Chapter 10 Object-Oriented Thinking

Object-Oriented Thinking

Procedural programming

- data and operations on the data are separate

OO programming

- places data and operations on the data in an object.
- organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities.
- Classes provide more flexibility and modularity for building reusable software. It makes programs easier to develop and easier to maintain.

A Java program can be viewed as a collection of cooperating objects.

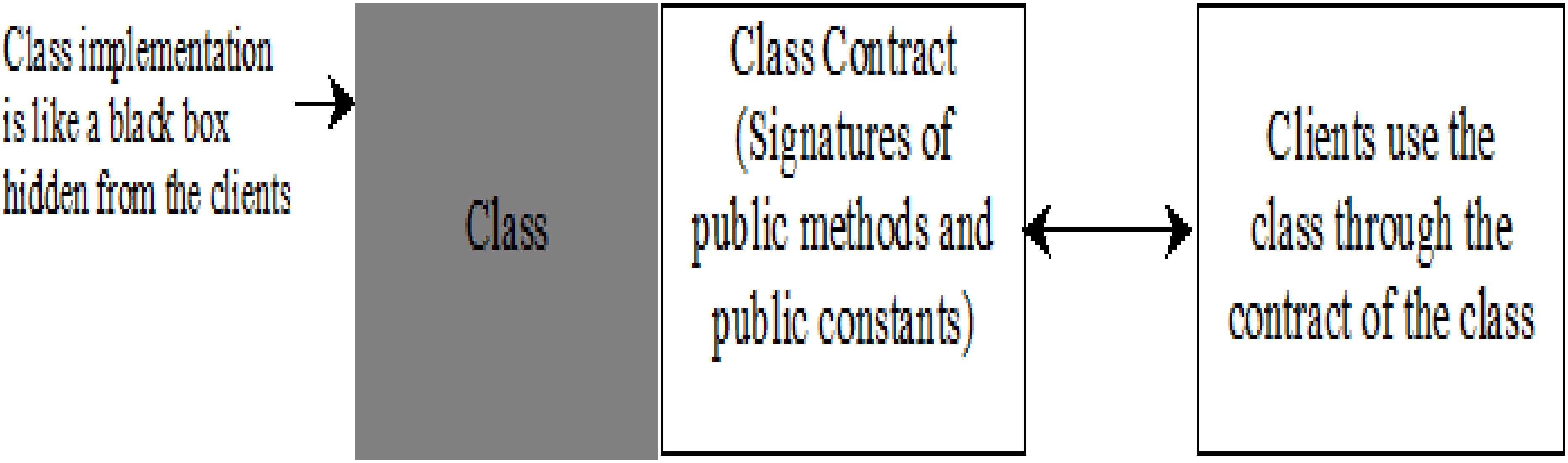
Class Abstraction and Encapsulation

Class abstraction :

separate **class implementation** from the **use of the class**.

Class Encapsulation :

- *the user* know how to use the class.
- but does not need to know how the class is implemented.



Class implementation is like a black box hidden from the clients

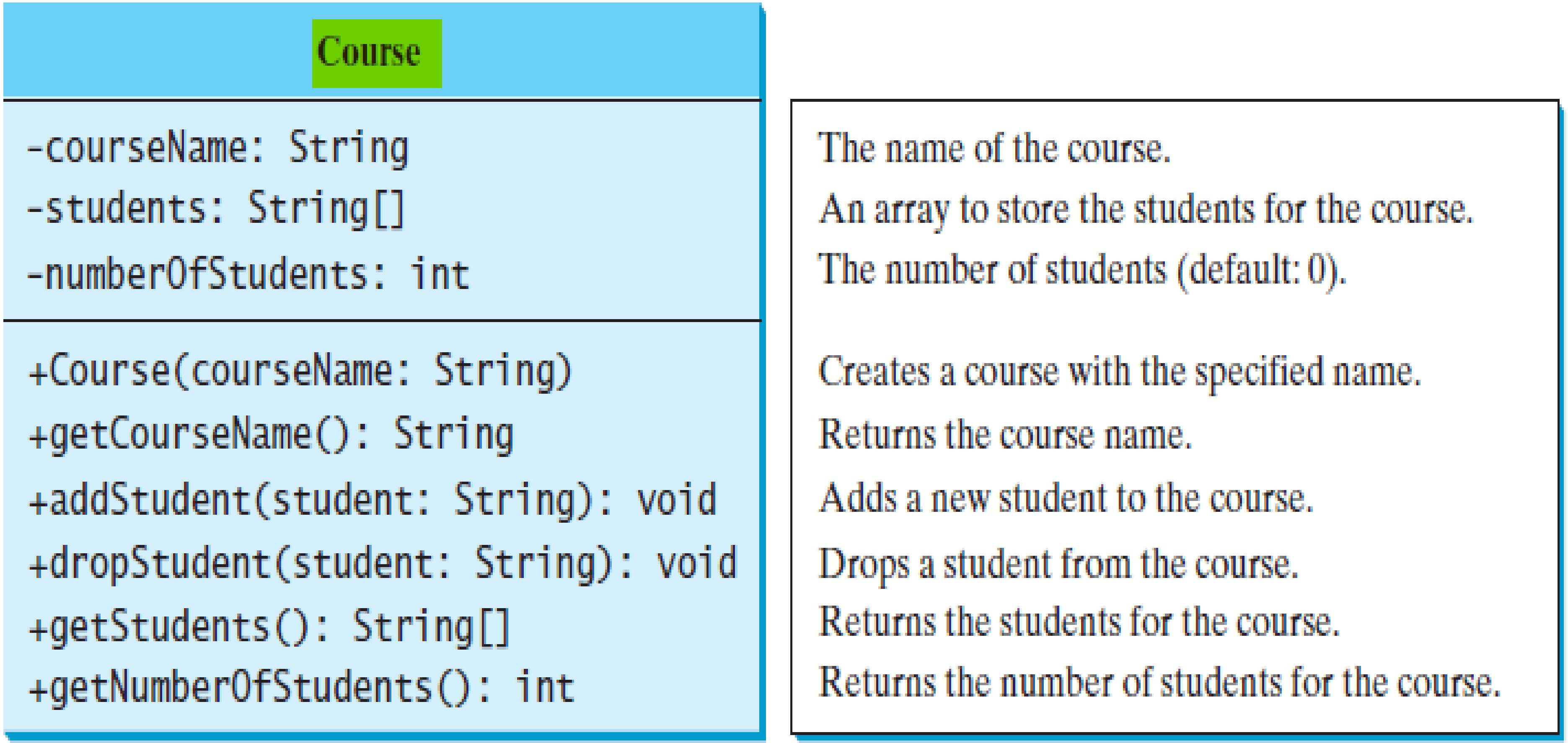
Class

Class Contract (Signatures of public methods and public constants)

Clients use the class through the contract of the class

Example: The Course Class

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 10.9.



Course

-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students for the course.
+getNumberOfStudents(): int	Returns the number of students for the course.

Write a test program to create two courses and adds students to them

TestCourse.java

create a course

add a student

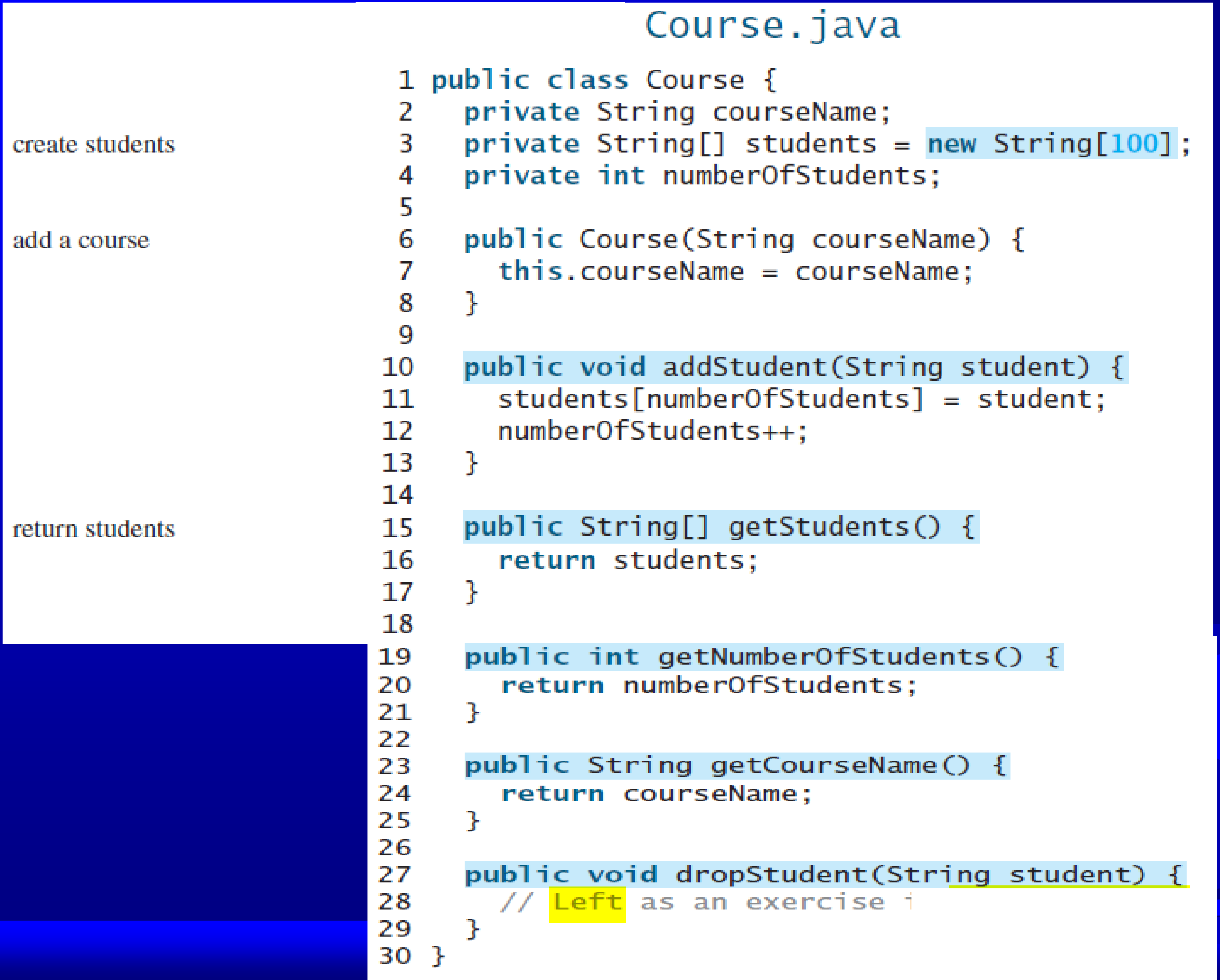
number of students

return students

```java
public class TestCourse {
  public static void main(String[] args) {
    Course course1 = new Course("Data Structures");
    Course course2 = new Course("Database Systems");

    course1.addStudent("Peter Jones");
    course1.addStudent("Brian Smith");
    course1.addStudent("Anne Kennedy");

    course2.addStudent("Peter Jones");
    course2.addStudent("Steve Smith");

    System.out.println("Number of students in course1: "
      + course1.getNumberOfStudents());
    String[] students = course1.getStudents();
    for (int i = 0; i < course1.getNumberOfStudents(); i++)
      System.out.print(students[i] + ", ");

    System.out.println();
    System.out.print("Number of students in course2: "
      + course2.getNumberOfStudents());
  }
}
```

```
Number of students in course1: 3
Peter Jones, Brian Smith, Anne Kennedy,
Number of students in course2: 2
```

