BIRZEIT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Instructor :Murad Njoum
Office : Masri322

Chapter 10 Thinking in Objects and Strings

Simple Methods for **String** Objects

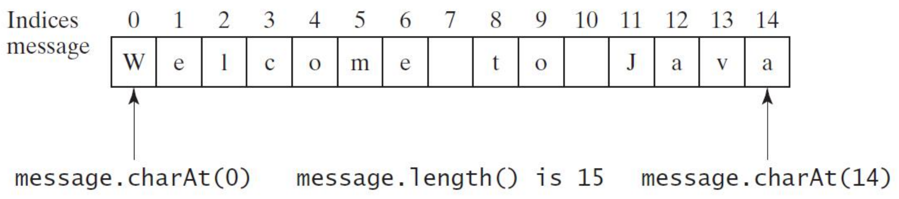
Strings are objects in Java. The methods in the preceding table can only be invoked from a specific string instance. For this reason, these methods are called ***instance methods***. A non-instance method is called a *static method*. A static method can be invoked without using an object. All the methods defined in the **Math** class are static methods. They are not tied to a specific object instance. The syntax to invoke an instance method is

referenceVariable.methodName(arguments).

Getting String Length

```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is "
  + message.length());
```

Getting Characters from a String



```
String message = "Welcome to Java";
System.out.println("The first character in message is "
  + message.charAt(0));
```

"Welcome".toLowerCase() returns a new string, welcome.

"Welcome".toUpperCase() returns a new string, WELCOME.

" Welcome ".trim() returns a new string, Welcome.

String Concatenation

```
String s3 = s1.concat(s2); or String s3 = s1 + s2;

// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

Reading a String from the Console

```
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

Reading a Character from the Console

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

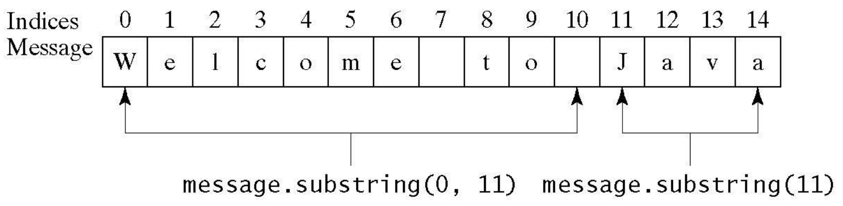
Comparing Strings

Method	Description
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.

```
import java.util.Scanner;
public class OrderTwoCities {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
 // Prompt the user to enter two cities
System.out.print(‘Enter the first city: ‘);
String city1 = input.nextLine(); System.out.print(‘Enter the second city: ‘);
String city2 = input.nextLine();
if (city1.compareTo(city2) < 0)
 System.out.println(‘The cities in alphabetical order are ‘ + city1 + ‘ ‘ + city2);
 else System.out.println(‘The cities in alphabetical order are ‘ + city2 + ‘ ‘ + city1);
 }
}
```

Obtaining Substrings

Method	Description
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex – 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.

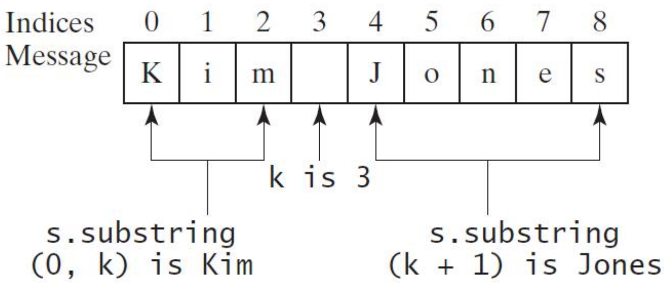


Finding a Character or a Substring in a String

Method	Description
`indexOf(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	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

```
int k = s.indexOf(' ');
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
```



Conversion between Strings and Numbers

```
int intValue = Integer.parseInt(intString);
double doubleValue = Double.parseDouble(doubleString);

String s = number + "";
```

int intValue = Integer.parseInt("10");

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**. 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'.

String.valueOf(tokens[0]).

Formatting Output

Use the printf statement.

