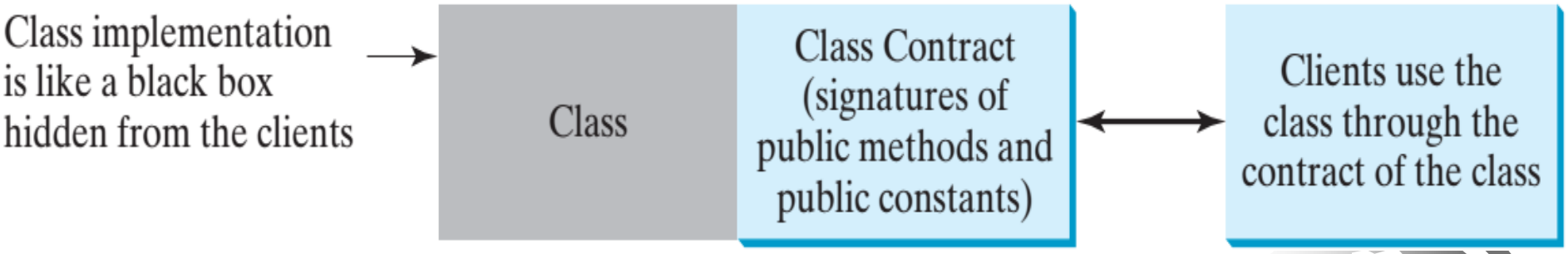
Thinking in Objects

Class Abstraction and Encapsulation

类抽象和封装

* 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.



Object-Oriented Thinking

* Classes provide more *flexibility* and *modularity* for building reusable software.

Class Relationships

* The common relationships among classes are
  - *association*, 关联
  - *aggregation*, 聚集
  - *composition*, 组合
  - *inheritance*. 继承

- This section explores association, aggregation, and composition.
- The inheritance relationship will be introduced in the next chapter.

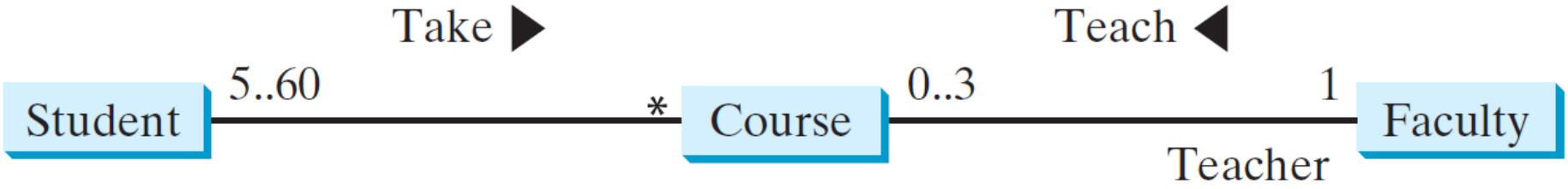
Association 关联

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

- 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.

- In Java code, you can implement associations by using data fields and methods.



Aggregation and Composition
（聚集和组合）

* Aggregation is a special form of association that represents an ownership relationship between two objects.

- Aggregation models has-a relationships.

- The owner object is called an *aggregating object*, and its class is called an *aggregating class*.

- The subject object is called an *aggregated object*, and its class is called an *aggregated class*.

* If an object is exclusively owned by an aggregating object, the relationship is referred to as a *composition*.

