

Lecture 8

Thinking in Objects



- 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.
- The user of the class does not need to know how the class is implemented.
- The detail of implementation is encapsulated and hidden from the user.

Class implementation
is like a black box
hidden from the clients

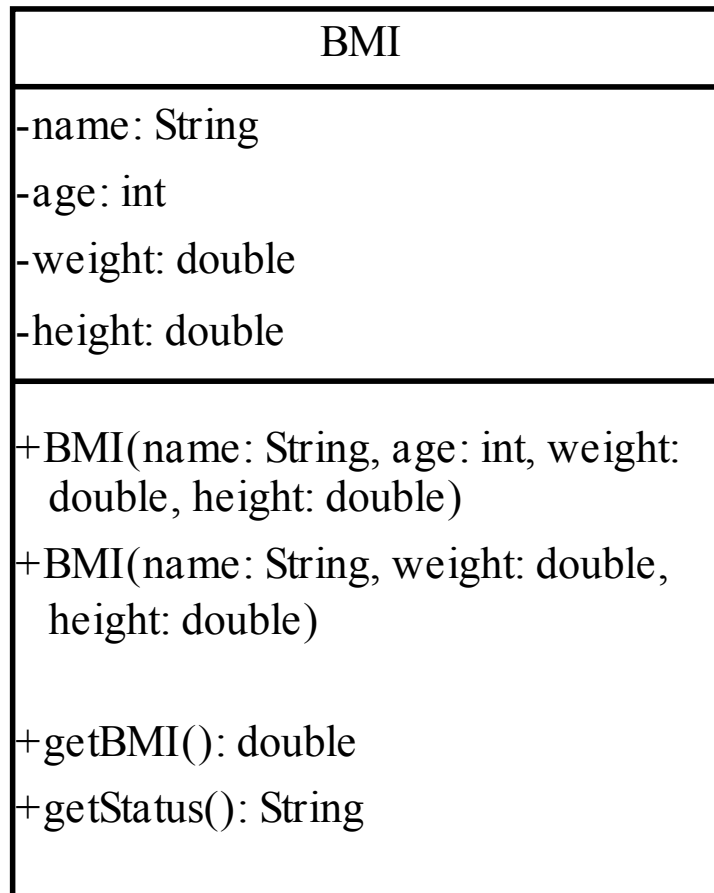
class

Class Contract (Signature
of public methods and
public constants)

Clients use the class
through the contract
of the class



| Loan | |
|---|---|
| -annualInterestRate: double | The annual interest rate of the loan (default: 2.5). |
| -numberOfYears: int | The number of years for the loan (default: 1) |
| -loanAmount: double | The loan amount (default: 1000). |
| -loanDate: Date | The date this loan was created. |
| +Loan() | Constructs a default Loan object. |
| +Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double) | Constructs a loan with specified interest rate, years, and loan amount. |
| +getAnnualInterestRate(): double | Returns the annual interest rate of this loan. |
| +getNumberOfYears(): int | Returns the number of the years of this loan. |
| +getLoanAmount(): double | Returns the amount of this loan. |
| +getLoanDate(): Date | Returns the date of the creation of this loan. |
| +setAnnualInterestRate(annualInterestRate: double): void | Sets a new annual interest rate to this loan. |
| +setNumberOfYears(numberOfYears: int): void | Sets a new number of years to this loan. |
| +setLoanAmount(loanAmount: double): void | Sets a new amount to this loan. |
| +getMonthlyPayment(): double | Returns the monthly payment of this loan. |
| +getTotalPayment(): double | Returns the total payment of this loan. |



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

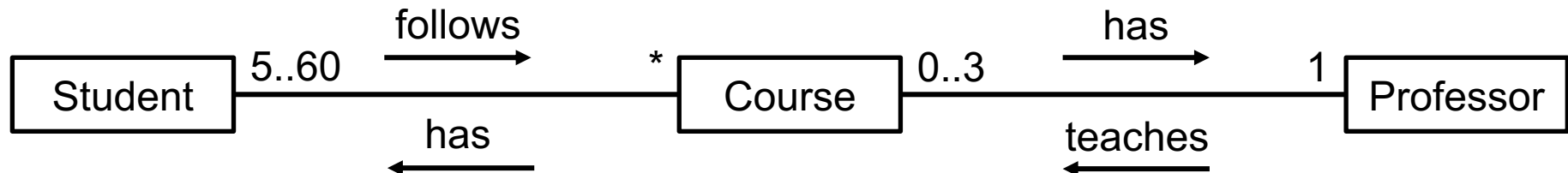
Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