System.out.printf(format, items);

Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

Frequently-Used Specifiers

Specifier	Output	Example
%b	a boolean value	true or false
%c	a character	'a'
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	"Java is cool"

```
                                                                 items
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);


display            count is 5 and amount is 45.560000
```

FormatDemo

The example gives a program that uses **printf** to display a table.

```
public class FormatDemo {
  public static void main(String[] args)
   { // Display the header of the table
    System.out.printf(‘%-10s%-10s%-10s%-10s%-10s\n‘, ‘Degrees‘, ‘Radians‘, ‘Sine‘,
       ‘Cosine‘, ‘Tangent‘ );
  // Display values for 30 degrees

int degrees = 30;
double radians = Math.toRadians(degrees);

System.out.printf(‘%-10d%-10.4f%-10.4f%-10.4f%-10.4f\n‘, degrees, radians,
        Math.sin(radians), Math.cos(radians), Math.tan(radians));
 // Display values for 60 degrees degrees = 60; radians = Math.toRadians(degrees);
System.out.printf(‘%-10d%-10.4f%-10.4f%-10.4f%-10.4f\n‘, degrees, radians,
          Math.sin(radians), Math.cos(radians), Math.tan(radians));
  }
 }
```

The String Class

- ❑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

Constructing 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";

Strings Are Immutable

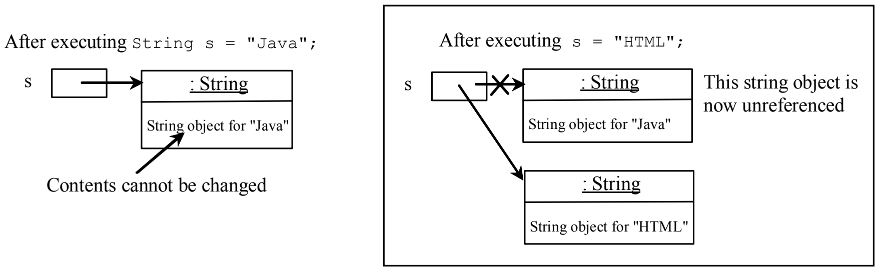
A String object is **immutable**; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java"; //contents cannot change
s = "HTML";
```

Trace Code

String s = "Java";

s = "HTML";



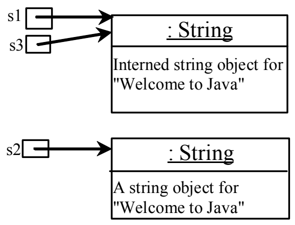
After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

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 interned (مُدرّب). 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"

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.

display

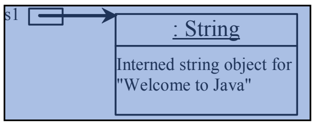
s1 == s2 is false

s1 == s3 is true

```
String str1= new String("Java");
        String str2 = "Java";
        System.out.println(str1.compareTo(str2)==0);
        true
```

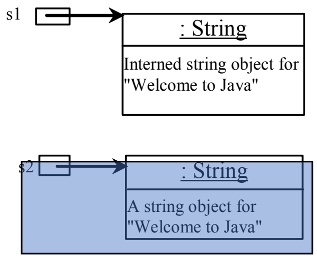
Trace Code

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



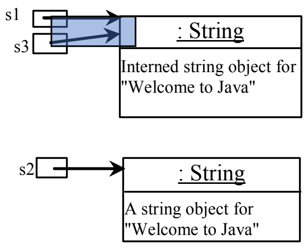
Trace Code

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


s1
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

Trace Code

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


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

Replacing and Splitting Strings

java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching 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.

Examples

"Welcome".replace('e', 'A') returns a new string, WAlcomA.

"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replaceAll("e", "AB") returns a new string, WABlcomAB.

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

"Welcomel".replaceAll("el", "AB") returns a new string, WABcomAB.

Splitting a String

```
String str="Java#HTML#Perl";

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

displays

Java HTML Perl

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

StringBuilderis same as the StringBuffer , that is it stores the object in heap and it can also be modified .
The main difference between the StringBuffer and StringBuilder is thatStringBuilder is also not thread safe.

each method in StringBuffer is synchronizedthat is StringBuffer is thread safe. Due to this it does not allow two threads to simultaneously access the same method .