Class Representation

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

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

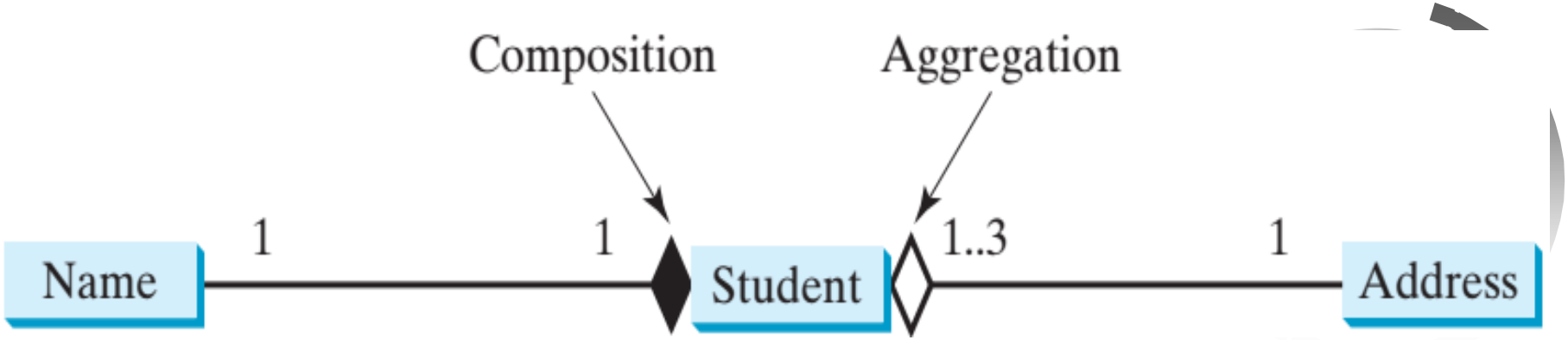
Aggregated class

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

Aggregating class

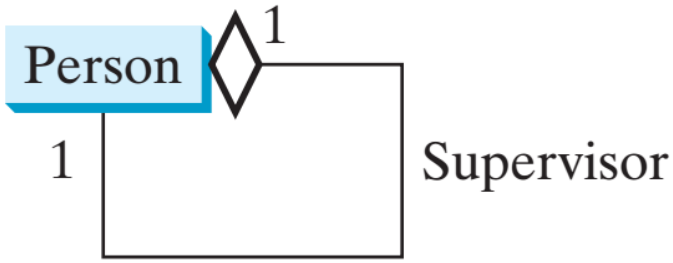
```
public class Address {
  ...
}
```

Aggregated class



Aggregation Between Same Class

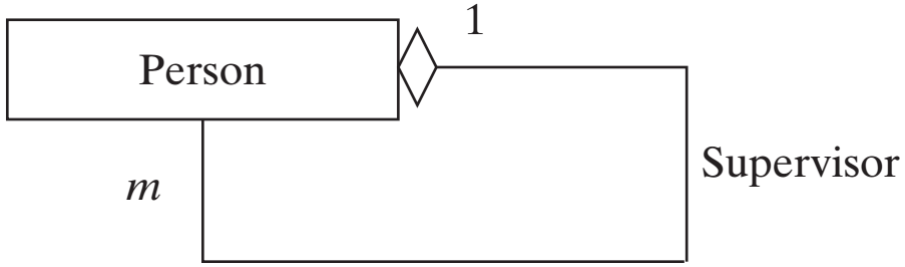
* Aggregation may exist between objects of the same class.




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

  ...
}
```

a person has a supervisor




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

A person can have several supervisors

Wrapper Classes 包装类

* Owing to performance considerations, primitive data type values are not objects in Java.

* However, many Java methods require the use of objects as arguments.

* Java offers a convenient way to wrap a primitive data type into an object.

* Java provides

  - Boolean, Character, Double, Float, Byte, Short, Integer, and Long wrapper classes in the java.lang package for primitive data types.

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 +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

The `Integer` Class and the `Double` Class

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

Numeric Wrapper Class Constructors

* The wrapper classes do not have no-arg constructors.

* The instances of all wrapper classes are immutable;

  - this means that, once the objects are created, their internal values cannot be changed.

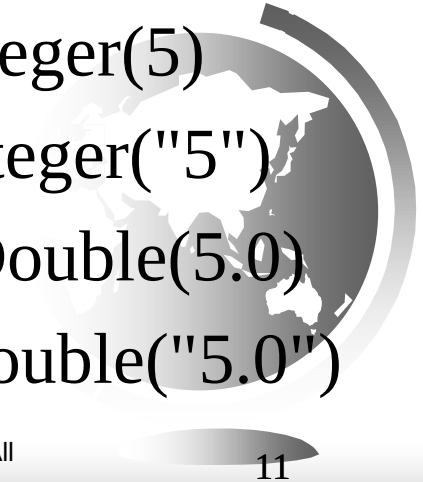
* You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value, such as

  public Integer(int value) example: new Integer(5)

  public Integer(String s) example: new Integer("5")

  public Double(double value) example: new Double(5.0)

  public Double(String s) example: 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.

Conversion Methods

* Each numeric wrapper class contains the methods doubleValue(), floatValue(), intValue(), longValue(), and shortValue() for returning a double, float, int, long, or short value for the wrapper object.

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

* The numeric wrapper classes contain the compareTo method for comparing two numbers and returns 1, 0, or -1(represent greater than, equal to, or less than).