- **Association**
- **Aggregation**
- **Composition**
- Inheritance (after midterm)

Association: is a general binary relationship that describes an activity between two classes.



| Course | |
|-------------------------------------|--|
| -courseName: String | |
| -students: String[] | |
| -numberOfStudents: int | |
| +Course(courseName: String) | |
| +getCourseName(): String | |
| +addStudent(student: String): void | |
| +dropStudent(student: String): void | |
| +getStudents(): String[] | |
| +getNumberOfStudents(): int | |

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

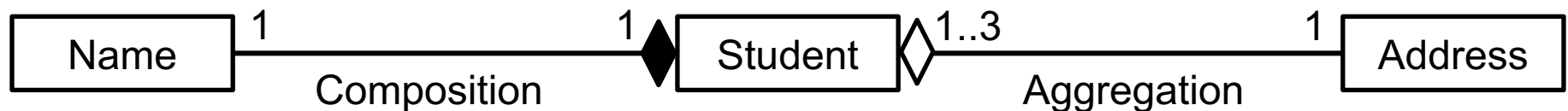
Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.



- **Aggregation** models **has-a** relationships and represents an ownership relationship between two objects.
- Composition is actually a special case of the aggregation relationship.
 - 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.



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

- Example

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

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

Aggregated classes

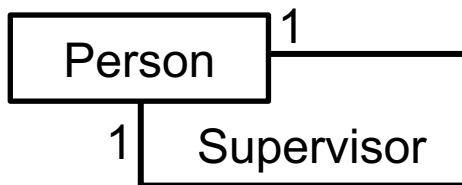
```
public class Student {  
    private Name name;  
    private Address address;  
  
    ...  
}
```

Aggregating class

- Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.



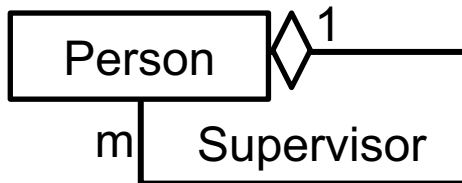
- Aggregation may exist between objects of the same class.
- For example, a person may have a supervisor.



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



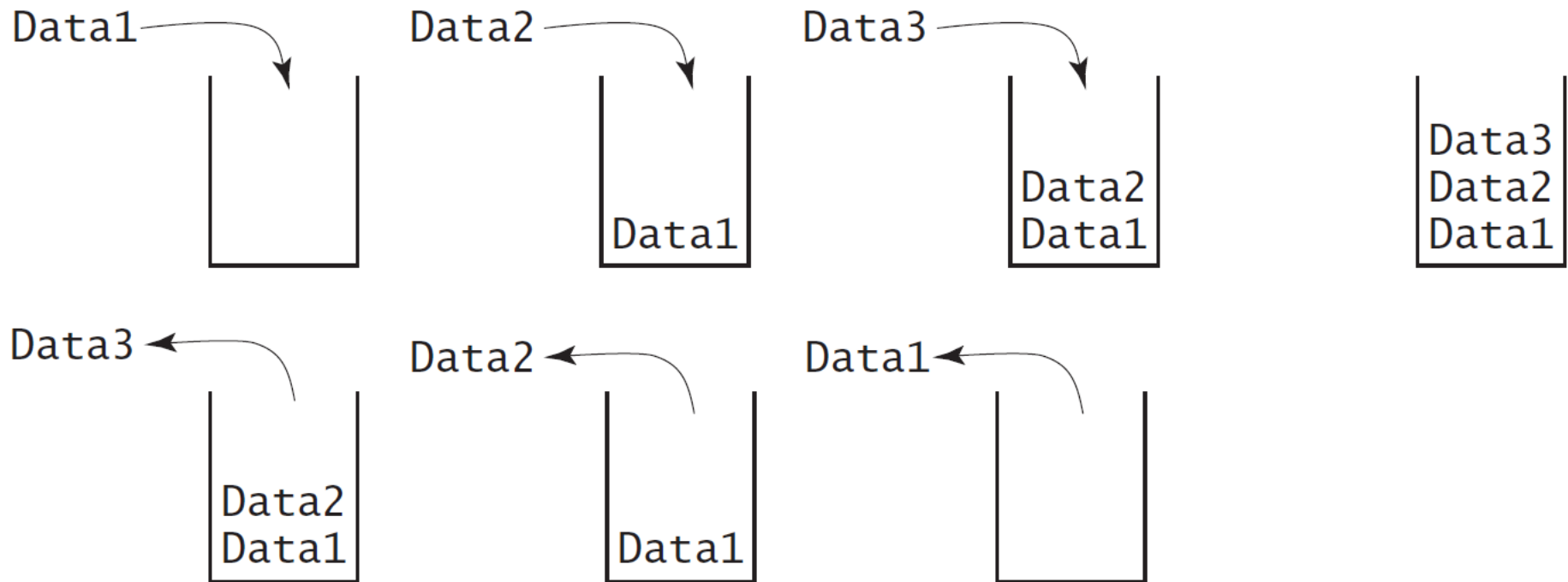
- What happens if a person has several supervisors?