Course class is implemented



Course.java

create students

add a course

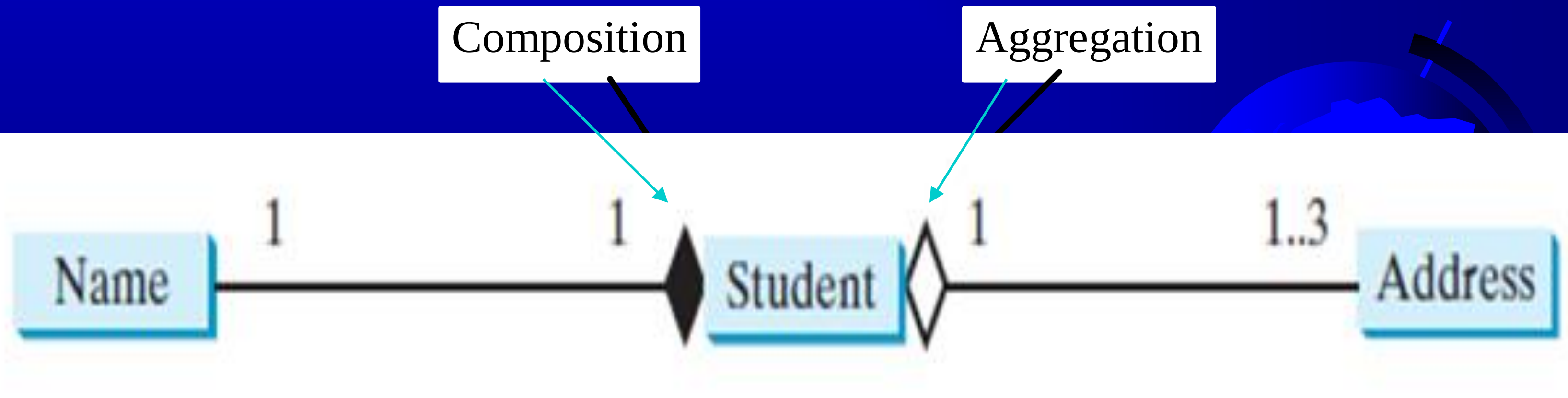
return students

```java
public class Course {
  private String courseName;
  private String[] students = new String[100];
  private int numberOfStudents;

  public Course(String courseName) {
    this.courseName = courseName;
  }

  public void addStudent(String student) {
    students[numberOfStudents] = student;
    numberOfStudents++;
  }

  public String[] getStudents() {
    return students;
  }

  public int getNumberOfStudents() {
    return numberOfStudents;
  }

  public String getCourseName() {
    return courseName;
  }

  public void dropStudent(String student) {
    // Left as an exercise i
  }
}
```

Object Composition

An object can **contain** another object

- models ***has-a*** relationships

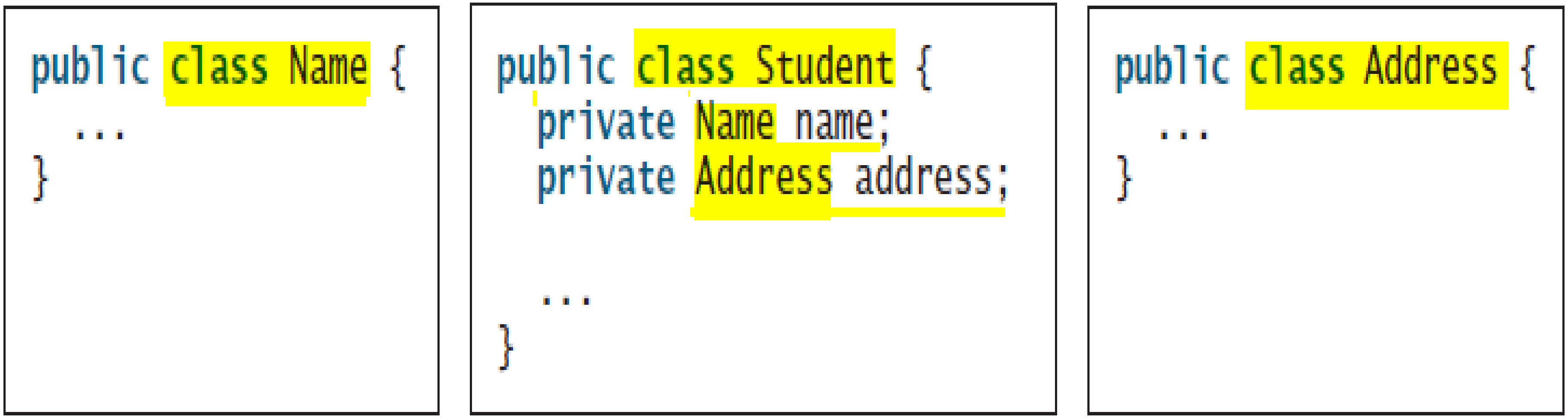


Composition

Aggregation

A student has a name and an address.

An aggregation relationship is usually represented as a data field in the aggregating class.



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

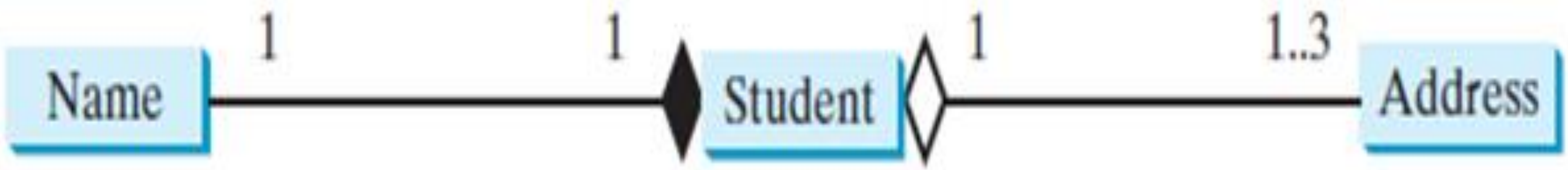
Aggregated class

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

  ...
}
```

Aggregating class

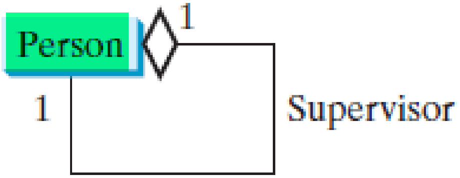
```
public class Address {
  ...
}
```

Aggregated class



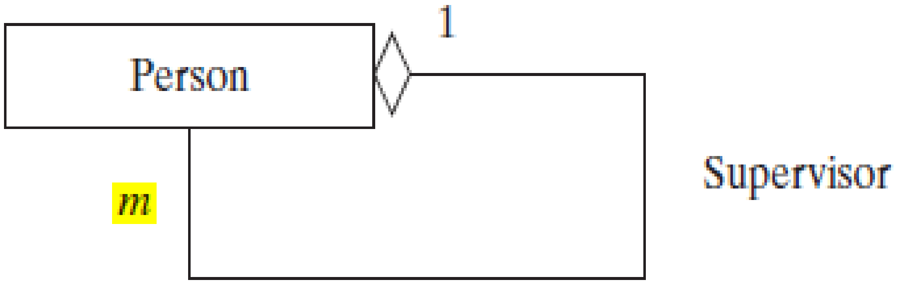
A student has a name and an address.

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;

    ...
 }
```

If a person may have several supervisors

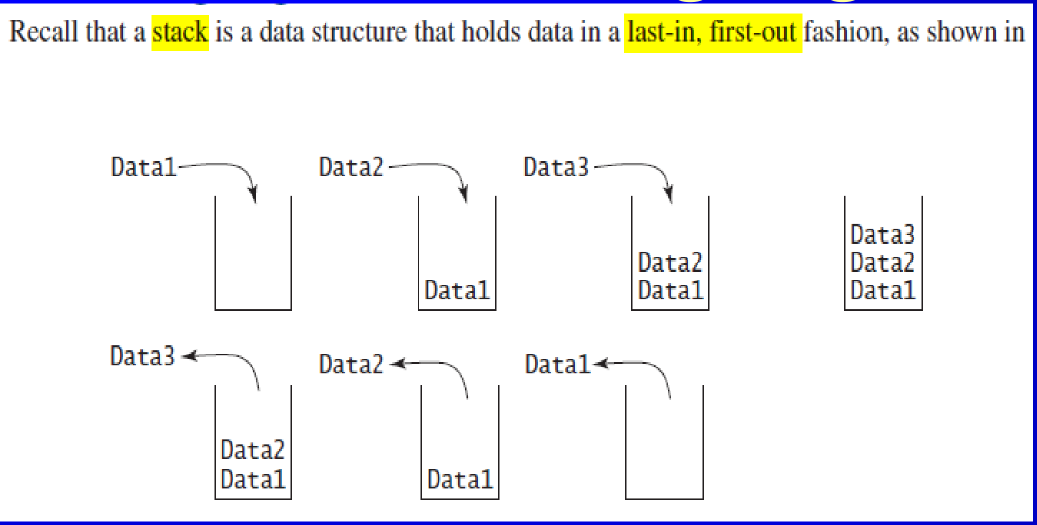


(a)

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

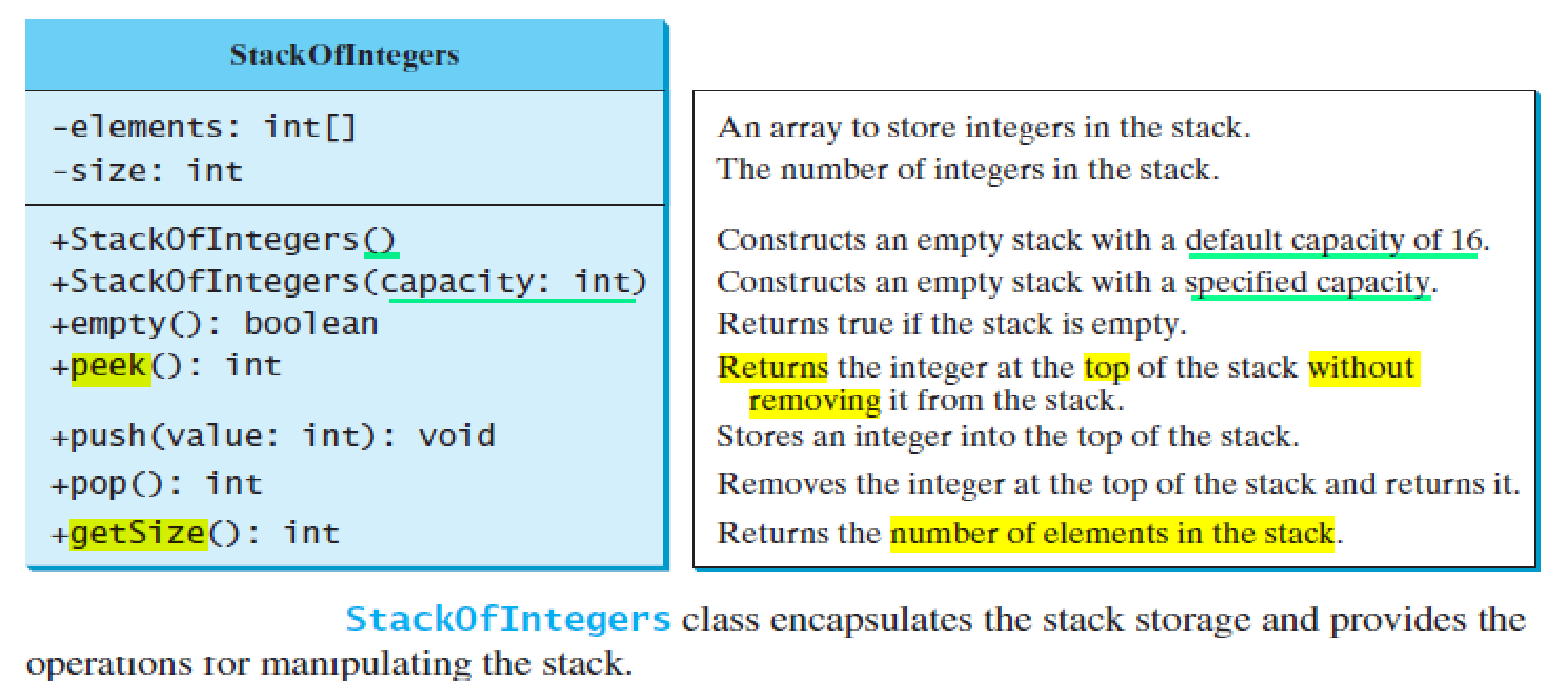
(b)