```
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;
```

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");
```

Parsing Strings into Numbers

* Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on specified redix.

* Example:

    Integer.parseInt("11", 2) returns 3;

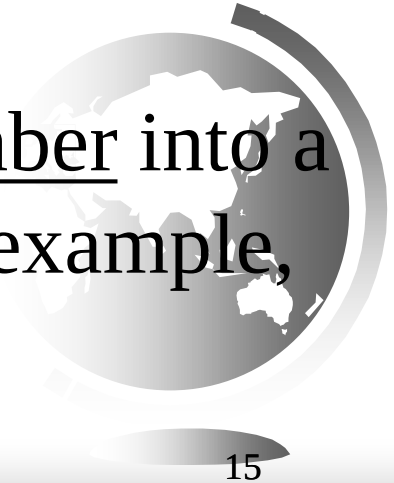
    Integer.parseInt("12", 8) returns 10;

    Integer.parseInt("13", 10) returns 13;

    Integer.parseInt("1A", 16) returns 26;

* Note that you can convert a decimal number into a hex number using the format method. For example,

    String.format("%x", 26) returns 1A;



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 intObject = new Integer (2);
```

(a)

Equivalent

```
Integer intObject = 2;
```

autoboxing

(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.

* Both are *immutable*.

* Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 92233720368547758O7 * 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);
```

The `String` Class

- Constructing a 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

Constructing Strings

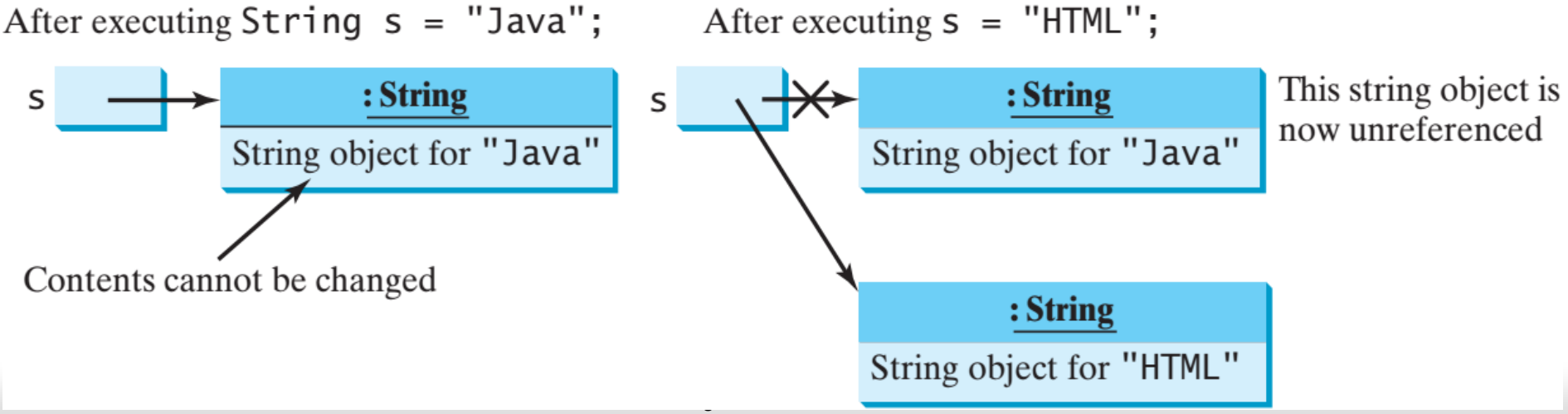
String message = new String("Welcome to Java");

shorthand: String message = "Welcome to Java";

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";



