UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Dasar Dasar Pemrograman 2

Acuan: Introduction to Java Programming and Data Structure

Sumber Slide: Liang, Pearson + https://www.baeldung.com/java-composition-aggregation-association

Dimodifikasi untuk Fasilkom UI oleh Ade Azurat

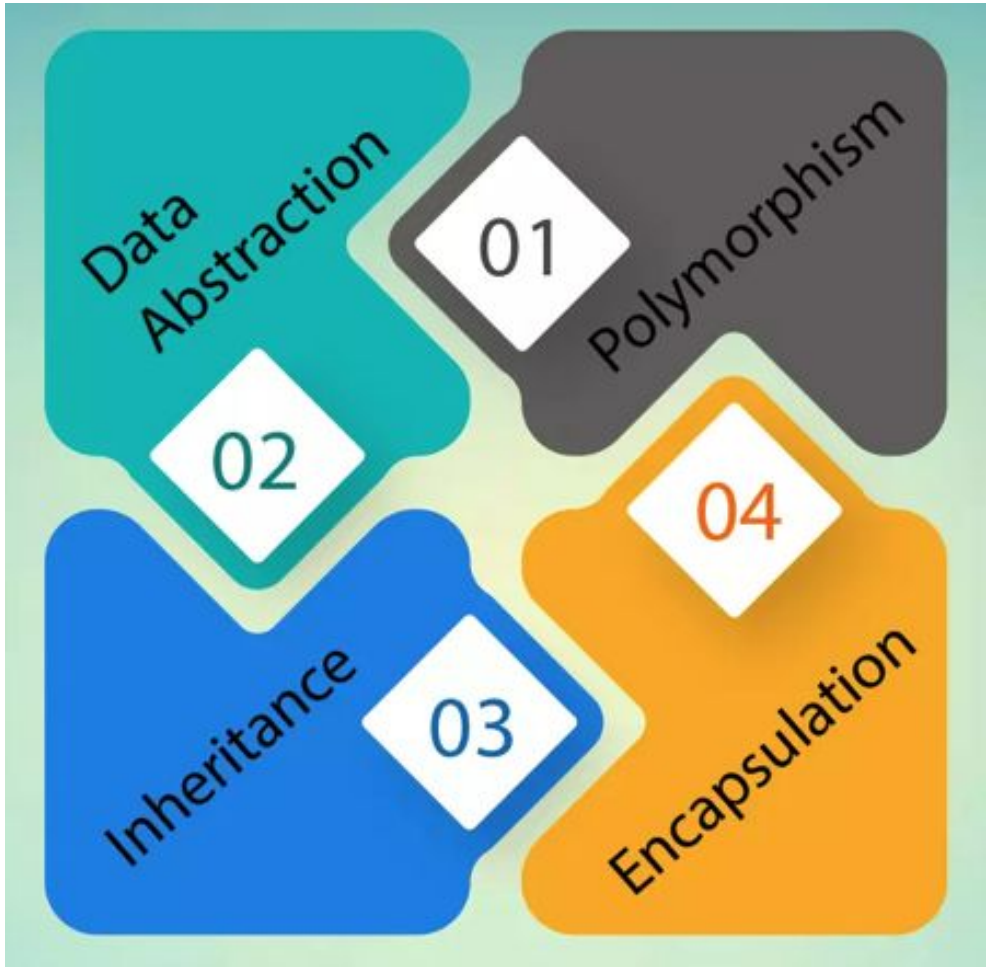
Topik 05: Ch.10 Object Oriented Thinking

Motivations

You see the object-oriented programming from the preceding chapter.

This chapter will demonstrate how to solve problems using the object-oriented paradigm.

This section, we start by discussing the encapsulation



Sumber gambar: https://images.theengineeringprojects.com/image/webp/2021/11/6-2.jpg.webp?ssl=1

Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).
- To create objects for primitive values using the wrapper classes (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, and **Boolean**) (§10.7).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).
- To use the **BigInteger** and **BigDecimal** classes for computing very large numbers with arbitrary precisions (§10.9).
- To use the **String** class to process immutable strings (§10.10).
- To use the **StringBuilder** and **StringBuffer** classes to process mutable strings (§10.11).

Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops

Chapter 9 introduced classes and objects. It provide more flexibility and modularity for building reusable software.

This section (Chapter 10) improves the solution 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 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

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.

Example: The Course Class

Lihat: https://replit.com/@AdeAzurat/DDP2-Pekan07-OOThinking#Exercise10_09.java

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.

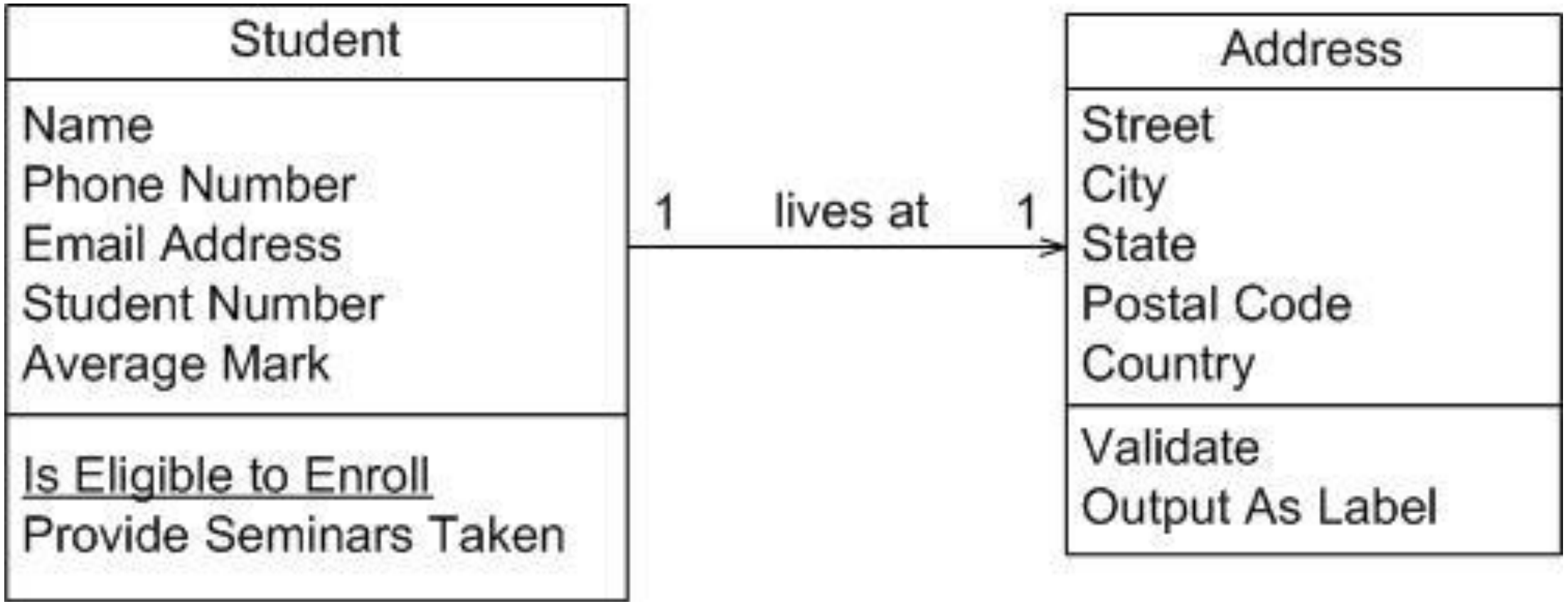
Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.