`StringBuilder` 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.
+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 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

```
public class Driver {

    public static void main(String[] args) {
        StringBuilder stringBuilder = new    StringBuilder("Welcome Java");

        stringBuilder.append(" Comp231");
        System.out.println(stringBuilder);


        stringBuilder.insert(12, " and HTML");
        System.out.println(stringBuilder);

        stringBuilder.delete(8, 22);
        System.out.println(stringBuilder);

        stringBuilder.replace(8, 15, "HTML");
        System.out.println(stringBuilder);

        stringBuilder.deleteCharAt(8) ;
        System.out.println(stringBuilder);

        stringBuilder.setCharAt(0, 'w') ;
        System.out.println(stringBuilder);

        stringBuilder.reverse() ;
        System.out.println(stringBuilder);

    }

}
```

```
Welcome Java Comp231
Welcome Java and HTML
Comp231
Welcome Comp231
Welcome HTML
Welcome TML
welcome TML
LMT emoclew
```

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. **default capacity = 16**
+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.

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.

```
 import java.util.Scanner;
public class PalindromeIgnoreNonAlphanumeric {
  /** 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();

    // Display result
    System.out.println('Ignoring non-alphanumeric characters, \nis '
      + s + ' a palindrome? ' + isPalindrome(s));
  }

  /** Return true if a string is a palindrome */
  public static boolean isPalindrome(String s) {
    // Create a new string by eliminating non-alphanumeric chars
    String s1 = filter(s);

    // Create a new string that is the reversal of s1
    String s2 = reverse(s1);

    // Compare if the reversal is the same as the original string
    return s2.equals(s1); }
```

```
/** Create a new string by eliminating non-alphanumeric chars */
  public static String filter(String s) {
    // Create a string builder
    StringBuilder stringBuilder = new StringBuilder();

    // Examine each char in the string to skip alphanumeric char
    for (int i = 0; i < s.length(); i++) {
      if (Character.isLetterOrDigit(s.charAt(i))) {
        stringBuilder.append(s.charAt(i));
      }
    }

    // Return a new filtered string
    return stringBuilder.toString();
  }

  /** Create a new string by reversing a specified string */
  public static String reverse(String s) {
    StringBuilder stringBuilder = new StringBuilder(s);
    stringBuilder.reverse(); // Invoke reverse in StringBuilder
    return stringBuilder.toString();  }
}
```

Regular Expressions

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.

Matching Strings

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

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

Regular Expression Syntax

"Java".matches("J..a");

"Java".matches("J(av|ba)a");

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"

Regular Expression

abc exactly this sequence of three letter

[abc] any *one* of the letters a, b, or c

[^abc] any character *except* one of the letters a, b, or c
(immediately within an open bracket, **^** mean
“not,” but anywhere else it just means the
character **^**)

[a-z] any *one* character from a through z, inclusive

[a-zA-Z0-9] any *one* letter or digit

Regular Expression

If one pattern is followed by another, the two patterns must match consecutively

- For example, [A-Za-z]+[0-9] will match one or more letters immediately followed by one digit
- The vertical bar, |, is used to separate alternatives
- For example, the pattern abc|xyz will match either abc or xyz

X?	optional, X occurs once or not at all
X*	X occurs zero or more times
X+	X occurs one or more times
X{n}	X occurs exactly n times
X{n,}	X occurs n or more times
X{n, m}	X occurs at least n but not more than m times

Regular Expression

.	any one character except a line terminator
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\x0B\f\r]
\S	a non-whitespace character: [^\s]
\w	a word character: [a-zA-Z0-9]
\W	a non-word character: [^\w]
^	the beginning of a line
$	the end of a line

```
System.out.println(‘Java2‘.matches(‘Java[\\d]‘));
System.out.println(‘%Java2‘.matches(‘[\\D]Java[\\d]‘));
System.out.println(‘Javavva‘.matches(‘^J.*a%‘));
System.out.println(‘%Java1‘.matches(‘[\\W]Java[\\w]‘));
```