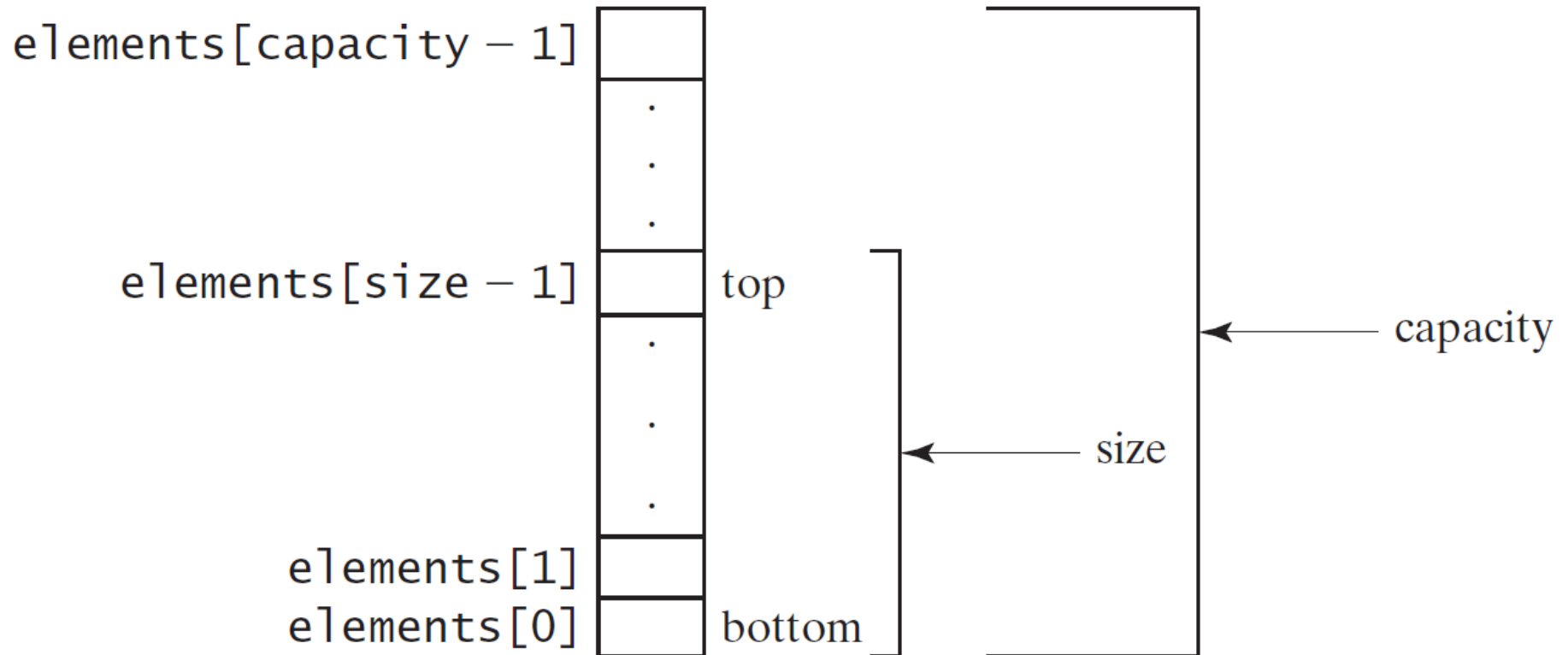


```
public class Person {
    // The type for the
    // data is the class itself
    private Person[] supervisors;
    ...
}
```