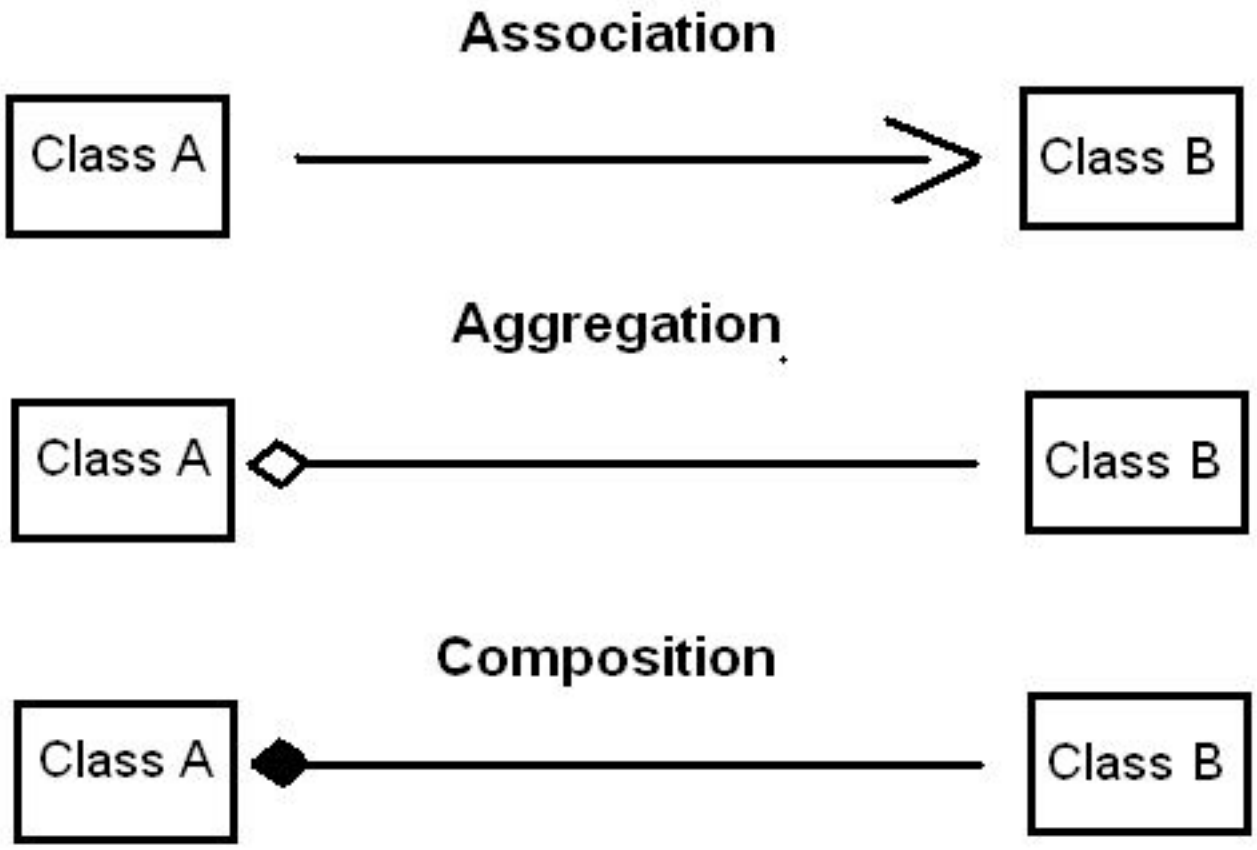
UML: Class Diagram



Association, Aggregation and Composition

- Association is a relationship between two objects.
  - In other words, association defines the multiplicity between objects.
  - You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects.
- Aggregation is a special form of association.
- Composition is a special form of aggregation.

