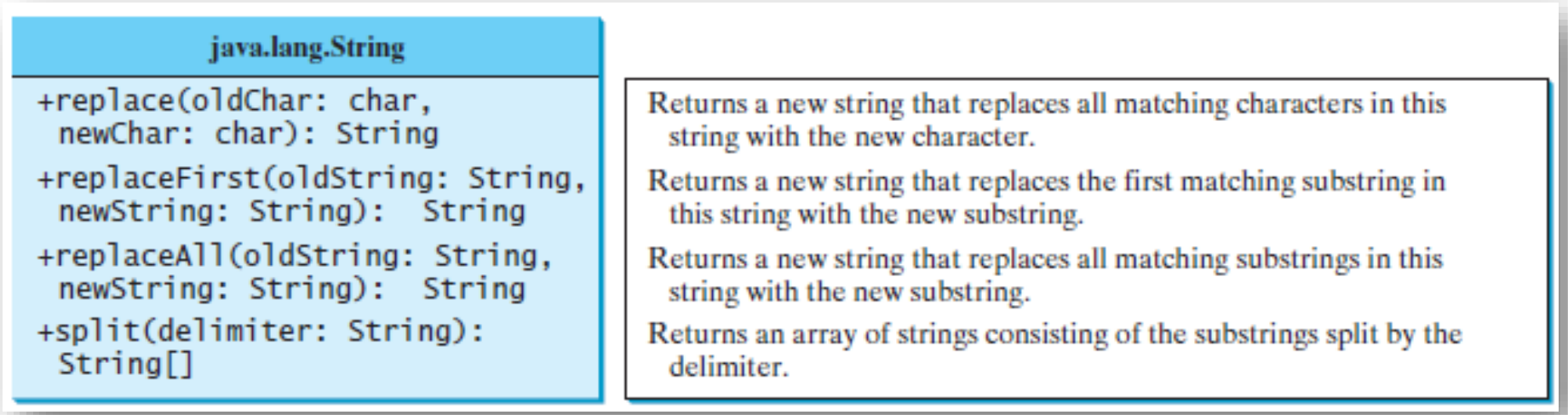


- A new object is created if you use the new operator.
- If you use the string initializer, no new object is created if the interned object is already created.
- So you will display:

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

Replacing and Splitting Strings

- The methods **replace**, **replaceFirst**, and **replaceAll** return a new string derived from the original string (without changing the original string!).



- The **split** method returns an array of strings.

Replacing Strings

- Several versions of the **replace** methods are provided to replace a character or a substring in the string with a new character or a new substring.

```
"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.
```

Splitting a String

- The **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] + " ");
```

Displays:

Java HTML Perl

Matching, Replacing and Splitting by Patterns

- You can match, replace, or split a string by specifying a pattern.
- This is an extremely useful and powerful feature, commonly known as *regular expression*. Regular expression is complex to beginning students. For this reason, two simple patterns are used here.
- The below two statements give the same result:

```
"Java".matches("Java");
"Java".equals("Java");
```

- The matches method is more powerful than equals method:

```
"Java is fun".matches("Java.*");
"Java is cool".matches("Java.*");
```

Matching, Replacing and Splitting by Patterns

- The replaceAll, replaceFirst, and split methods can be used with a regular expression.
- For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression [$+#] specifies a pattern that matches $, +, or #.

So, the output is aNNNbNNNNNNc.

Matching, Replacing and Splitting by Patterns

- The following statement splits the string into an array of strings delimited by some punctuation marks.

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

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

Thus, the string is split into **Java**, **C**, **C#**, and **C++**, which are stored in array **tokens**.

Converison 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();
```

- To convert an array of characters into a string, use the **String(char[])** constructor or the **valueOf(char[])** method of String class.

