FAKULTAS
ILMU
KOMPUTER

Thinking in Objects

Dasar – Dasar Pemrograman 2

Dinial Utami Nurul Qomariah

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

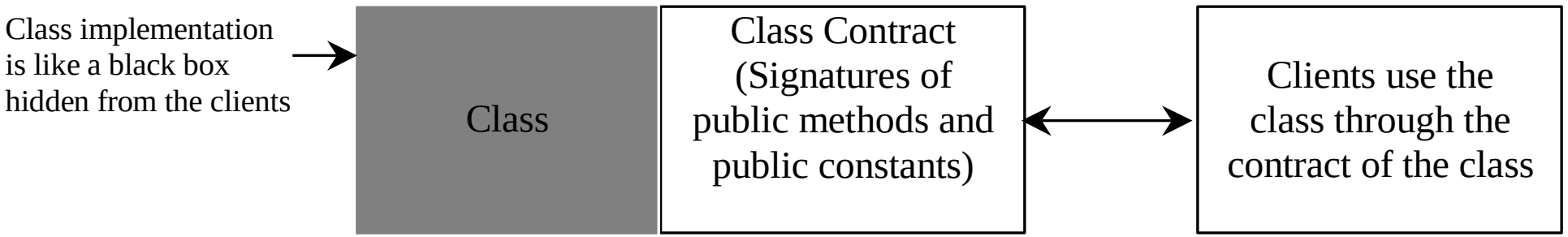
FAKULTAS ILMU KOMPUTER

Credits

- Liang, Introduction to Java Programming, 11th Edition, Ch. 1
- Downey & Mayfield, Think Java: How to Think Like a Computer Scientist, Ch. 1
- Slide Kuliah Dasar-Dasar Pemrograman 2 Semester Genap 2019/2020

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**.



Example : Designing the Loan 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.

- Dalam class diagram,
- tanda – merepresentasikan private member,
- tanda + merepresentasikan public member.

https://liveexample.pearsoncmg.com/html/Loan.html

https://liveexample.pearsoncmg.com/html/TestLoanClass.html

Example : The BMI Class

BMI
-name: String -age: int -weight: double -height: double
+BMI(name: String, age: int, weight: double, height: double) +BMI(name: String, weight: double, height: double) +getBMI(): double +getStatus(): String

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.)

https://liveexample.pearsoncmg.com/html/BMI.html

https://liveexample.pearsoncmg.com/html/UseBMIClass.html

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.

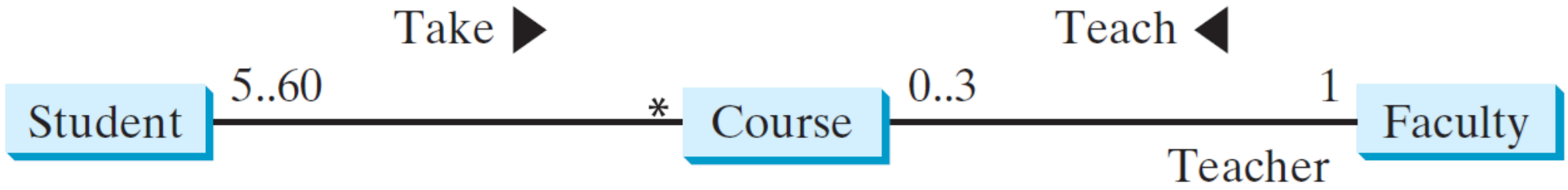
https://liveexample.pearsoncmg.com/html/Course.html

https://liveexample.pearsoncmg.com/html/TestCourse.html

Interaksi Antar Object

Object Composition

- Composition is actually a special case of the aggregation 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**.