| 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): int | 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. |





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

NOTE:

- The wrapper classes do not have no-arg constructors.
- The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

**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
```


- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods



- 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:
 - `public Integer(int value)`
 - `public Integer(String s)`
 - `public Double(double value)`
 - `public Double(String s)`



- 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.

- Each numeric wrapper class implements the abstract methods:
 - `doubleValue`
 - `floatValue`
 - `intValue`
 - `longValue`
 - `shortValue`
- which are defined in the `Number` class. These methods “convert” objects into primitive type values.



- 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");`

- We 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.



- JDK 1.5 allows primitive type and wrapper classes to be converted automatically.

- For example

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

Equivalent (since JDK 1.5)

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

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

Unboxing



- If you need to compute with very large integers or high precision floating-point values, you can use
 - `BigInteger` and `BigDecimal` classes
- in the `java.math` package.
- Both are immutable.
- Both extend the `Number` class and implement the `Comparable` interface.


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

```
BigDecimal a = new BigDecimal(1.0);  
BigDecimal b = new BigDecimal(3);  
BigDecimal c = a.divide(b, 20,  
BigDecimal.ROUND_UP);  
System.out.println(c);
```

- **Constructing a String:**
 - `String message = "Welcome to Java";`
 - `String message = new String("Welcome to Java");`
 - `String s = new String();`
- **Obtaining String length and Retrieving Individual Characters in a string**
- **String Concatenation** (`concat`)
- **Substrings** (`substring(index)`, `substring(start, end)`)
- **Comparisons** (`equals`, `compareTo`)
- **String Conversions**
- **Finding a Character or a Substring in a String**
- **Conversions between Strings and Arrays**
- **Converting Characters and Numeric Values to Strings**

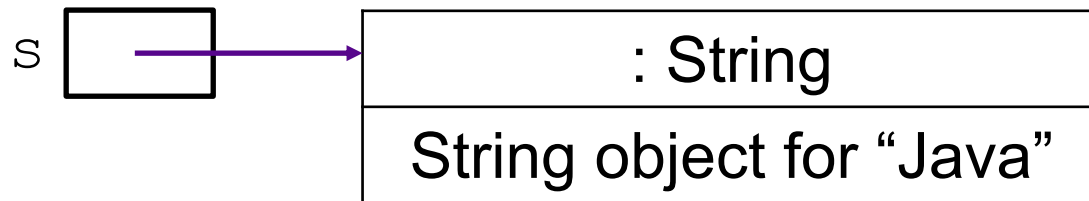
- `String newString = new String(stringLiteral);`
- `String message = new String("Welcome to Java");`
- Since strings are used frequently, Java provides a shorthand initializer for creating a string:
- `String message = "Welcome to Java";`

- 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";
```