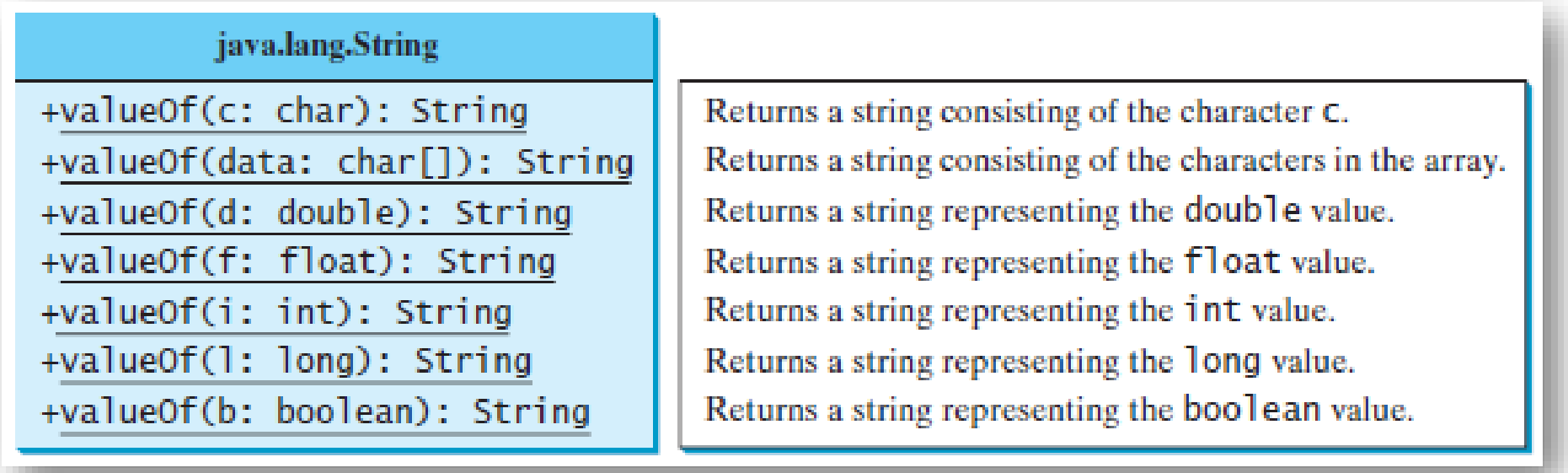
```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

Convert Character and Numbers to Strings

- The String class provides several static **valueOf** methods for converting a character, an array of characters, and numeric values to strings.
- These methods have the same name valueOf with different argument types **char, char[], double, long, int,** and **float**.

Convert Character and Numbers to Strings

- For example, to convert a double value to a string, use `String.valueOf(5.44)`.
- The return value is a string consists of characters '5', '.', '4', and '4'.



Formating Strings

- The **String** class contains the static **format** method to return a formatted string.

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

- For example:

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

Displays

```
 45.56    14AB
```

StringBuilder and StringBuffer

- The **StringBuilder/StringBuffer** class is an alternative to the **String** class.
- In general, a **StringBuilder/StringBuffer** can be used wherever a string is used.
- **StringBuilder/StringBuffer** is more flexible than **String**.
- You can add, insert, or append new contents into **StringBuilder** and **StringBuffer** objects, whereas the value of a **String** object is fixed once the string is created.

StringBuilder Constructors

- The StringBuilder class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder.
- 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.

Modifying Strings in the Builder

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 Examples

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

`stringBuilder.insert(11, "HTML and ");` changes the builder to Welcome to HTML and Java

`stringBuilder.delete(8, 11)` changes the builder to Welcome Java.

`stringBuilder.deleteCharAt(8)` changes the builder to Welcome o Java.

`stringBuilder.reverse()` changes the builder to avaJ ot emocleW.

`stringBuilder.replace(11, 15, "HTML")` changes the builder to Welcome to HTML.

`stringBuilder.setCharAt(0, 'w')` sets the builder to welcome to Java.

Modifying Strings in the Builder

- All these modification methods except **setCharAt** do two things:
  - Change the contents of the string builder
  - Return the reference of the string builder
- Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored:

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
              stringBuilder.reverse();
```

The 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.

Note:

(1) The capacity is the number of characters the string builder is able to store without having to increase its size.

(2) The **length()** method returns the number of characters actually stored in the string builder.

(3) The **setLength(newLength)** method sets the length of the string builder.

(4) If the builder's capacity is exceeded, capacity is automatically increased: **2 * (the previous capacity + 1)**

Problem: Checking Palindromes Ignoring Non-alphanumeric Characters

This example gives a program that counts the number of occurrence of each letter in a string. Assume the letters are not case-sensitive.

Palindrome Check

Intro to Java Programming, Y. Daniel Liang - PalindromeIgnoreNonAlphanumeric.java (pearsoncmg.com)