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

Aggregated class

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

  ...
}
```

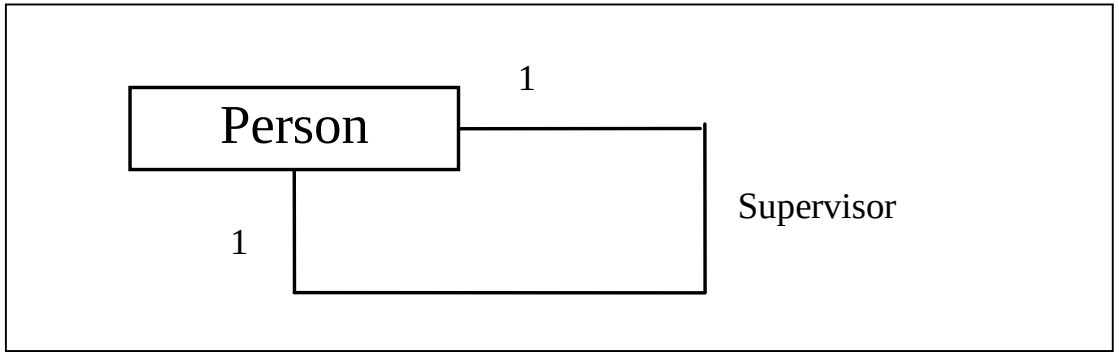
Aggregating class

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

Aggregated class

Aggregation Between Same Class

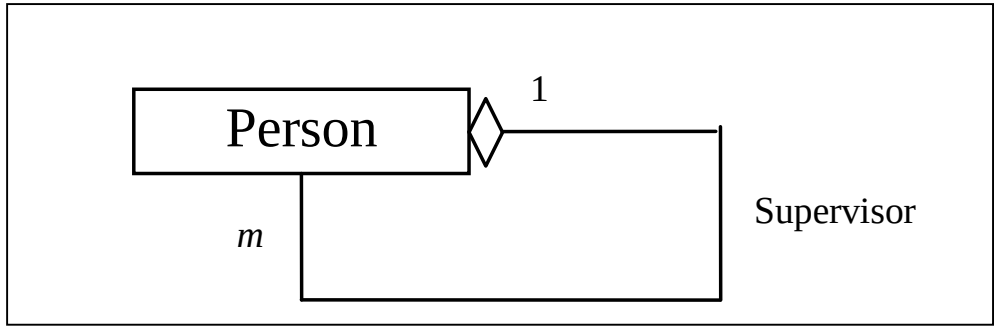
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.




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

Aggregation Between Same Class

What happens if a person has several supervisors?




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

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.

Composition

Composition a **special case** from agregation

Composition : every car has an Machine

Aggregation : car may have or not a Passengers

Wrapper Classes

Wrapper Classes

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

NOTE:

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

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

Why Wrapper Class is important?

Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types), not primitive 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 +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

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)
```

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

- This method exists in the numeric wrapper classes.
- 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

- `parseInt` method in the Integer class to parse a numeric string into an int value
- `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.

Autoboxing and Unboxing

- **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes.

Example 1

```
char ch = ‘a’;
// Autoboxing- primitive to Character object conversion
Character a = ch;
```

Example 2

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
// Autoboxing because ArrayList stores only objects
arrayList.add(25);
```

Autoboxing and Unboxing

- **Unboxing**: Automatically converting an object of a wrapper class to its corresponding primitive type.

Example 1

```
Character ch = 'a';
// unboxing - Character object to primitive conversion
char a = ch;
```

Example 2

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(24);
// unboxing because get method returns an Integer object
int num = arrayList.get(0);
```

Automatic Conversion Between Primitive Types and Wrapper Class Types

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

(a)

Equivalent

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

New JDK 1.5 boxing

(b)