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

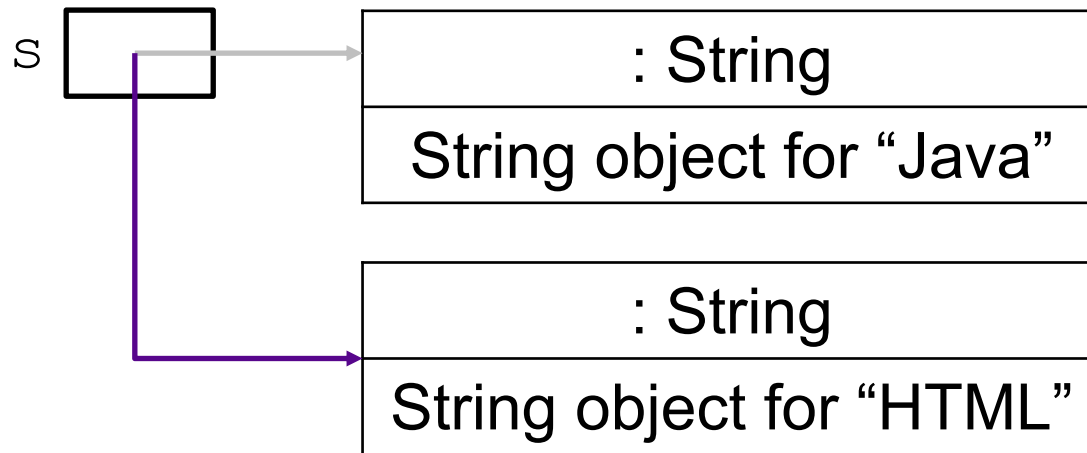
```
s = "HTML";
```



Contents cannot be changed

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

```
s = "HTML";
```



This string object is now unreferenced

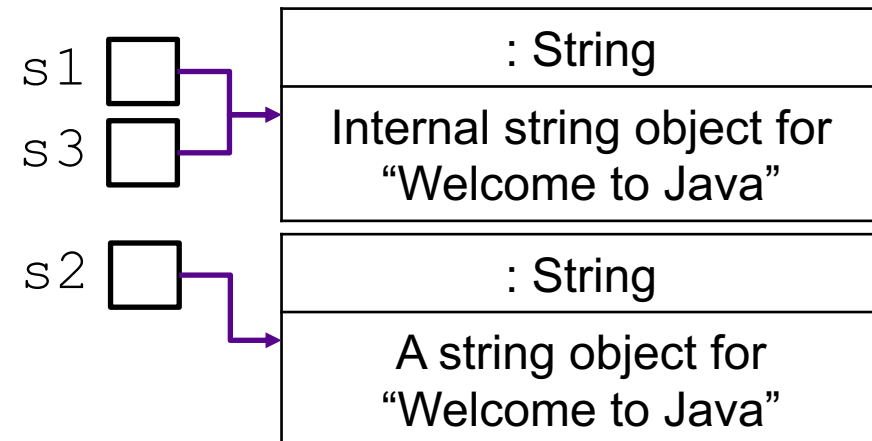
- Since strings are immutable and are frequently used, to improve efficiency and save memory,
- the JVM uses a unique instance for string literals with the same character sequence.
- Such an instance is called interned.

- 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.

```
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));
```