Designing a Class for Stacks



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

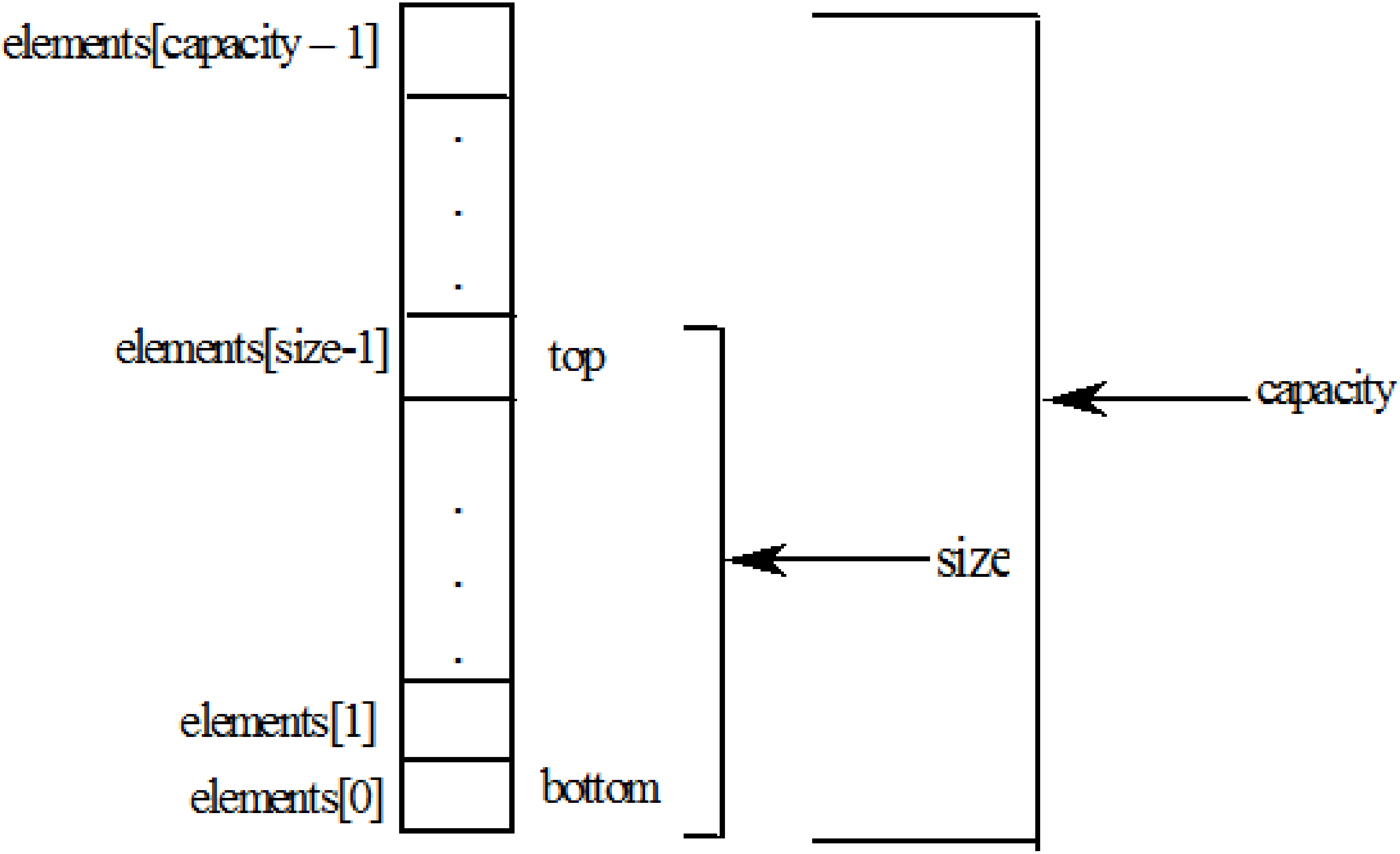
Next, we define a class to model stacks that hold **int** values.

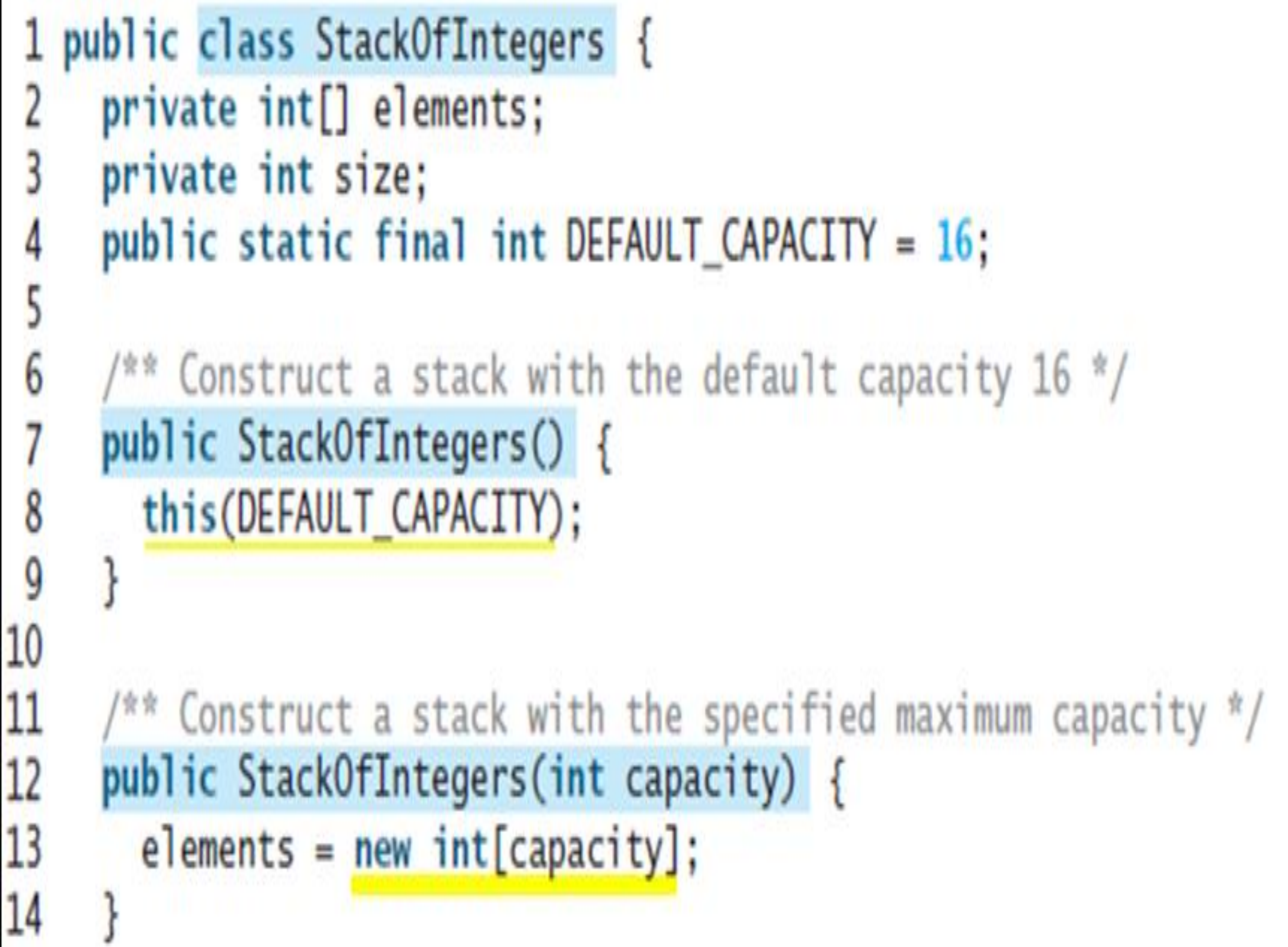


StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): void	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

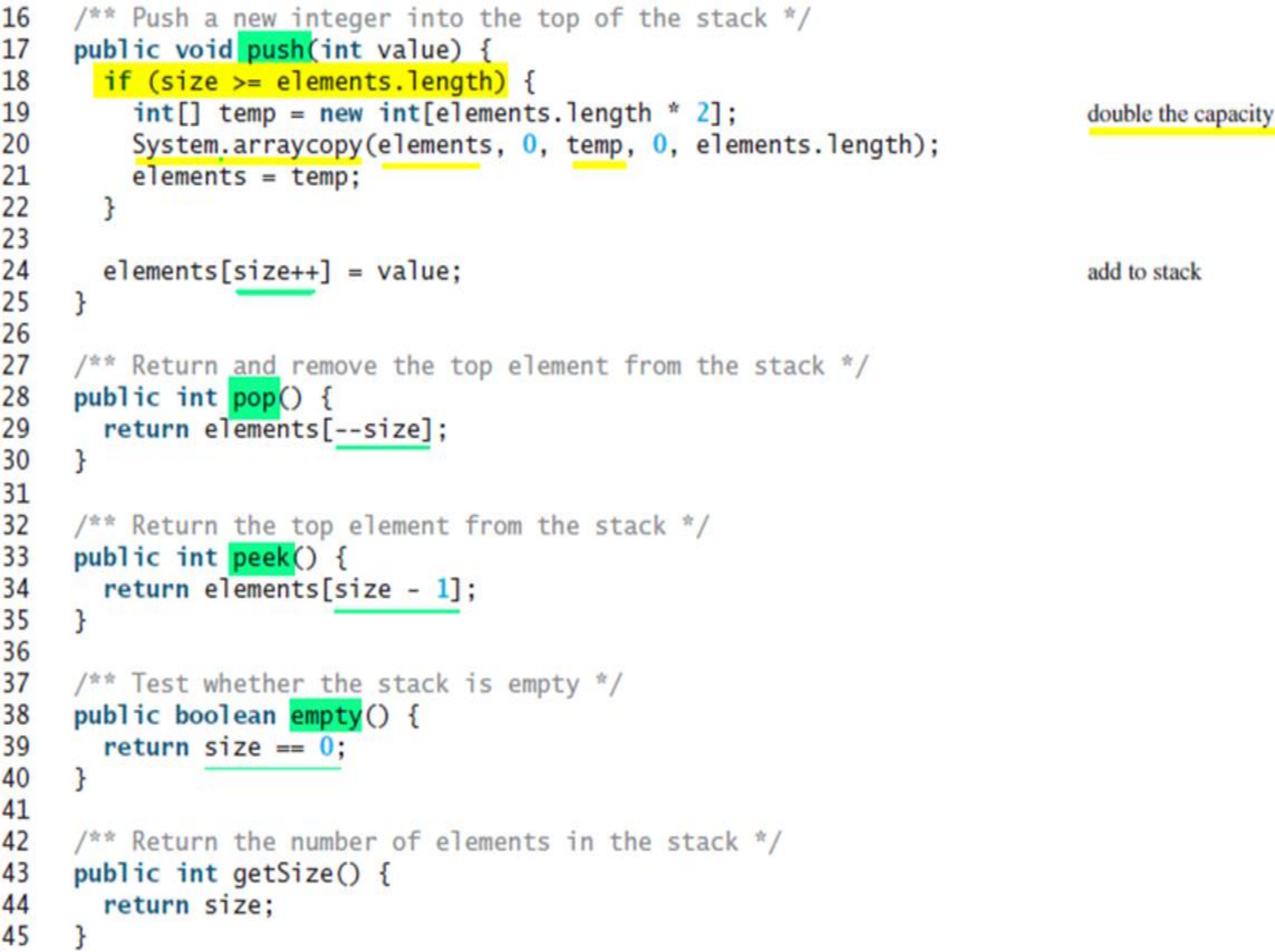
StackOfIntegers class encapsulates the stack storage and provides the operations for manipulating the stack.