Regular Expression

```
String str = new String(‘Welcome student comp231 to Java course‘ );
String s1 = new String(‘Welcome‘ );
String s2 = new String(‘Welcome‘ );
String s3 = new String(‘WE1come‘ );

System.out.println(str.matches(‘Welcome‘) + ‘ ‘ + str.matches(‘Welcome.*‘));//1
System.out.println(s1.matches(‘W[wce]lcome‘) + ‘ ‘ + s1.matches(‘W[xza]lcome‘));//2
System.out.println(s1.matches(‘W[^wce]lcome‘) + ‘ ‘ + s1.matches(‘W[^xza]lcome‘));//3
System.out.println(s2.matches(‘W[a-c]lcome‘) + ‘ ‘ + s2.matches(‘W[a-gA-G0-9]lcome‘));//4
System.out.println(s3.matches(‘W[A-Za-z]+[0-9]come‘) + ‘ ‘ + s3.matches(‘W[a-gA-G0-9]lcome‘));
System.out.println(s2.matches(‘W(el|al )come‘));//6

String s4 = new String(‘Wel‘ );
System.out.println(s4.matches(‘Wel ?‘) + ‘ ‘ + s4.matches(‘We ?‘));// 7
System.out.println(s4.matches(‘Welc*‘) + ‘ ‘ + s4.matches(‘We*‘));// 8
System.out.println(s4.matches(‘Wel+‘) + ‘ ‘ + s4.matches(‘We+‘));// 9

String s5 = new String(‘Wel‘ );
System.out.println(s5.matches(‘Wel{1}‘) + ‘ ‘ + s5.matches(‘Wel{1,}‘) + ‘ ‘ + s5.matches(‘Wel{1,2}‘));//10
```

1.false true
2.true false
3.false true
4.false true
5.true false
6.true
7.true false
8.true false
9. true false
10.true true true

Replacing and Splitting Strings

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.

Examples

```
String s = "Java Java Java".replaceAll("v\\w", "wi") ;
```

Jawi Jawi Jawi

```
String s = "Java Java Java".replaceFirst("v\\w", "wi") ;
```

Jawi Java Java

```
String[] s = "Java1HTML2Perl".split("\\d");
```

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 in this section. Please refer to Supplement III.F, “Regular Expressions,” for further studies.

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

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

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.

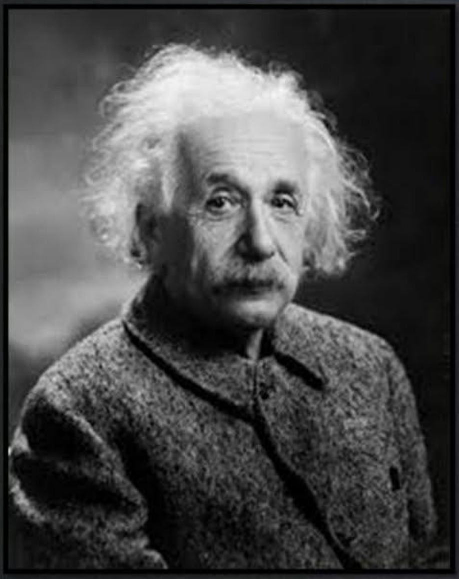
Matching, Replacing and Splitting by Patterns