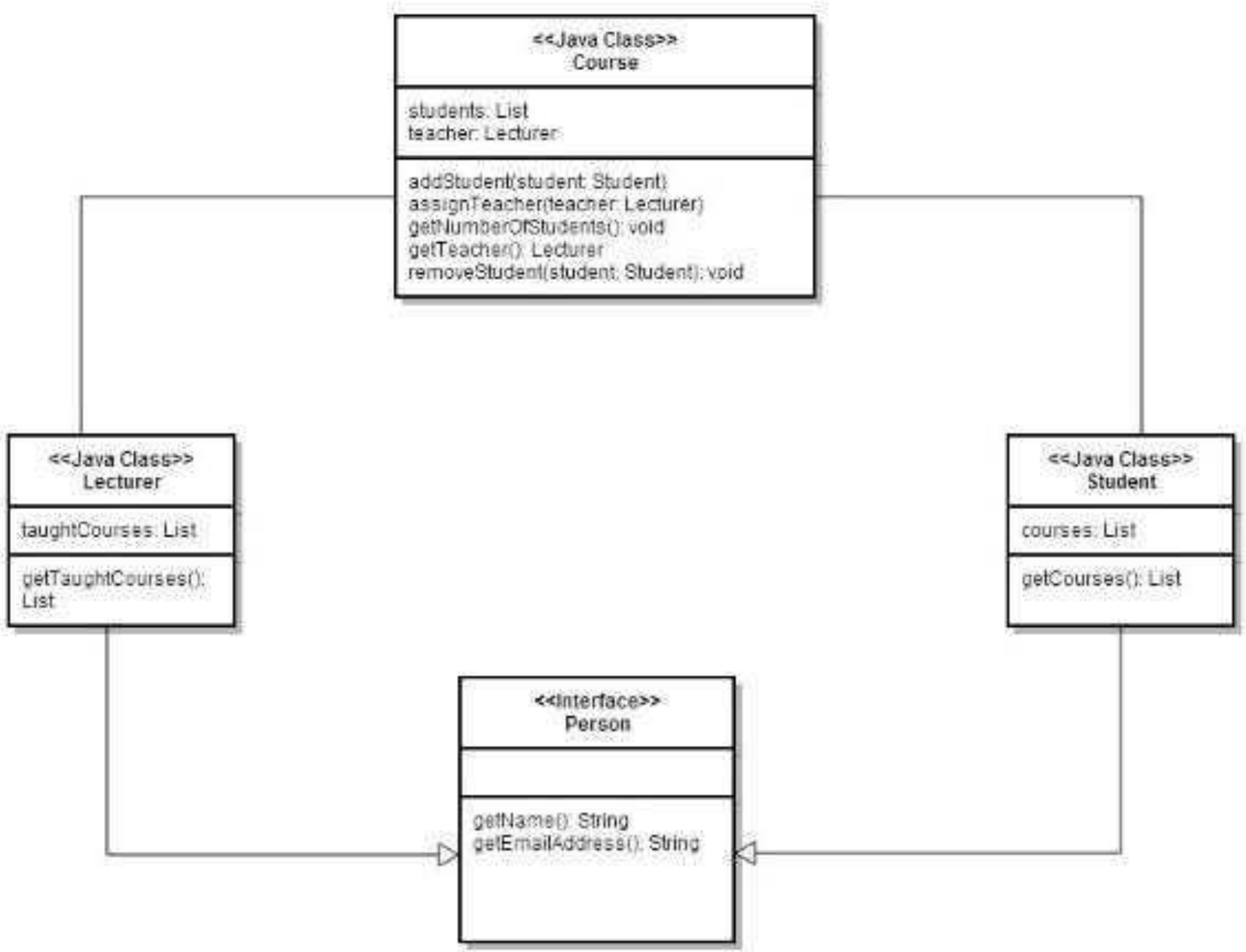
UML: Class Relations



A Student and a course are having an association.

A Student has an address. If the student was gone, the address remains.

A university and a faculty are having an Composition. A faculty cannot exists without a university