displays
s1 == s2 is false
s1 == s3 is true



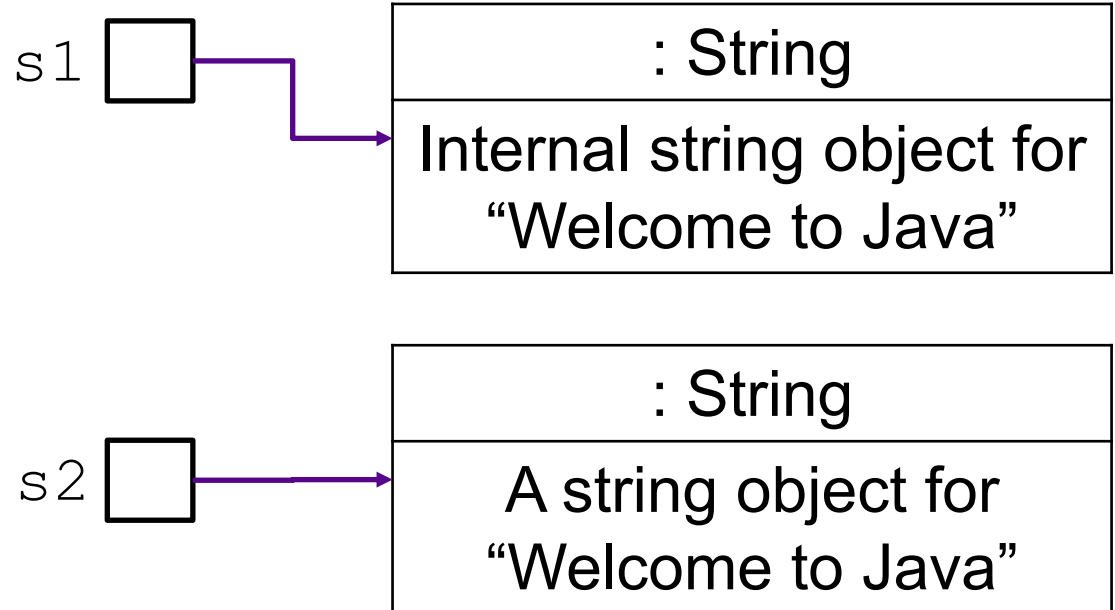

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

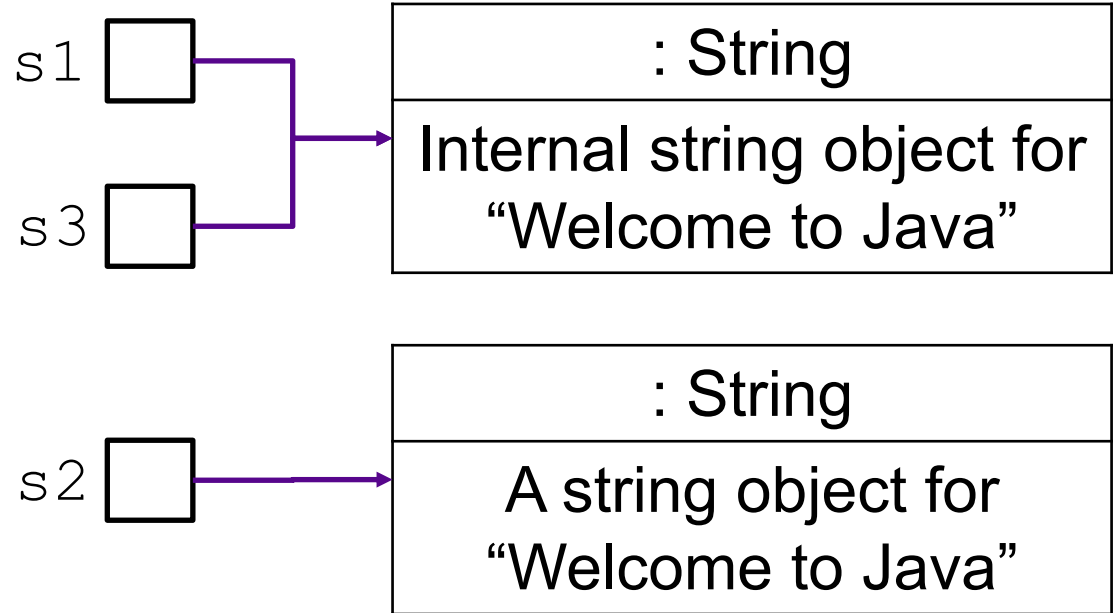
```
String s3 = "Welcome to Java";
```



```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```



```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```





java.lang.String

+replace(oldChar: char,
newChar: char): String

Returns a new string that replaces all matching character 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 replace 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`.

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

displays

Java HTML Perl



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

- 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 a string buffer, whereas the value of a `String` object is fixed once the string is created.

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: <i>aPrimitiveType</i>): 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. |
| +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: <i>aPrimitiveType</i>): 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 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.append("Java");  
StringBuilder.insert(11, "HTML and ");  
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.
```

java.lang.StringBuilder

+toString(): String

+capacity(): int

+charAt(index: int): char

+length(): int

+setLength(newLength: int): void

+substring(startIndex: int): String

+substring(startIndex: int, endIndex: int):
String

+trimToSize(): void

Returns a string object from the string builder.

Returns the capacity of this string builder.

Returns the character at the specified index.

Returns the number of characters in this builder.

Sets a new length in this builder.

Returns a substring starting at startIndex.

Returns a substring from startIndex to endIndex-1.

Reduces the storage size used for the string builder.



- 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 in this section.

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

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

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

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



- The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression.
- Example return 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`.



- 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]);
```

- A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings.
- Regular expression is a powerful tool for string manipulations.
- You can use regular expressions for matching, replacing, and splitting strings.