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

Unboxing

BigInteger and BigDecimal

- The `BigInteger` and `BigDecimal` classes in the `java.math` package for compute very large integers or high precision floating-point values.
- 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);
```

Java Built-in Classes

The `String` Class

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:

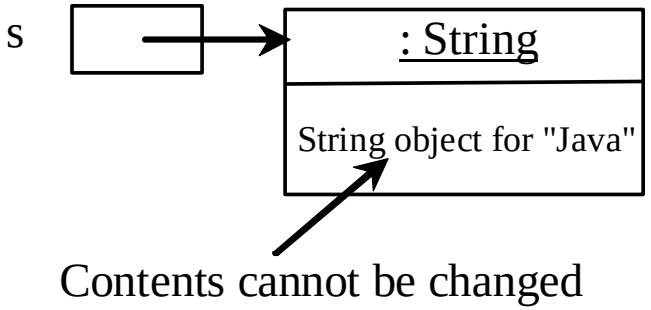
```
String message = "Welcome to Java";
```

Strings Are Immutable

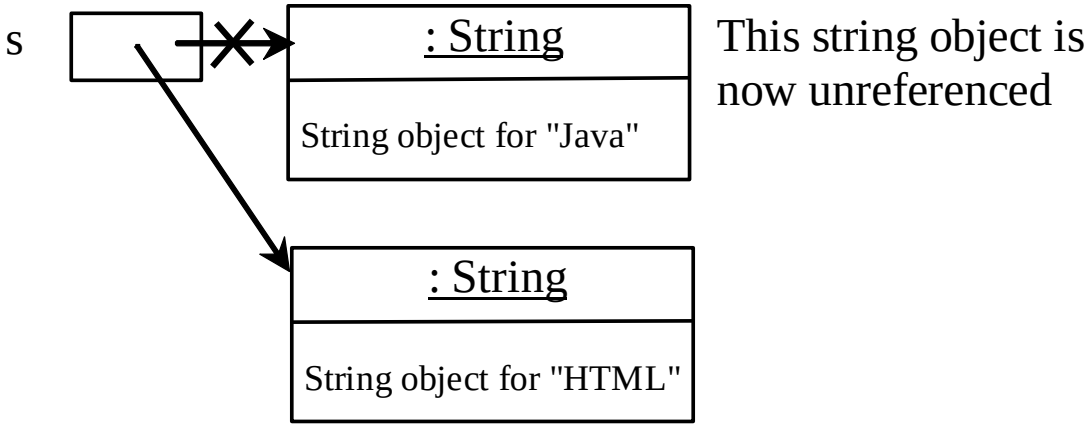
A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

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

After executing `String s = "Java";`



After executing `s = "HTML";`



Interned Strings (Java String Pool)

- 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***.

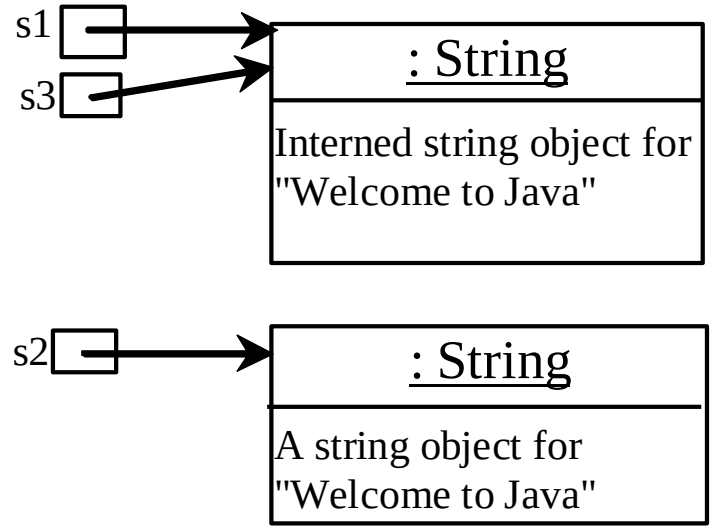
Interned Strings Example

```
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));
```



Output

s1 == s 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 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.

Replacing and Splitting Strings

`"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] + " ");
```

Output :

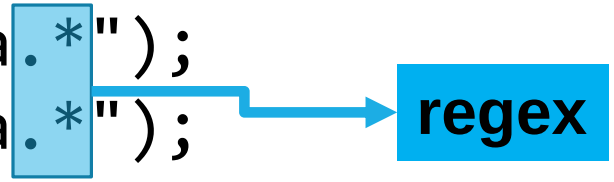
```
Java HTML Perl
```

Matching by Patterns

- matches function can be used to match a String object with another String object, or with a regex (regular expression).
- A **regex** is a particular pattern for matching a set of strings.

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

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



Regex 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"

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);
```

Output : `aNNNbNNNNNNc`.

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]);
```

Output :

Java
C
C#
C++

Convert Character and Numbers to Strings

- using 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:

```
String.valueOf(5.44)
```

Output

```
"5.44"
```

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.

StringBuilder Constructors

StringBuilder adalah class untuk string yang bersifat mutable.
StringBuilder lebih cocok digunakan untuk membuat string yang akan dikenakan banyak modifikasi.

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.

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.

Examples

```
StringBuilder stringBuilder = new StringBuilder("Welcome to ");
stringBuilder.append("Java"); // Welcome to Java
stringBuilder.insert(11, "HTML and "); // Welcome to HTML and Java
stringBuilder.delete(8, 11) // Welcome HTML and Java
stringBuilder.deleteCharAt(8) // Welcome TML and Java.
stringBuilder.reverse() // avaJ dna LMT emocleW.
stringBuilder.replace(11, 15, "HTML") // avaJ dna LMHTMLocleW.
stringBuilder.setCharAt(0, 'w') // wvaJ dna LMHTMLocleW.
```

Checking Palindromes

```
/** Return true if a string is a palindrome */
public static boolean isPalindrome(String s) {
    String s1 = filter(s);
    String s2 = reverse(s1);
    return s2.equals(s1);
}
/** Create a new string by eliminating non-alphanumeric chars */
public static String filter(String s) {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        if (Character.isLetterOrDigit(s.charAt(i))) {
            stringBuilder.append(s.charAt(i));
        }
    }
    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();
    return stringBuilder.toString();
}
```

To Be Continued …