Implementing `StackOfIntegers` Class





```java
public class StackOfIntegers {
  private int[] elements;
  private int size;
  public static final int DEFAULT_CAPACITY = 16;

  /** Construct a stack with the default capacity 16 */
  public StackOfIntegers() {
    this(DEFAULT_CAPACITY);
  }

  /** Construct a stack with the specified maximum capacity */
  public StackOfIntegers(int capacity) {
    elements = new int[capacity];
  }
```



```
  /** Push a new integer into the top of the stack */
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];                double the capacity
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }

    elements[size++] = value;                                   add to stack
  }

  /** Return and remove the top element from the stack */
  public int pop() {
    return elements[--size];
  }

  /** Return the top element from the stack */
  public int peek() {
    return elements[size - 1];
  }

  /** Test whether the stack is empty */
  public boolean empty() {
    return size == 0;
  }

  /** Return the number of elements in the stack */
  public int getSize() {
    return size;
  }
```

Designing a Class

Coherence

A class should describe a single entity, and all the class operations should support a coherent purpose.

You should not combine students and staff in the same class, because students and staff have different entities.

Separating responsibilities

A single entity with too many responsibilities can be broken into several classes to separate responsibilities.

- E.g., the classes **String and StringBuilder** deal with strings, but have different responsibilities.
  - the **String** class deals with immutable strings
  - the **StringBuilder** class is for creating mutable strings

    （last sections of this Chapter）

Consistency

Popular **styles**:

Place the data declaration before the constructor, and place constructors before methods.

Always provide a **constructor to initialize variables**

Always provide **a public no-arg constructor** for constructing a default instance.

Encapsulation

A class should use the **private** modifier to hide its data from direct access by clients.

- Provide a **get** method only if you want the field to be readable
- provide a **set** method only if you want the field to be updateable.

For example, the **Course** class

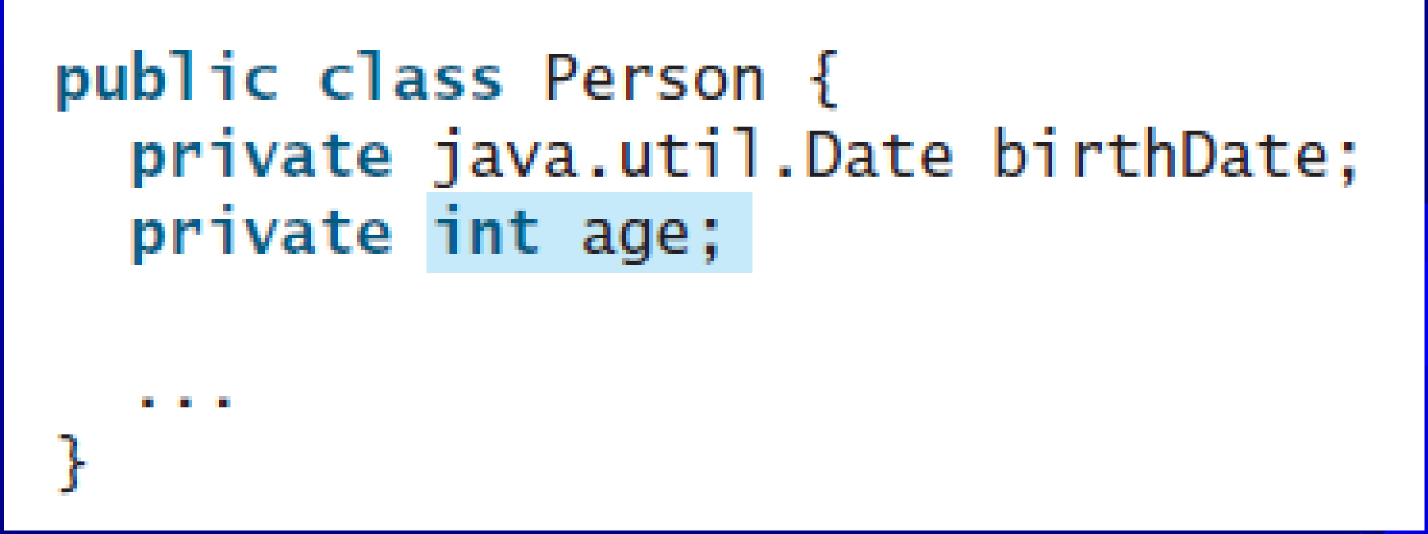
- provides a **get** method for **courseName**
- but no **set** method, because the user is not allowed to change the course name, once it is created.

Clarity

You should not declare a data field that can be derived from other data fields.

For example, the following **Person** class

– age should not be declared as a data field.



```
public class Person {
  private java.util.Date birthDate;
  private int age;

  ...
}
```

Using the static Modifier

Instance variable/method: dependent on a specific instance

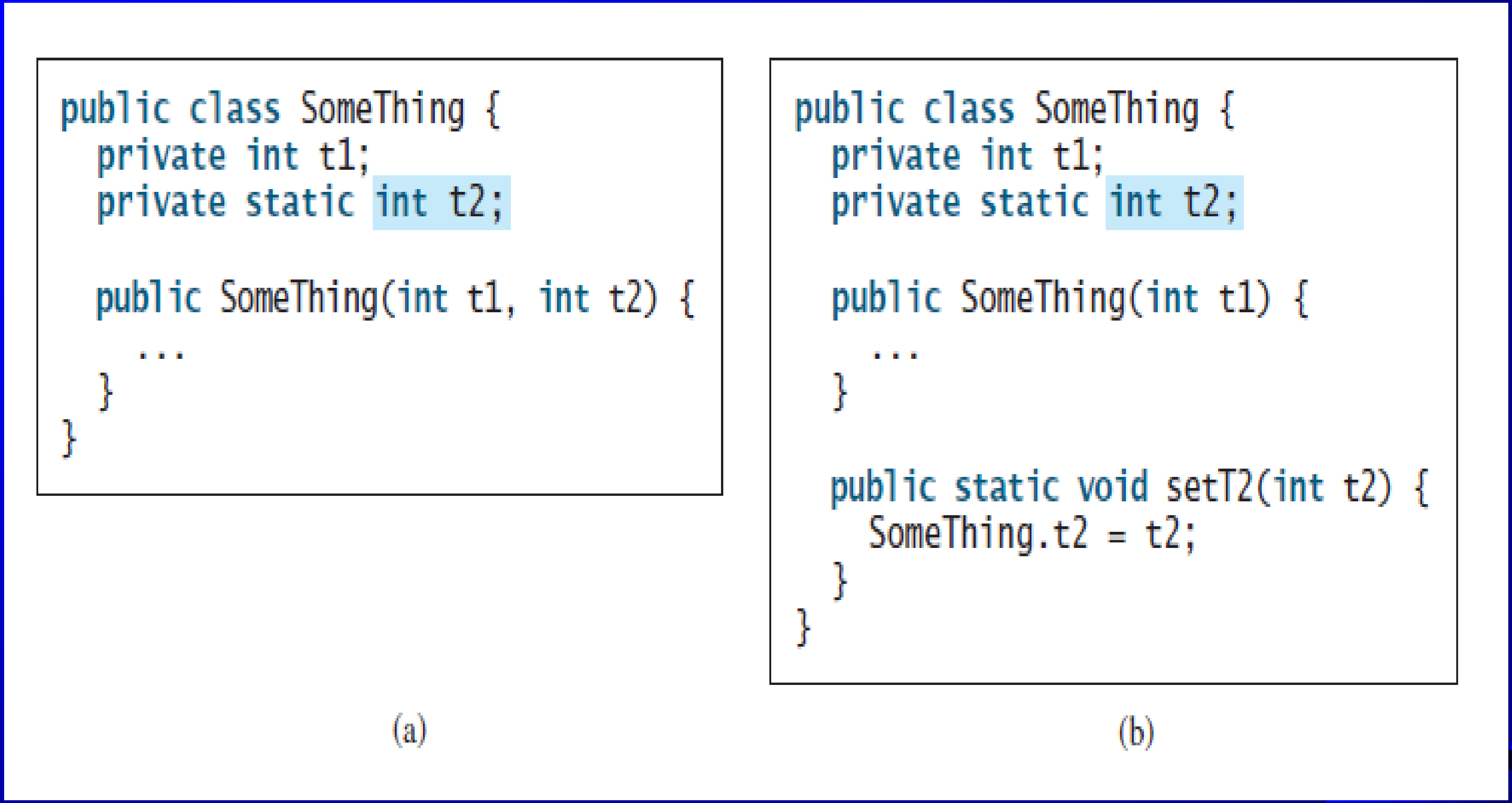
Static variable/method: shared by all the instances of a class

For example, in the class **Circle**

- the static variable **numberOfObjects**
- the static method **getNumberOfObjects**

* Shared by all the objects of the **SimpleCircle1** class , not tied to any specific instance