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

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

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

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

```
"Java is powerful".matches("Java.*")
```



| Regular Expression | Matches | Example |
|-----------------------------------|--|--|
| <code>x</code> | a specified character <code>x</code> | <code>Java</code> matches <code>Java</code> |
| <code>.</code> | any single character | <code>Java</code> matches <code>J..a</code> |
| <code>(ab cd)</code> | <code>ab</code> or <code>cd</code> | <code>ten</code> matches <code>t(en im)</code> |
| <code>[abc]</code> | <code>a</code> , <code>b</code> , or <code>c</code> | <code>Java</code> matches <code>Ja[uvw]a</code> |
| <code>[^abc]</code> | any character except <code>a</code> , <code>b</code> , or <code>c</code> | <code>Java</code> matches <code>Ja[^ars]a</code> |
| <code>[a-z]</code> | <code>a</code> through <code>z</code> | <code>Java</code> matches <code>[A-M]av[a-d]</code> |
| <code>[^a-z]</code> | any character except <code>a</code> through <code>z</code> | <code>Java</code> matches <code>Jav[^b-d]</code> |
| <code>[a-e[m-p]]</code> | <code>a</code> through <code>e</code> or <code>m</code> through <code>p</code> | <code>Java</code> matches <code>[A-G[I-M]]av[a-d]</code> |
| <code>[a-e&&[c-p]]</code> | intersection of <code>a-e</code> with <code>c-p</code> | <code>Java</code> matches <code>[A-P&&[I-M]]av[a-d]</code> |
| <code>\d</code> | a digit, same as <code>[0-9]</code> | <code>Java2</code> matches <code>"Java[\d]"</code> |
| <code>\D</code> | a non-digit | <code>\$Java</code> matches <code>"[\D][\D]ava"</code> |
| <code>\w</code> | a word character | <code>Java1</code> matches <code>"[\w]ava[\w]"</code> |
| <code>\W</code> | a non-word character | <code>\$Java</code> matches <code>"[\W][\w]ava"</code> |
| <code>\s</code> | a whitespace character | <code>"Java 2"</code> matches <code>"Java\s2"</code> |
| <code>\S</code> | a non-whitespace char | <code>Java</code> matches <code>"[\S]ava"</code> |
| <code>p*</code> | zero or more occurrences of pattern <code>p</code> | <code>aaaabb</code> matches <code>"a*bb"</code> <code>ababab</code> matches <code>"(ab)*"</code> |
| <code>p+</code> | one or more occurrences of pattern <code>p</code> | <code>a</code> matches <code>"a+b*"</code> <code>able</code> matches <code>"(ab)+.*"</code> |
| <code>p?</code> | zero or one occurrence of pattern <code>p</code> | <code>Java</code> matches <code>"J?Java"</code> <code>Java</code> matches <code>"J?ava"</code> |
| <code>p{n}</code> | exactly <code>n</code> occurrences of pattern <code>p</code> | <code>Java</code> matches <code>"Ja{1}.*"</code> <code>Java</code> does not match <code>".{2}"</code> |
| <code>p{n,}</code> | at least <code>n</code> occurrences of pattern <code>p</code> | <code>aaaa</code> matches <code>"a{1,}"</code> <code>a</code> does not match <code>"a{2,}"</code> |
| <code>p{n,m}</code> | between <code>n</code> and <code>m</code> occurrences (inclusive) | <code>aaaa</code> matches <code>"a{1,9}"</code> <code>abb</code> does not match <code>"a{2,9}bb"</code> |

java.lang.String

+matches(regex: String): boolean

Returns true if this string matches the pattern.

+replaceAll(regex: String,
replacement: String): String

Returns a new string that replaces all
matching substrings with the replacement.

+replaceFirst(regex: String,
replacement: String): String

Returns a new string that replaces the first
matching substring with the replacement.

+split(regex: String): String[]

Returns an array of strings consisting of the
substrings split by the matches.

- `String s = "Java Java
Java".replaceAll("v\\w", "wi") ;`
- `String s = "Java Java
Java".replaceFirst("v\\w", "wi") ;`
- `String[] s =
"Java1HTML2Perl".split("\\d") ;`