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]);
```

```
Java
C
C#
C++
```

بالعلم والأخلاق
ترتقي الأمم



"I am thankful for all of those who said NO to me. It's because of them I'm doing it myself"

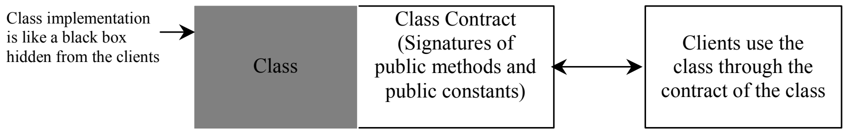
" أنا ممتن لجميع أولئك الذين قالوا لي لا.
لأنني بسببهم فعلتها بنفسي "

Thinking in Objects

You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.

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. 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.

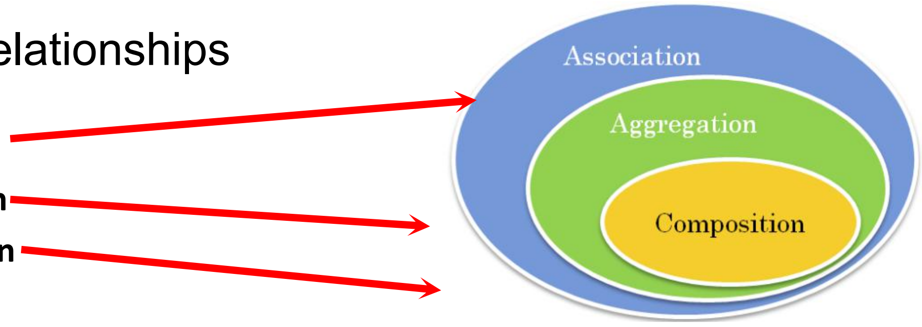

Class implementation is like a black box hidden from the clients
Class
Class Contract (Signatures of public methods and public constants)
Clients use the class through the contract of the class

Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 **using the object-oriented approach**. From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Class Relationships

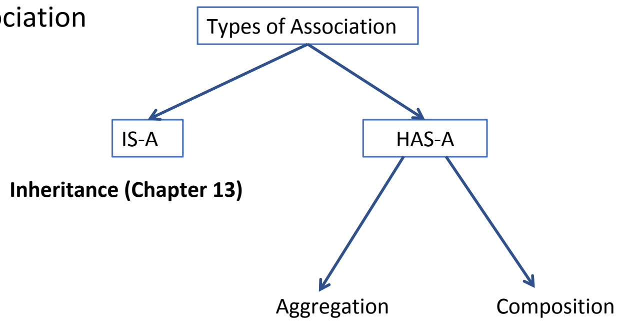
Association
Aggregation
Composition



Association in Java is a connection between two separate classes that is set up through their objects. Although, Java association can balance, one-to-one, one-to-many, and many-to-many relationships. It defines the multiplicity between objects.

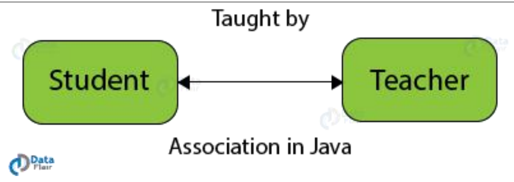
There are two types of Association

- ❖ **Aggregation**
- ❖ **Composition**



Is-A Relationship in Java

In Java, an **Is-A relationship depends on inheritance**. Further inheritance is of **two types, class inheritance and interface inheritance**. It is used for code reusability in Java.
For example, a Potato is a vegetable,
a Bus is a vehicle,
a Bulb is an electronic device and so on.
One of the properties of inheritance is that inheritance is **unidirectional** in nature. Like we can say that a **house is a building**. **But not all buildings are houses**. We can easily determine an Is-A relationship in Java. When there is an **extends** or implement keyword in the class declaration in Java, then the specific class is said to be following the Is-A relationship.



For example, if we talk about the association between a teacher and a student, multiple students can associateشريك with a single teacher and a single student is also associated with multiple teachers **but both can be created or deleted independently**. So, when a teacher leaves the school, we don't need to remove any students, and when a student leaves the school, we don't need to remove any teachers.

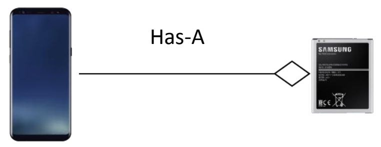
So, in the above example, the teacher **has many students and vice versa**, connectes to various objects. Thus, we can say the association in Java follows a **many-to-many** relationship.

```
class Teacher
{
  private String name;
  Teacher(String name)
  {
    this.name = name;
  }
  public String getTeacherName()
  {
    return this.name;
  }
}


class Student
{
  private String name;
  Student(String name)
  {
    this.name = name;
  }
  public String getStudentName()
  {
    return this.name;
  }
}
```