Which one is better?



```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

To change the static data field

- Use a **set** method
- **Do not** pass a parameter from a constructor to initialize a static data field.

Wrapper Classes

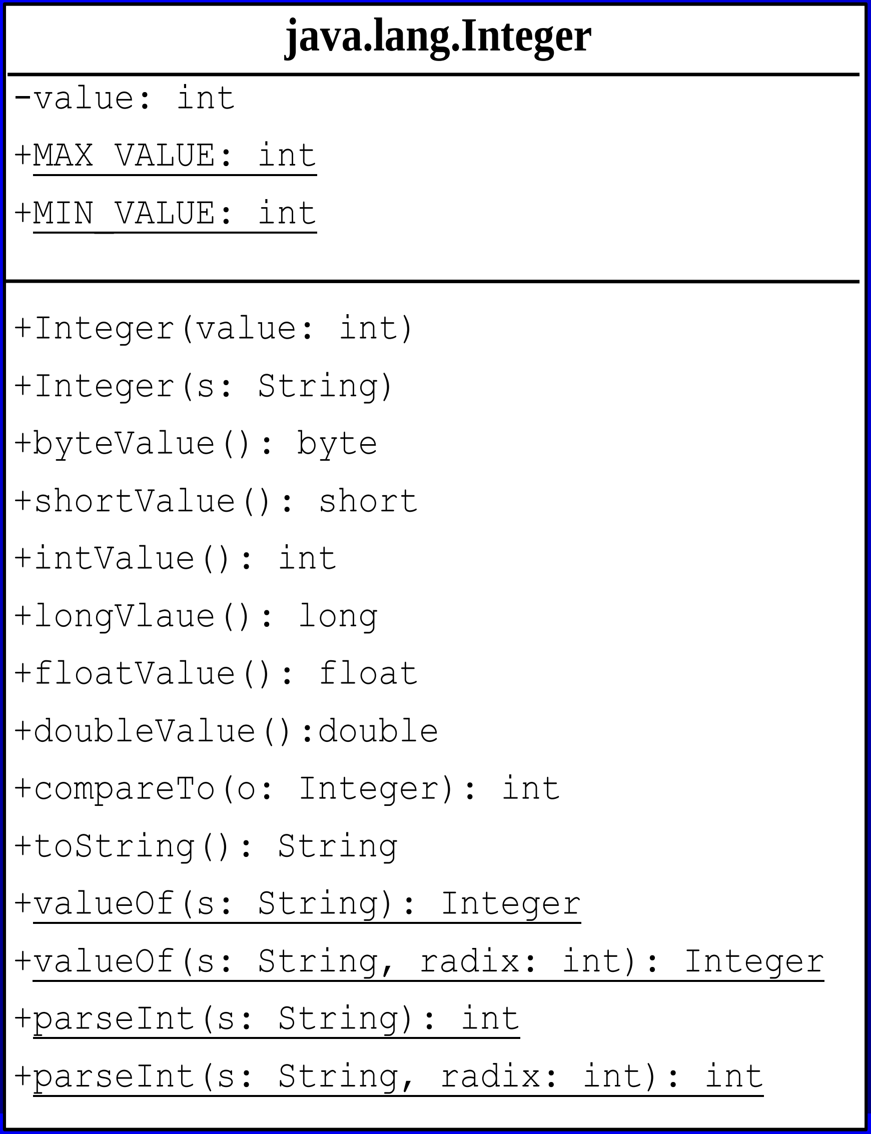
- ❑ Boolean
- ❑ Character
- ❑ Short
- ❑ Byte
- ❑ Integer
- ❑ Long
- ❑ Float
- ❑ Double

NOTE:

(1) Not have no-arg constructors.

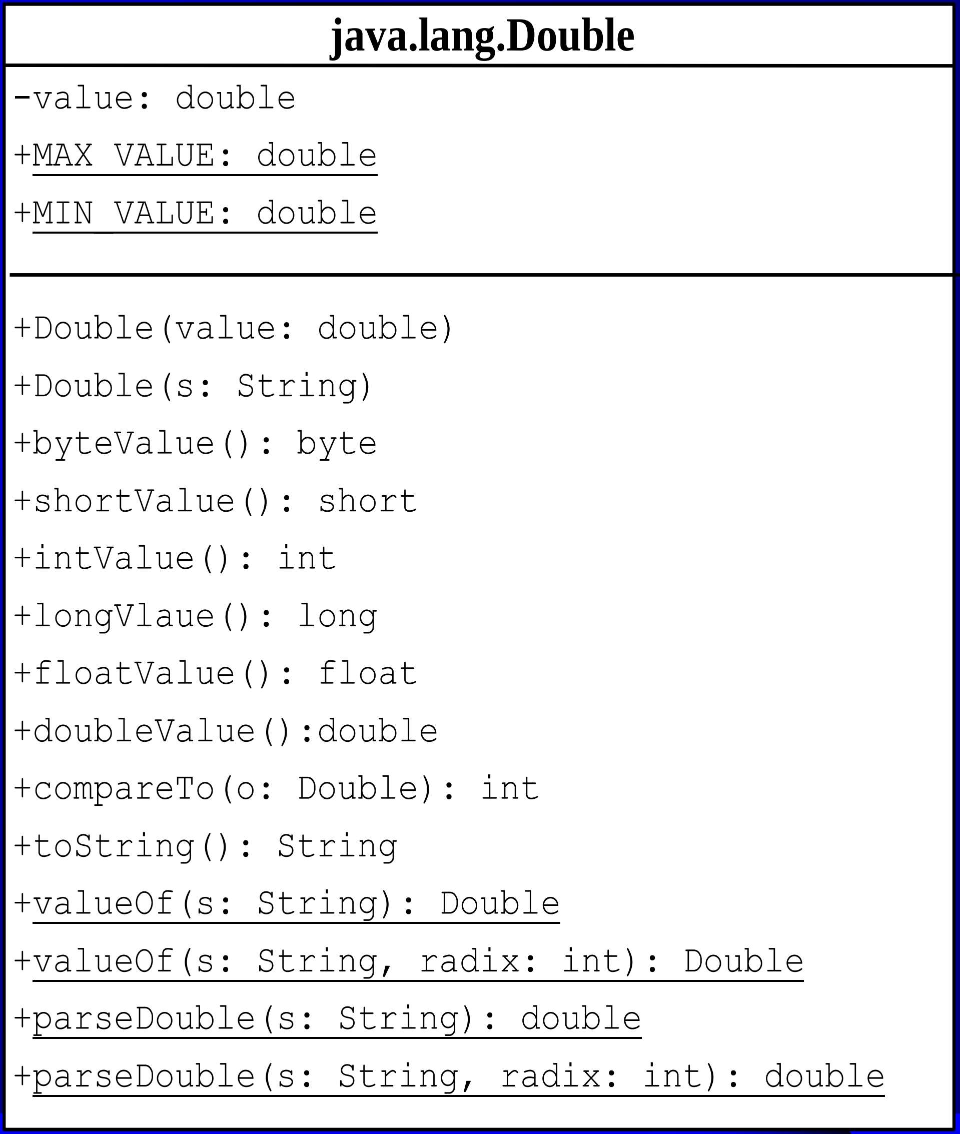
(2) Immutable : object values cannot be changed once being created.

The Integer and Double Classes



java.lang.Integer

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

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): 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
+longVlaue(): 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
```

The Integer Class and the Double Class

- Constructors
- Class Constants MAX_VALUE, MIN_VALUE
- Conversion Methods

Numeric Wrapper Class Constructors

To construct a wrapper object from a primitive data type value or from a string.

```
public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)
```

Numeric Wrapper Class Constants

MAX_VALUE

MIN_VALUE

the max/min primitive data type

- byte, short, int, long
- *positive* float , double

Conversion Methods

Methods: convert numeric objects

into primitive type values

doubleValue(), floatValue(),

intValue(), longValue(), shortValue()

The Static valueOf Methods

valueOf(String s) : creates a new object initialized with the string value

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

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

Automatic Conversion

Primitive Types ⇔ Wrapper Class Types

```
Integer[] intArray = {new Integer(2),
                      new Integer(4),
                      new Integer(3)};
```

can be simplified as:

```
Integer[] intArray = {2,4,3};
```

```
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

BigInteger and BigDecimal

BigInteger class

for very large integers

BigDecimal class

for high precision floating-point values

Both classes:

- in the java.math package.
- *immutable*.
- extend the Number class

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

LargeFactorial

```
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 : Construct Strings

String newString = new String(stringLiteral);

- **String s = new String();**
- **String message = new String("Welcome to Java");**

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

- **String message = "Welcome to Java";**

create a string from an array of characters. “Good Day”:

- **char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};**
- **String message = new String(charArray);**

A **String** variable holds a reference

to a **String** object that stores a string value

Strings Are Immutable

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

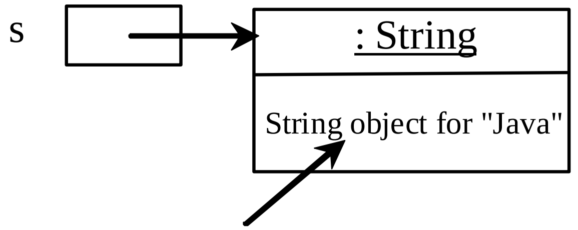
Does the following code change the contents of the string?

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

Trace Code

```
String s = "Java";

s = "HTML";
```

After executing `String s = "Java";`



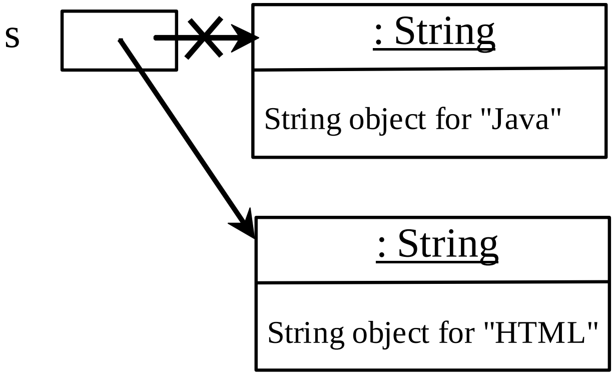
Contents cannot be changed

animation

Trace Code

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

After executing `s = "HTML";`



This string object is now unreferenced

Interned Strings

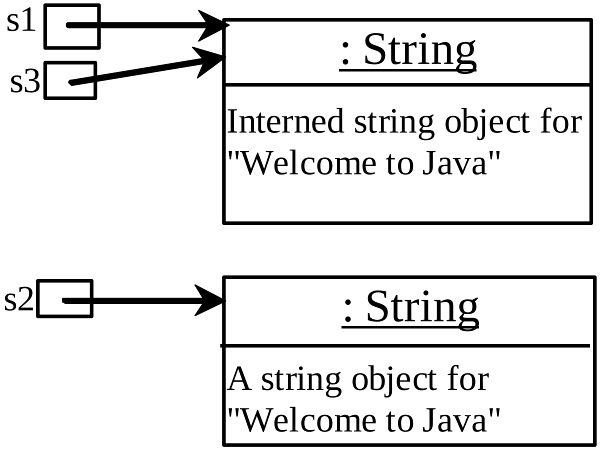
To improve efficiency and save memory, JVM uses a unique instance for **string literals**.

Such an instance is called ***interned***.

For example, the following statements:

Examples

```
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

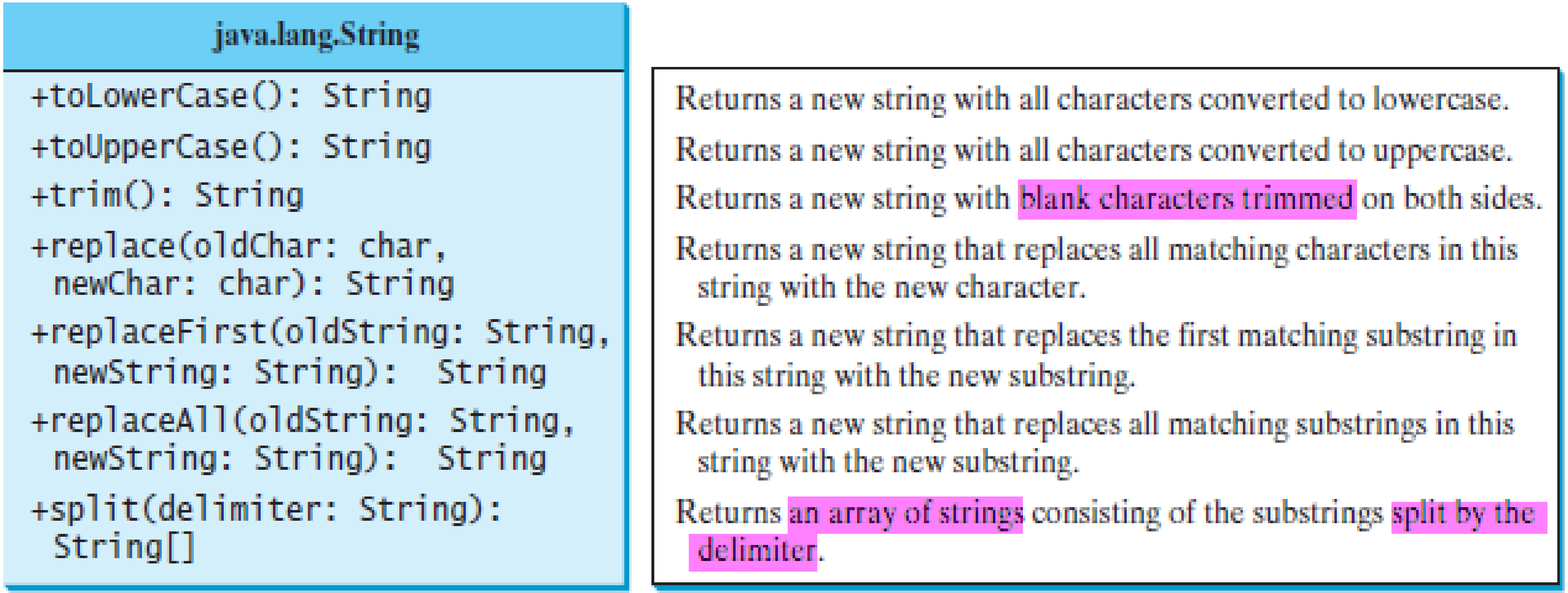
s1 == s2 is false

s1 == s3 is true

If you use the new operator, a new object is created

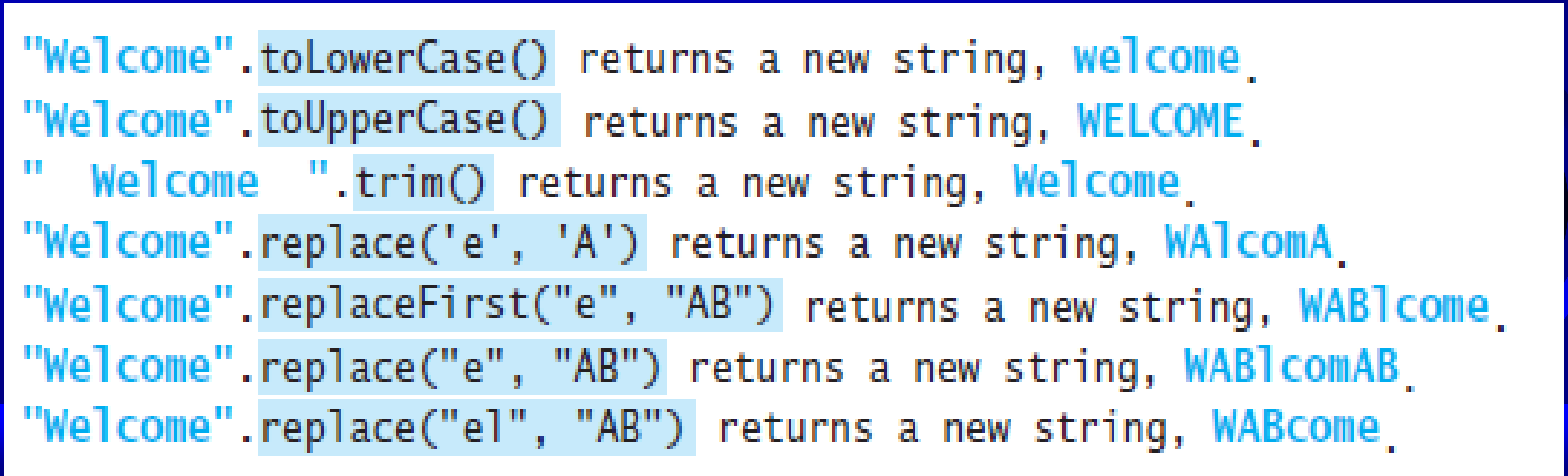
If you use the string initializer, no new object is created if the interned object is already created.

Converting, Replacing, and Splitting Strings



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 blank 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.