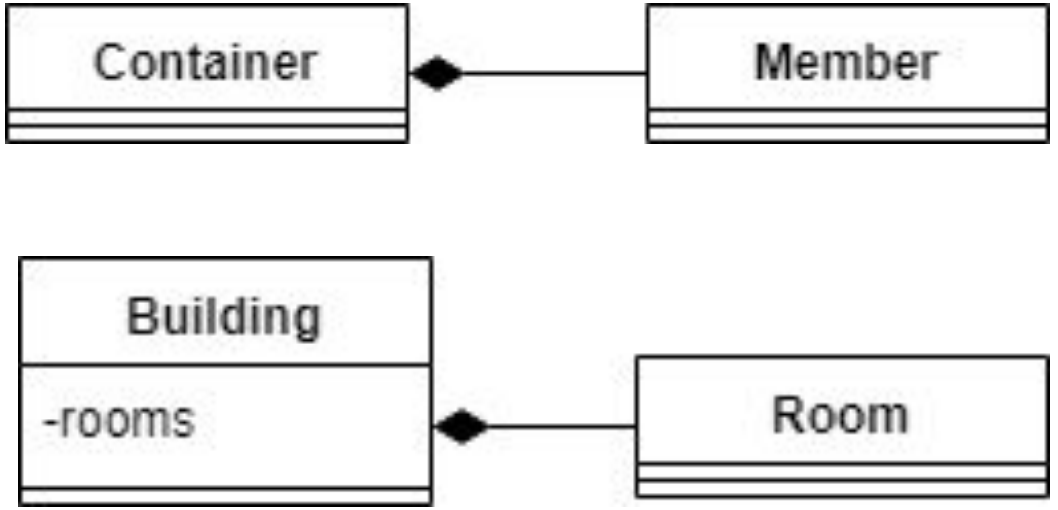

<<Java Class>>
Course
students: List
teacher: Lecturer
addStudent(student: Student)
assignTeacher(teacher: Lecturer)
getNumberOfStudents(): void
getTeacher(): Lecturer
removeStudent(student: Student): void
<<Java Class>>
Lecturer
taughtCourses: List
getTaughtCourses(): List
<<Java Class>>
Student
courses: List
getCourses(): List
<<Interface>>
Person
getName(): String
getEmailAddress(): String

Composition – Class Representation

- *Composition* is a "belongs-to" type of relationship. It means that one of the objects is a logically larger structure, which contains the other object. In other words, it's part or member of the other object.
- Composition is a strong kind of "has-a" relationship because the containing object owns it.
- Therefore, **the objects' lifecycles are tied. It means that if we destroy the owner object, its members also will be destroyed with it.**
- For example: A building is a composition of rooms. The room is destroyed with the building.

Composition – Class Representation

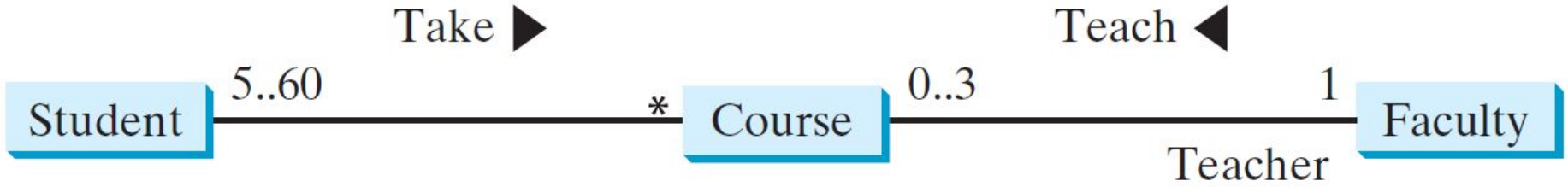
- Note, that the diamond is at the containing object and is the base of the line, not an arrowhead.
- So, then, we can use this UML construct for our Building-Room example:




```
class Building {
   Room[] room;
}
```

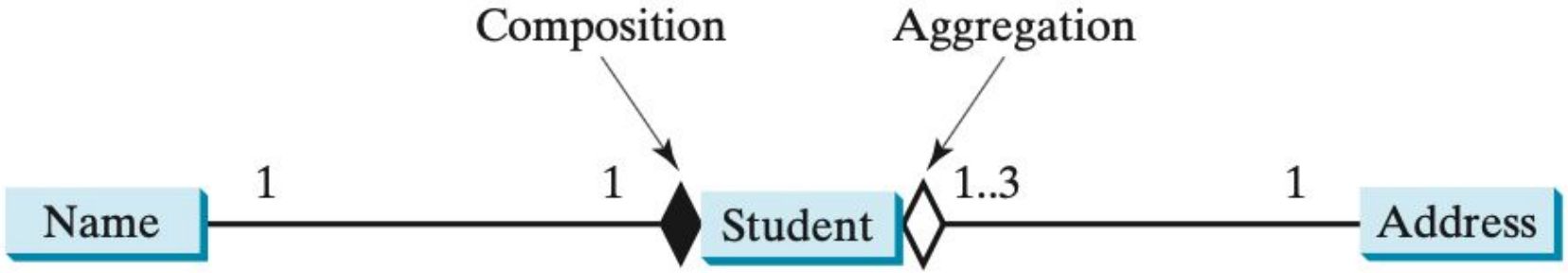
Aggregation

- 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*.