```
class AssociationDriver
{
  public static void main (String[] args)
  {
    Teacher teacherObj = new Teacher("Dr. Ahmad");
    Student studentObj = new Student("Renad");
    System.out.println(studentObj.getStudentName() +
        " is Student of " + teacherObj.getTeacherName());
  }
}
```

Aggregation (Aggregation follows a one-to-one relationship.)

- It represents the **Has-A** relationship.
- Aggregation in Java follows a **one-way or one-to-one** relationship.
- **Ending one entity won't affect another, both can be present independently.**

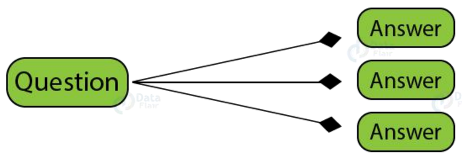


Aggregation in Java

Let's take the example of a mobile phone and a battery. A single battery can belong to a mobile phone, but if the mobile phone stops working, and we delete it from our database. The phone battery will not be deleted because it may still be functional. So in aggregation, while there is ownership, objects have their own lifecycle.

Composition :(The Composition follows a one-to-many relationship.)

- Suppose if we take an example of the relationship between questions and answers. Single questions can have multiple answers, but multiple answers can not have multiple questions. If **we delete questions**, answers will **automatically be deleted**. In this the entities are dependent.

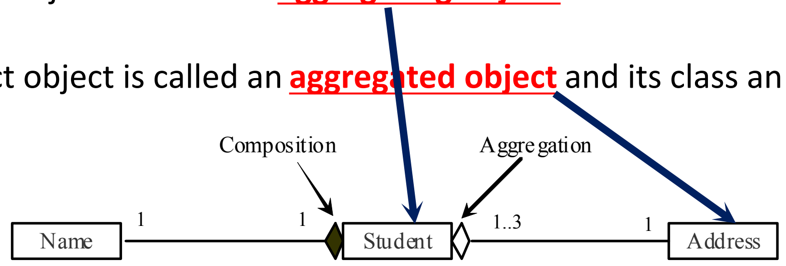


Composition in Java

Object Composition

Composition (consist of) is actually a special case of the

- aggregation (combination) relationship.
- 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.



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

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

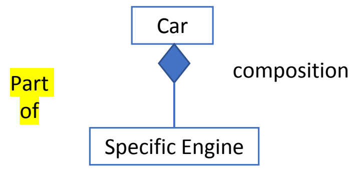
```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

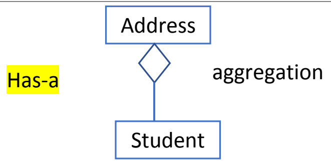
```
public class Address {
  ...
}
```

Aggregated class   Aggregating class   Aggregated class

1. Composition is a specialized form of aggregation in which if the parent object is destroyed, the child objects would cease to exist.

2. Aggregation is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well





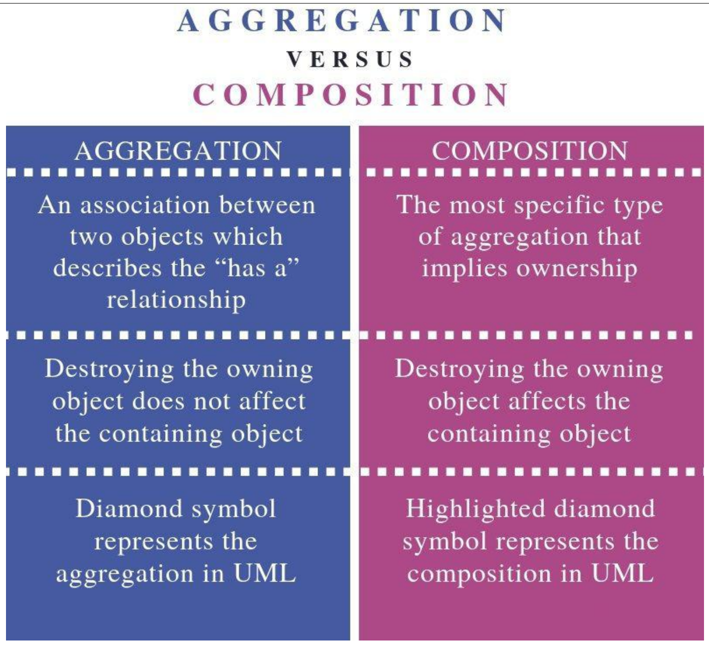
```
class Car {
private final Engine engine;
 Car( ){
    engine=new Engine( );
    }//final initialized once
}

 class Engine {
 private String type;
 }
…
Car car=new Car( );
…
```

```
class Student {
private Address address;
 Student(Address addr){
    address=addr;
    }
}
   class Address {
   String city;
   String state;
    Address(String city, String state){
       this.city=city; this.state=state;
       }
   }
      …
      Student student=new Student( );
      …
```

- **Create instance**:

(engine automatically created once), student has passed parameters from other methods

- **Delete instance**: delete car instance ,automatically engine instance deleted and can't passed to other car instance, but if class student deleted then address can be passed to other students

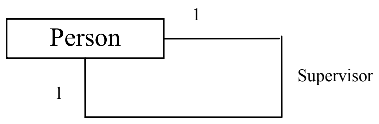


AGGREGATION
VERSUS
COMPOSITION

AGGREGATION	COMPOSITION
An association between two objects which describes the "has a" relationship	The most specific type of aggregation that implies ownership
Destroying the owning object does not affect the containing object	Destroying the owning object affects the containing object
Diamond symbol represents the aggregation in UML	Highlighted diamond symbol represents the composition in UML