```
"Welcome".toLowerCase() returns a new string, welcome.
"Welcome".toUpperCase() returns a new string, WELCOME.
"  Welcome  ".trim() returns a new string, Welcome.
"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.
```

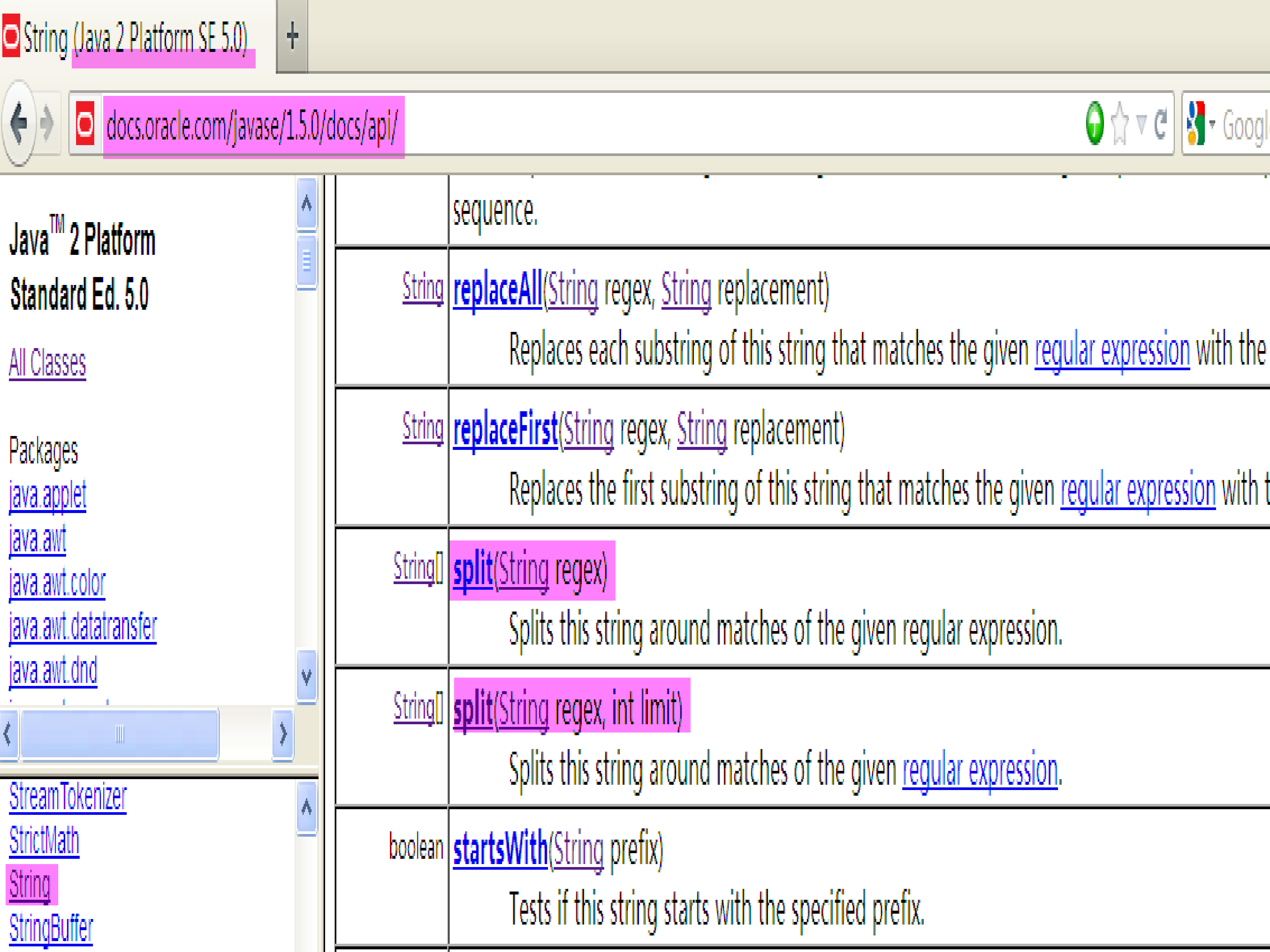
Splitting a String

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

```
Java HTML Perl
```

Have the same effects:

```
"Java#HTML#Perl".split("#", 0);
"Java#HTML#Perl".split("#");
```


String (Java 2 Platform SE 5.0)
docs.oracle.com/javase/1.5.0/docs/api/
Googl

Java™ 2 Platform
Standard Ed. 5.0

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd

StreamTokenizer
StrictMath
String
StringBuffer

	sequence.
String	replaceAll(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the
String	replaceFirst(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with t
String[]	split(String regex) Splits this string around matches of the given regular expression.
String[]	split(String regex, int limit) Splits this string around matches of the given regular expression.
boolean	startsWith(String prefix) Tests if this string starts with the specified prefix.

Matching, Replacing and Splitting by Patterns

Matches method:

- At first glance, similar to the **equals** method. E.g., both evaluate to **true**.

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

- However, more powerful. It can match a set of strings that follow a pattern.

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

// the regular expression ".*" matches any zero or more characters.

Matching, Replacing and Splitting by Patterns

The replaceAll, replaceFirst, and split methods can be used with a regular expression.

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

$, +, **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.

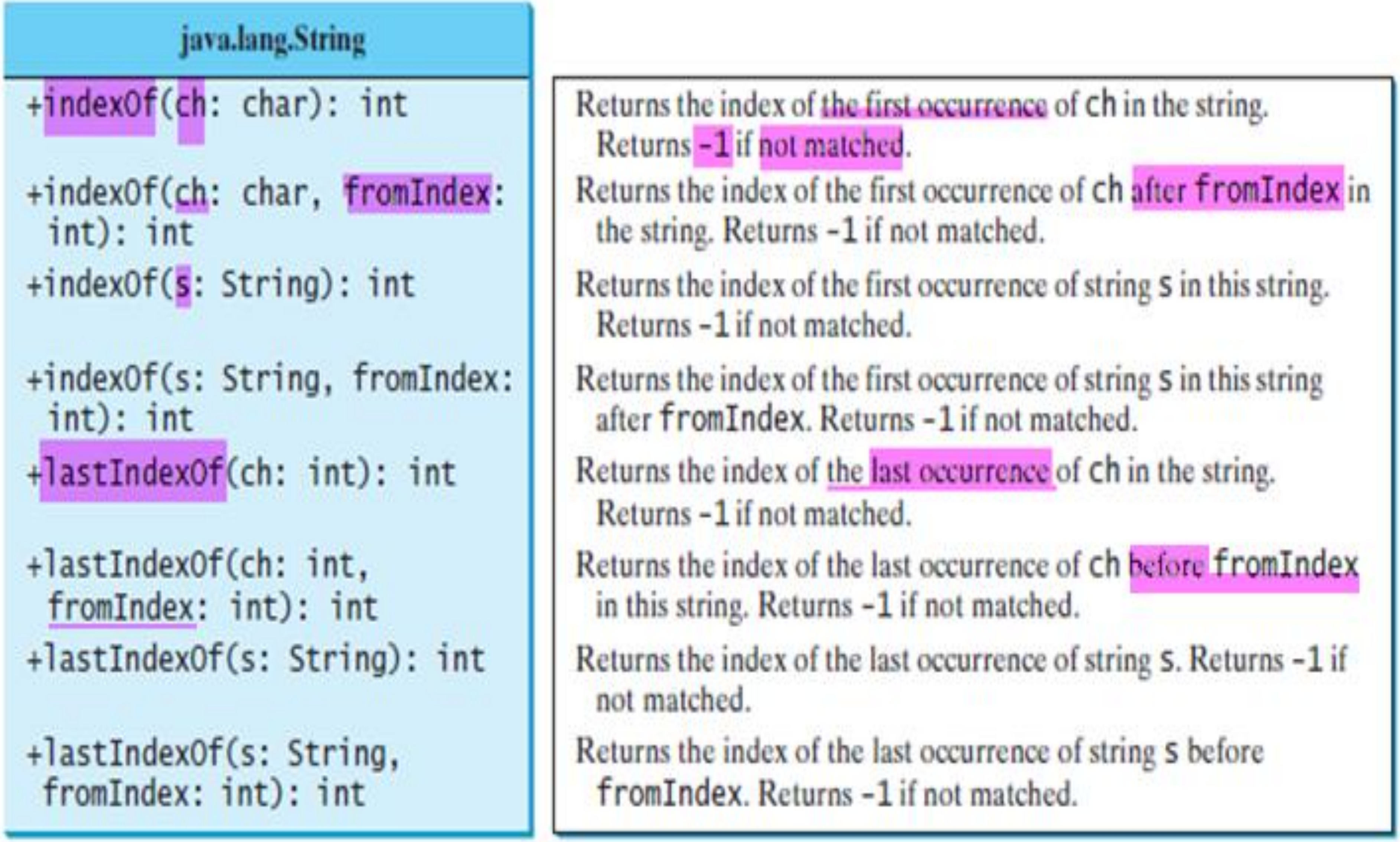
```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

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

., **,**, **:**, **;**, or **?**

Java, **C**, **C#,** **C++** : are stored into array **tokens**.