Interned Strings

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

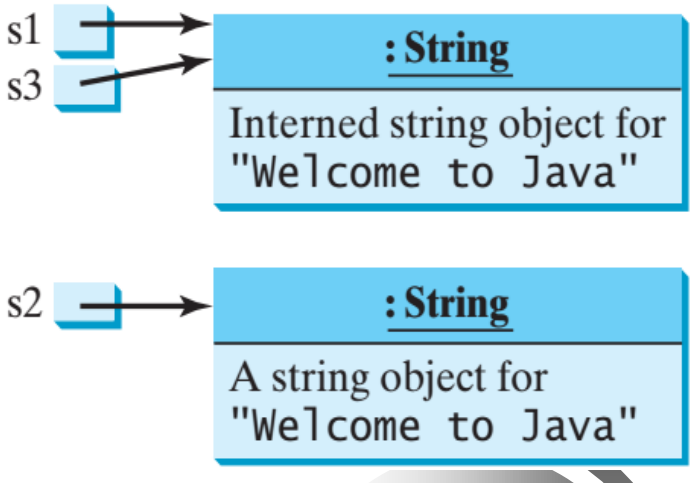
* Such an instance is called *interned*.

```
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)
```



display

s1 == s2 is false

s1 == s3 is true

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.

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".replace("el", "AB") returns a new string, WABcome.

```
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*, known as *regular expression* (abbreviated regex).

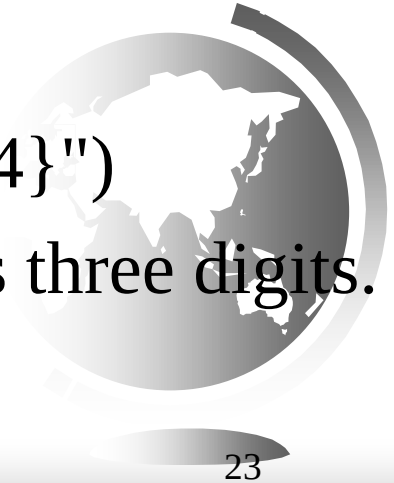
* **Matching**

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

\\ Java.* describes a string pattern that begins with *Java* followed by any zero or more characters.

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

\\d represents a single digit, and \\d{3} represents three digits.



Matching, Replacing and Splitting by Patterns

* **Replacing**

- 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

* **Splitting**

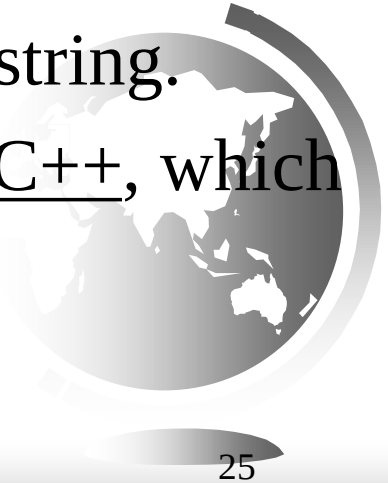
- The following statement splits the string into an array of strings delimited by some punctuation marks.

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

- In this example, the regular expression [.,:;?] specifies a pattern that matches ., ,, :, ;, or ?.

- Each character is a delimiter for splitting the string.

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



Conversion between Strings and Arrays

* Convert a string into an array of characters

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

* Copy a substring of the string into a character array

```
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 an array of characters into a string

```
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 contains the static methods for creating strings from primitive type values.

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.

- For example, to convert 5.44 to a string, use String.valueOf(5.44), which returns a string of the characters '5', '.', '4', and '4'.

Formatting Strings

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

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

```
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.

- 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.

- The 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` 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

* You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string 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.

Examples

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

stringBuilder.insert(11, "HTML and "); // Welcome to HTML and Java.
stringBuilder.delete(8, 11)                 // Welcome Java.
stringBuilder.deleteCharAt(8)               // Welcome o Java.
stringBuilder.reverse()                     //avaJ ot emocleW.
stringBuilder.replace(11, 15, "HTML")   //Welcome to HTML.
stringBuilder.setCharAt(0, 'w')              // welcome to Java.
```

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.

Appendix H

Regular Expressions

Regular Expression Syntax

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