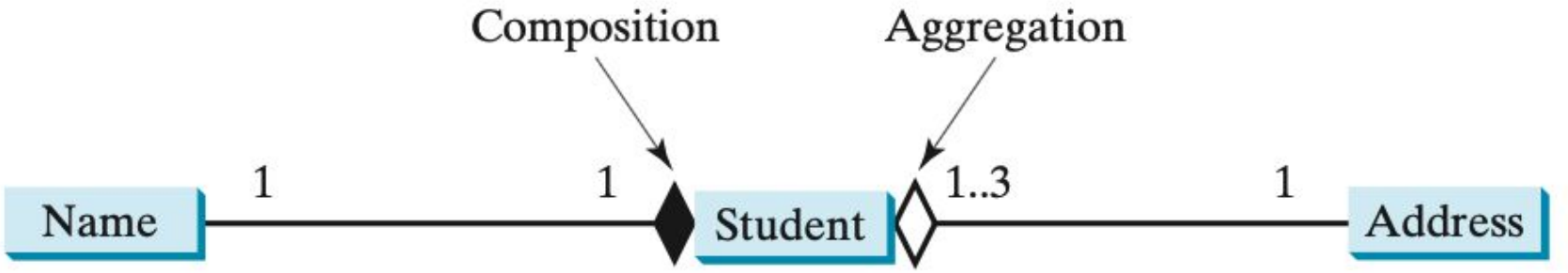
Aggregation or Composition?

- *Aggregation* is a special form of association that represents an ownership relationship between two objects but *not exclusive*.
  - When the owner object is destroyed, the dependent object may still live.
  - If object student is destroyed, object Address may still exist.
- Composition implies *exclusive* ownership.
  - One object owns another object.
  - When the owner object is destroyed, the dependent object is destroyed as well.
  - If object student is destroyed, object name is also destroyed.



Aggregation 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

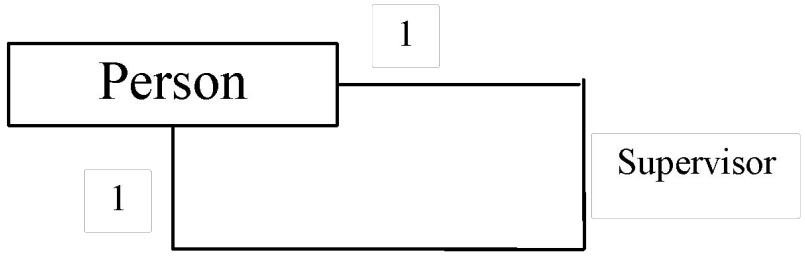
```
public class Address {
  ...
}
```

Aggregated class

Aggregation Between Same Class

Aggregation may exist between objects of the same class.

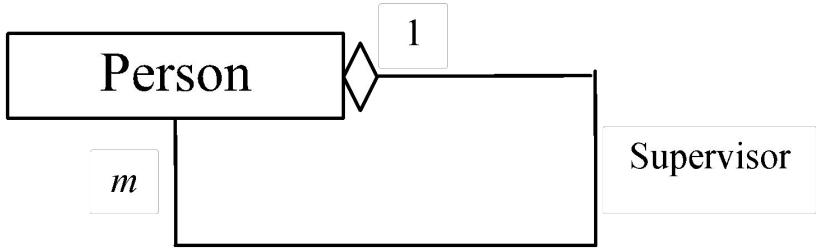
For example, a person may have a supervisor.




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

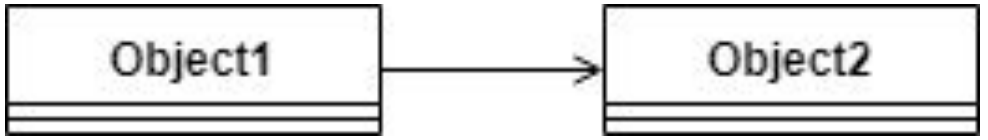
Aggregation Between Same Class

What happens if a person has several supervisors?