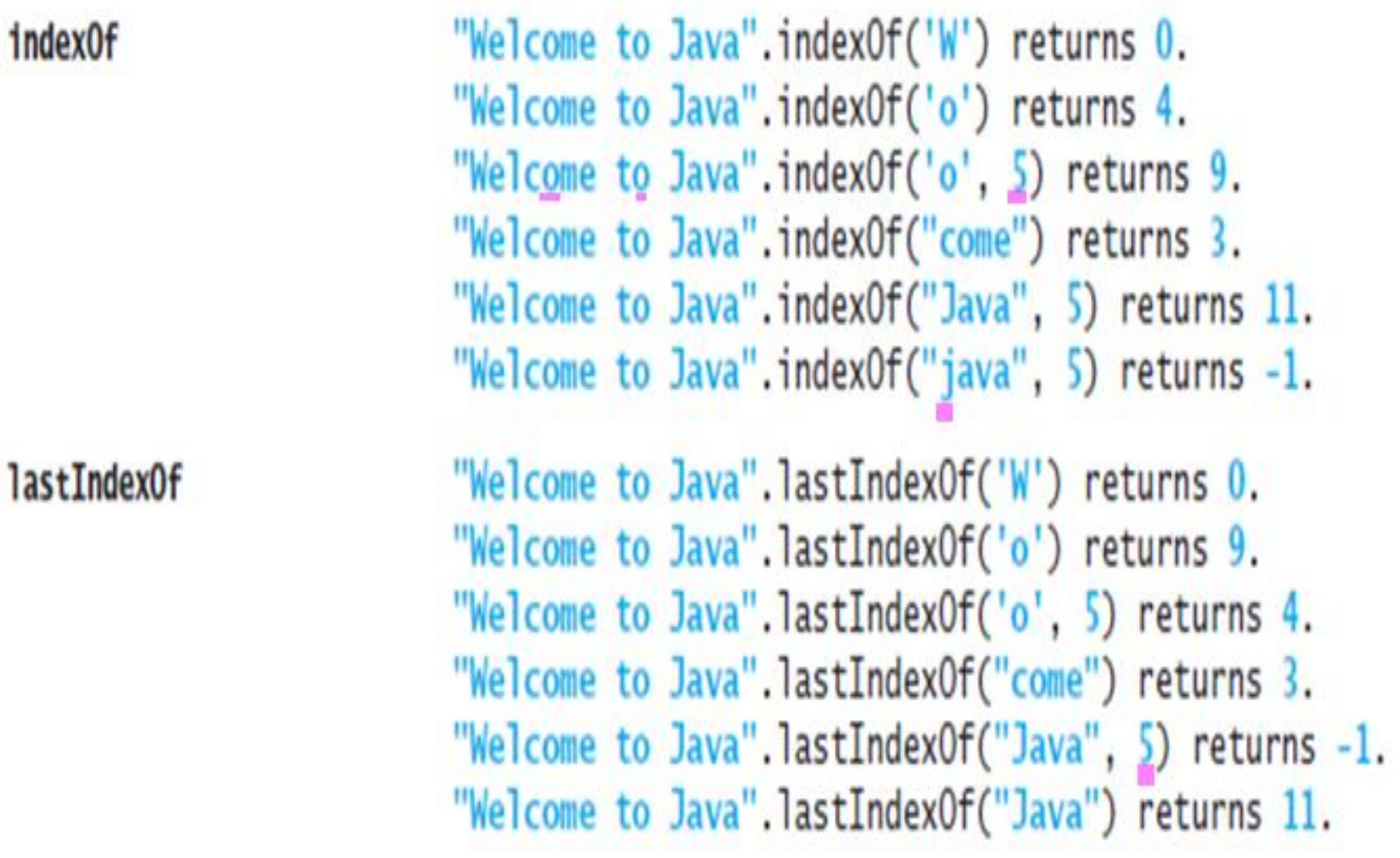
Finding a Character or a Substring in a String

The String class provides several overloaded methods:



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.

Finding a Character or a Substring in a String



indexOf	"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.
lastIndexOf	"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

convert a string to an array of characters

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

convert an array of characters to a string

- using the **String** constructor.

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

- using the **valueOf** method.

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

copy a substring (from index **srcBegin** to index **srcEnd-1)** into a character array (**dst** starting from index **dstBegin)**

- **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**

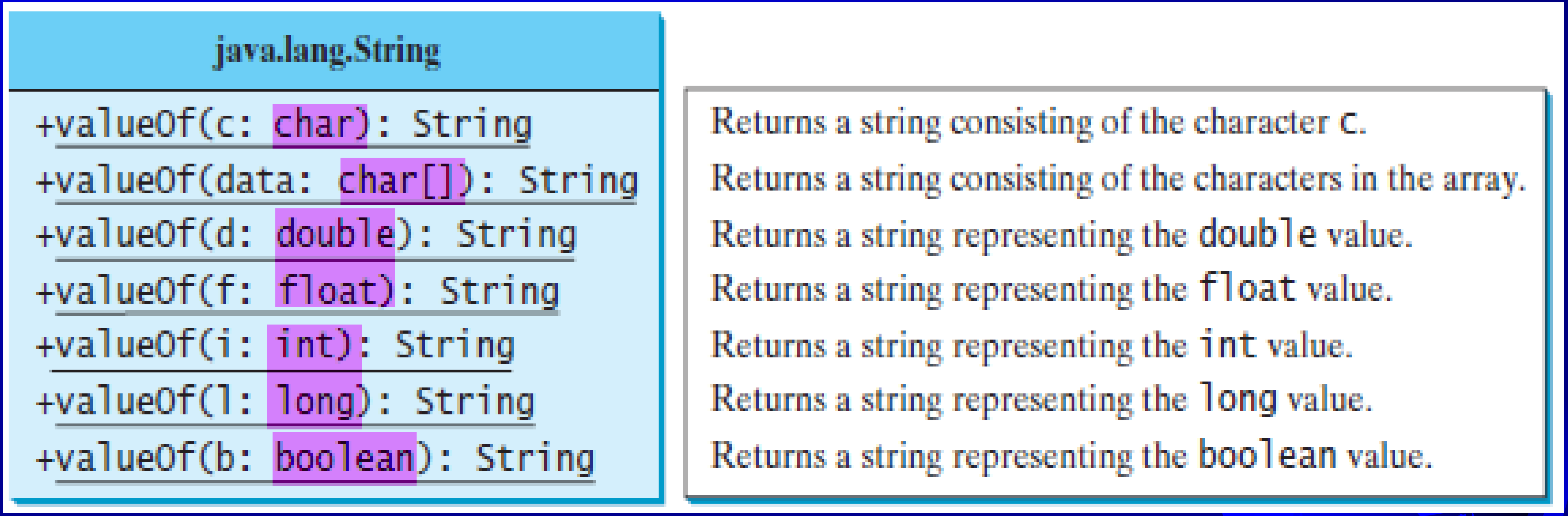
char[] dst = {**'J'**, **'A'**, **'V'**, **'A'**, **'1'**, **'3'**, **'0'**, **'1'**};

"CS3720".*getChars*(2, **6**, dst, **4**);

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

Convert Character and Numbers to Strings

String class provides overloaded static *valueOf* methods for converting a character, an array of 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
```

Returns a string consisting of the character c.

Returns a string consisting of the characters in the array.

Returns a string representing the double value.

Returns a string representing the float value.

Returns a string representing the int value.

Returns a string representing the long value.

Returns a string representing the boolean value.

For example, String.valueOf(5.44). the return value is string “5.44”

Convert Strings to Numbers

convert a string to a **double** value

Double.parseDouble(str)

convert a string to an **int** value.

Integer.parseInt(str)

Formatting Strings

static method in the **String** class to create a formatted string.

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

For example,

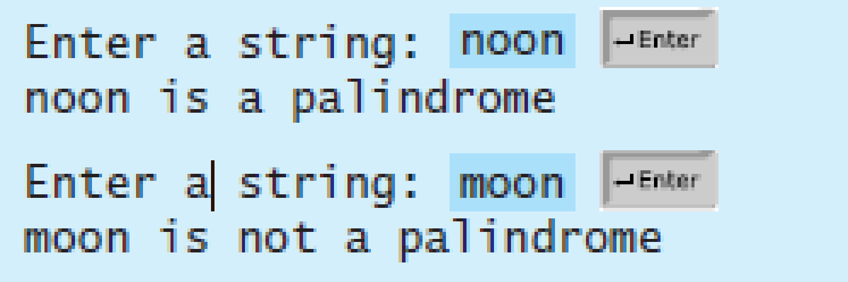
String s = ***String.format*("%5.2f", 45.556);**

– creates a formatted string **"45.56"**.

Problem: Finding Palindromes

Checking whether a string is a palindrome

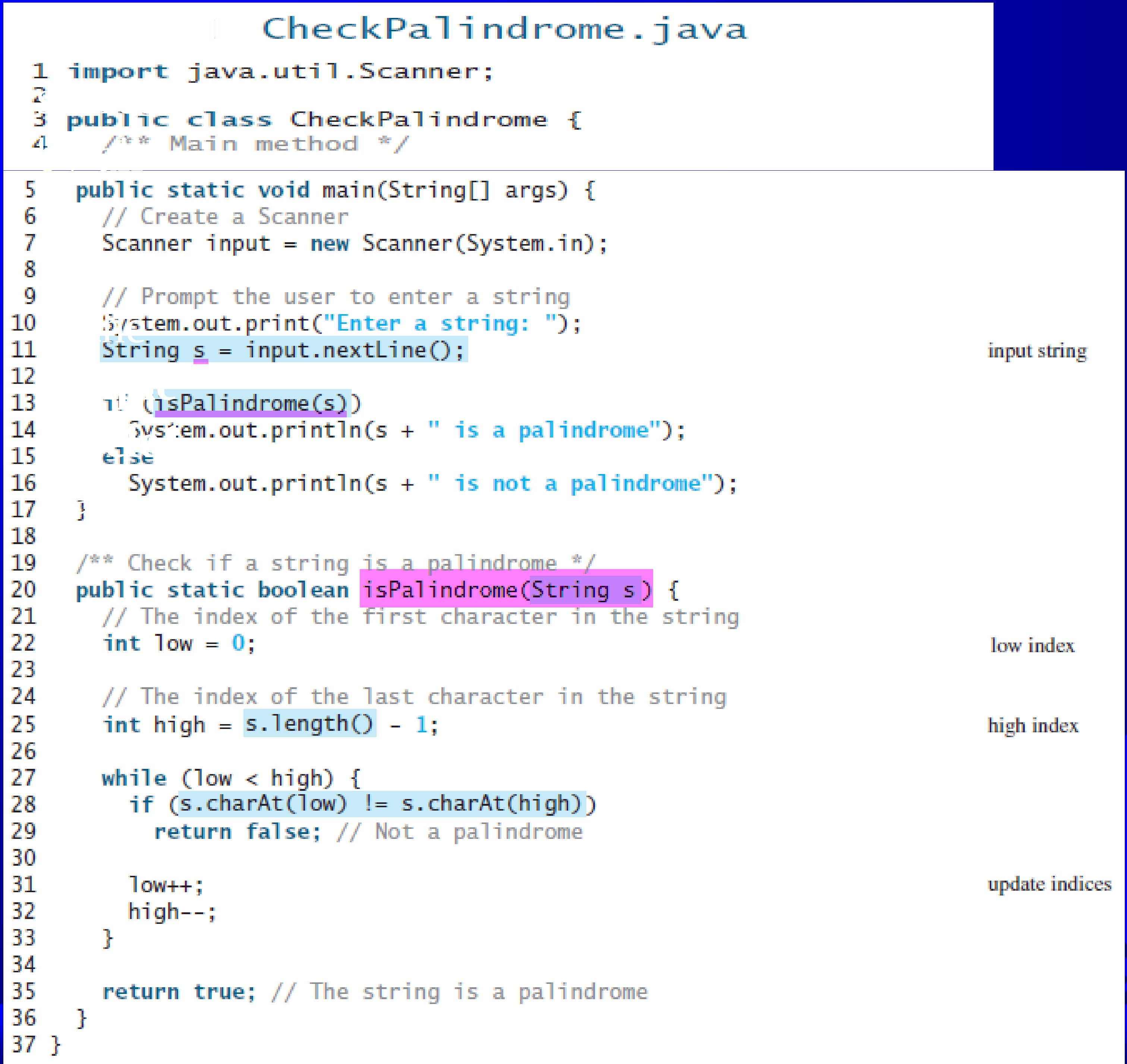
- a string that reads the same forward and backward. (e.g. “mom,” “dad,”and “noon,”)



```
Enter a string: noon
noon is a palindrome

Enter a string: moon
moon is not a palindrome
```

Solution :

- check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. This process continues until
  - a mismatch is found
  - or all characters are checked, except for the middle character if the string has an odd number of characters.



CheckPalindrome.java

```java
import java.util.Scanner;

public class CheckPalindrome {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    if (isPalindrome(s))
      System.out.println(s + " is a palindrome");
    else
      System.out.println(s + " is not a palindrome");
  }

  /** Check if a string is a palindrome */
  public static boolean isPalindrome(String s ) {
    // The index of the first character in the string
    int low = 0;