Aggregation Between Same Class

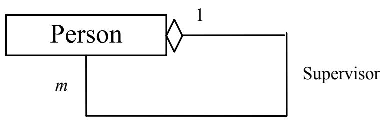
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.



```
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?

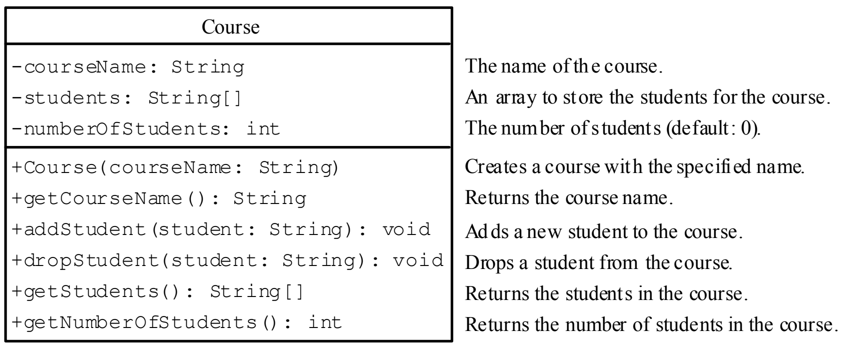


```
public class Person {
  ...
  private Person[] supervisors;
}
```

Overloading Constructors

- If you create a class from which you instantiate objects, Java automatically provides a constructor
- But, if you create your own constructor, the automatically created constructor no longer exists
- As with other methods, you can overload constructors
  - Overloading constructors provides a way to create objects with or without initial arguments, as needed

Example: The Course Class



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 in the course.
+getNumberOfStudents(): int	Returns the number of students in the course.

Course TestCourse Run

```
public class Course {
  private String courseName;
  private String[] students = new String[4];
  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) {
for (int i = 0; i < numberOfStudents; i++) {
    if (students[i].equals(student)) {
    // Move students[i + 1] to students[i], etc.
   for (int k = i + 1; k < numberOfStudents; k++) {
      students[k - 1] = students[k];
     }
     numberOfStudents--;
     break;}
}}
}
```

Wrapper Classes

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

NOTE:

(1) The wrapper classes **do not have <u>no-arg</u> constructors**.

(2) 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

java.lang.Integer	java.lang.Double
-value: int	-value: double
+MAX_VALUE: int	+MAX_VALUE: double
+MIN_VALUE: int	+MIN_VALUE: double
+Integer(value: int)	+Double(value: double)
+Integer(s: String)	+Double(s: String)
+byteValue(): byte	+byteValue(): byte
+shortValue(): short	+shortValue(): short
+intValue(): int	+intValue(): int
+longVlaue(): long	+longVlaue(): long
+floatValue(): float	+floatValue(): float
+doubleValue():double	+doubleValue():double
+compareTo(o: Integer): int	+compareTo(o: Double): int
+toString(): String	+toString(): String
+valueOf(s: String): Integer	+valueOf(s: String): Double
+valueOf(s: String, radix: int): Integer	+valueOf(s: String, radix: int): Double
+parseInt(s: String): int	+parseDouble(s: String): double
+parseInt(s: String, radix: int): int	+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

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**)

```
Integer ints=new Integer(10);

System.out.print(ints.floatValue()); // 10.0
```

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.
- ❖ The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45),

and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

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

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.

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

(a)

Equivalent

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

(b)

New JDK 1.5 boxing

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

Unboxing

Vis versa is also true.

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

LargeFactorial Run

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

```
0.33333333333333333334
```

```
package test;

import java.util.Scanner;
import java.math.*;

public class LargeFactorial {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print('Enter an integer: ' );
    int n = input.nextInt( );
    System.out.println(n + ' ! is \n' + factorial(n));
 input.close( );
}

  public static BigInteger factorial(long n) {
    BigInteger result = BigInteger.ONE; // Assign 1 to result
    for (int i = 1; i <= n; i++) // Multiply each i
      result = result.multiply(BigInteger.valueOf(i));

    return result;
  }
}
```