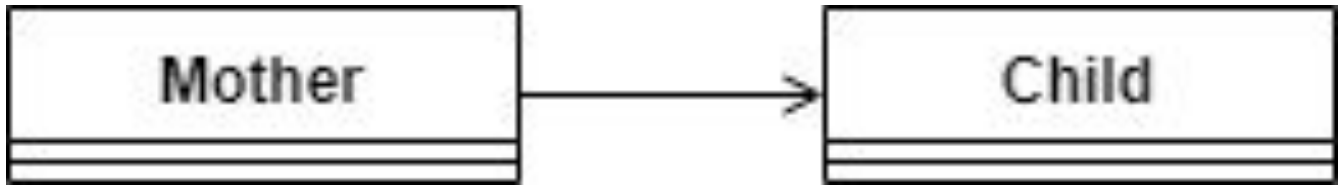



```
public class Person {
   ...
   private Person[] supervisors;
}
```

Association



- Association is the weakest relationship between the three. **It isn't a "has-a" relationship**, none of the objects are parts or members of another.
- **Association only means that the objects "know" each other.** For example: A student and a course (or some courses). A mother and her children.

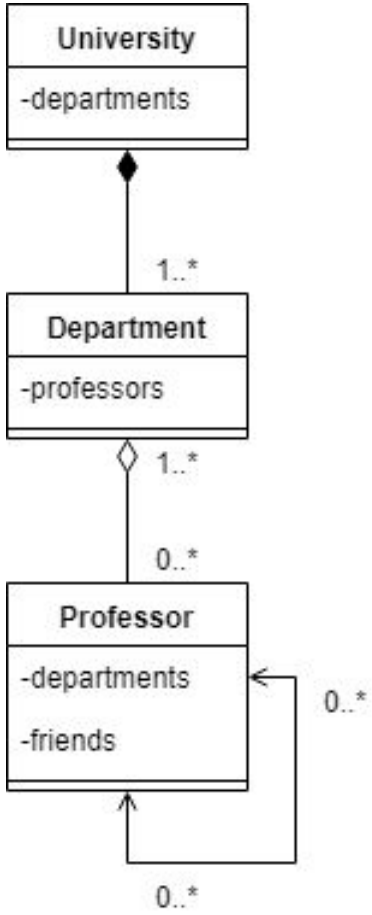



```
class Child {
}

class Mother {
    Child[] children;
}
```

Example

We will model a university, which has its departments. Professors work in each department, who also has friends among each other.