    // The index of the last character in the string
    int high = s.length() - 1;

    while (low < high) {
      if (s.charAt(low) != s.charAt(high))
        return false; // Not a palindrome

      low++;
      high--;
    }

    return true; // The string is a palindrome
  }
}
```

input string

low index

high index

update indices

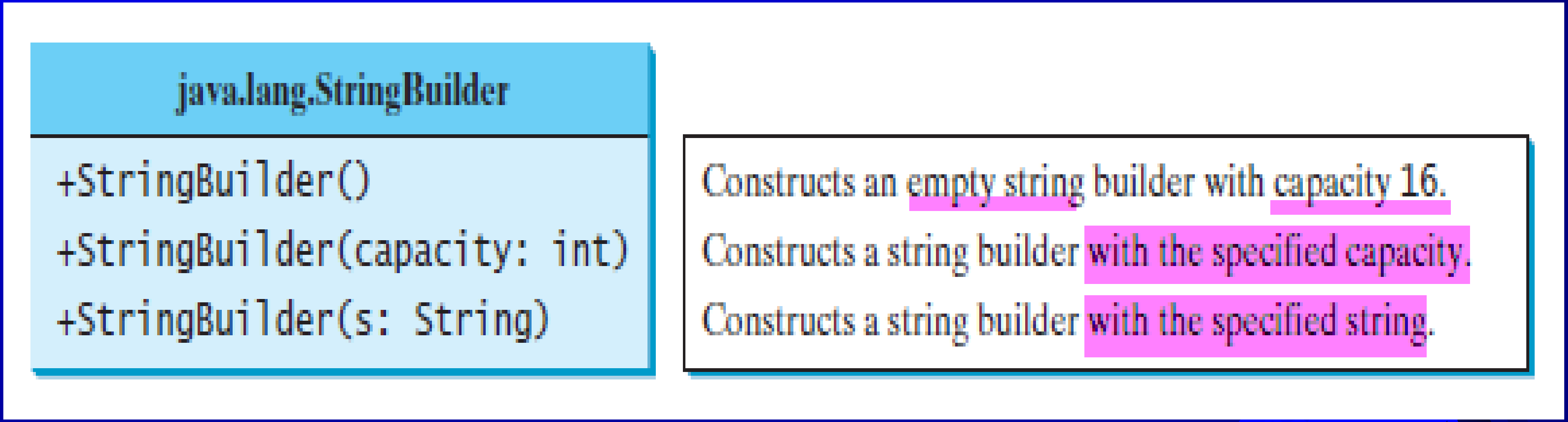
StringBuilder and StringBuffer

The **StringBuilder/StringBuffer** class

- an alternative to the String class.
  - can be used wherever a string is used.
- more flexible than String.
  - You can add, insert, or append new contents into a string buffer
  - However, the value of a String object is fixed once the string is created.

If a string does not require any change, use String rather than StringBuilder.

Three StringBuilder Constructors



java.lang.StringBuilder

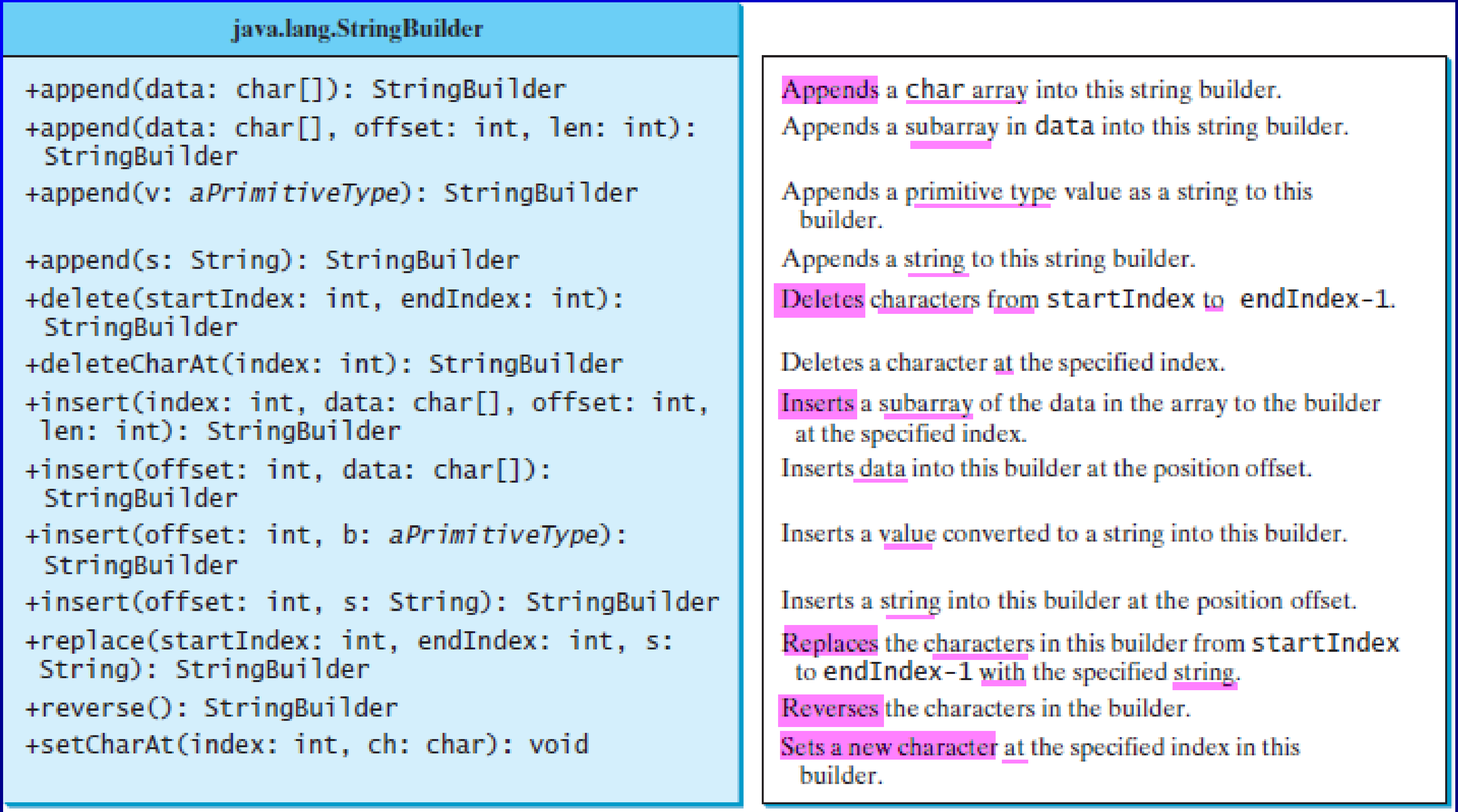
```
+StringBuilder()
+StringBuilder(capacity: int)
+StringBuilder(s: String)
```

Constructs an empty string builder with capacity 16.

Constructs a string builder with the specified capacity.

Constructs a string builder with the specified string.

Modifying Strings in the Builder

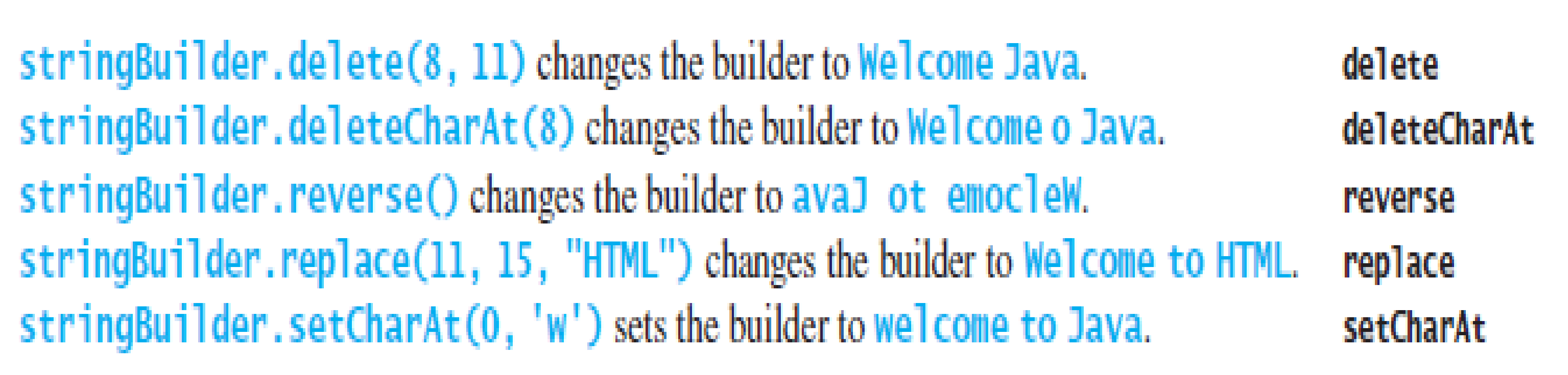


java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a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

Examples

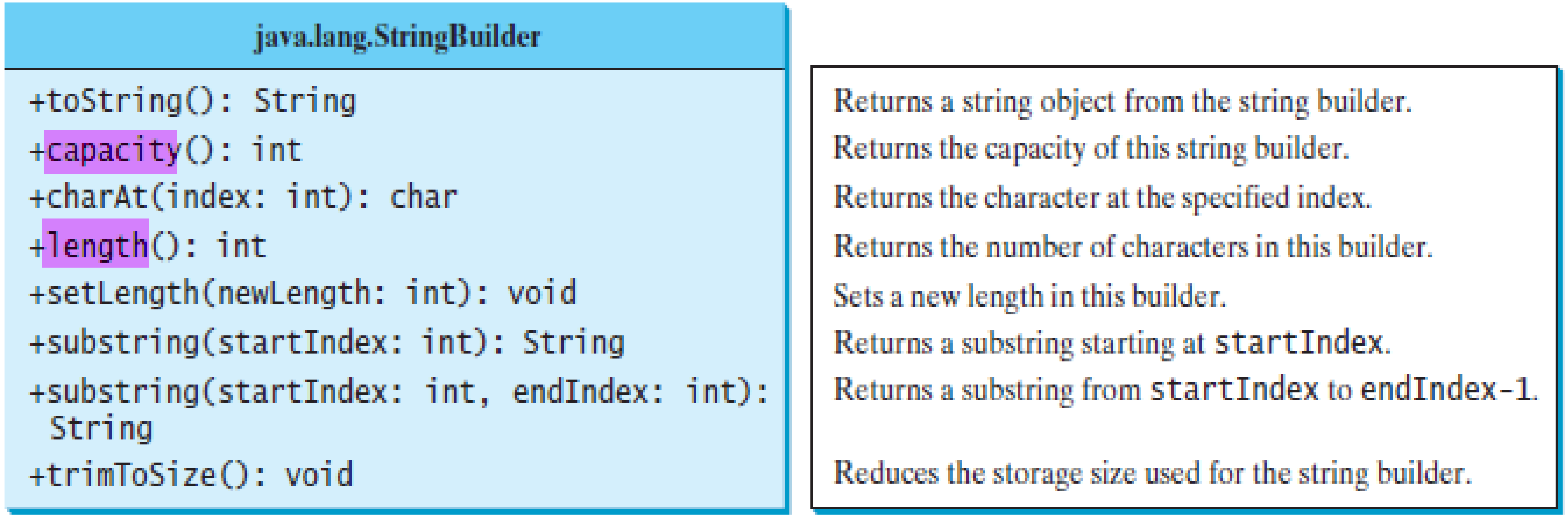
```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

- form a new string "Welcome to Java”



stringBuilder.delete(8, 11) changes the builder to Welcome Java.	delete
stringBuilder.deleteCharAt(8) changes the builder to Welcome o Java.	deleteCharAt
stringBuilder.reverse() changes the builder to avaJ ot emocleW.	reverse
stringBuilder.replace(11, 15, "HTML") changes the builder to Welcome to HTML.	replace
stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java.	setCharAt

capacity, length 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.

The StringBuilder class contains the methods for modifying string builders.

Capacity: current size of the builder

- without having to increase its size.

Length: actual size of the string stored in the builder

- The length of the string is always less than or equal to the capacity of the builder.
- The builder's capacity is automatically increased if more characters are added to exceed its capacity.
- If the capacity is too large, you will waste memory space. You can use the trimToSize() method to reduce the capacity to the actual size.