- Will the departments exist after we close the university? Of course not, therefore it's a composition.
- But the professors will still exist (hopefully). We have to decide which is more logical: if we consider professors as parts of the departments or not. Alternatively: are they members of the departments or not? Yes, they are. Hence it's an aggregation. On top of that, a professor can work in multiple departments.
- The relationship between professors is association because it doesn't make any sense to say that a professor is part of another one.




```
class University {
    Department[] department;
}

class Department {
    Professor[] professors;
}

class Professor {
    Department[] department;
    Professor[] friends;
}
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Ada Pertanyaan?

- ❑ Aggregation dan Composition
- ❑ Object sebagai Field?
- ❑ Perbedaan Association, Aggregation, Composition
- ❑ Aggregation pada satu kelas yang sama

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FACULTY OF COMPUTER SCIENCE

How To Build Class Diagram?

Applying Object Oriented Methodology

OO Methodology

- A methodology (sometimes simply called a method) is a set of processes and heuristics used to break down the complexity of a programming problem.
- Keep in mind what you're trying to discover:
  - What are the objects? (How do you partition your project into its component parts?)
  - What are their methods? (What messages do you need to send to each object?)

Class Responsibility Collaboration

- You start out with a set of blank 3 x 5 cards, and you write on them.
- Each card represents a single class, and on the card you write:
  - The name of the class
  - Its responsibilities
  - Its collaborators

CRC (cont.)

- The name of the class.
  - It’s important that this name capture the essence of what the class does,
  - so that it makes sense at a glance.

CRC (cont.)

- The “responsibilities” of the class:
  - what it should do.
  - This can typically be summarized by just stating the names of the methods.

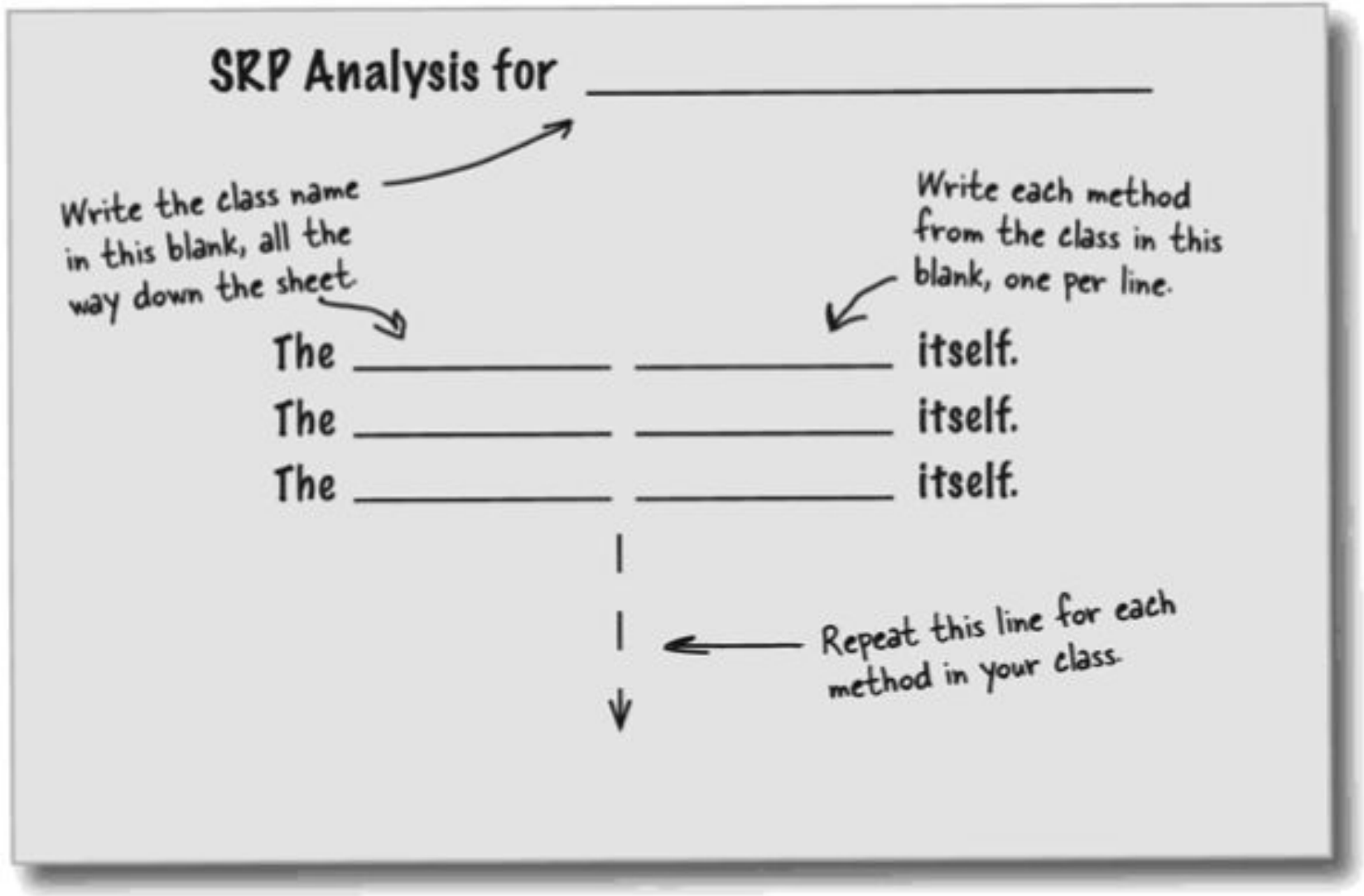
CRC (cont.)

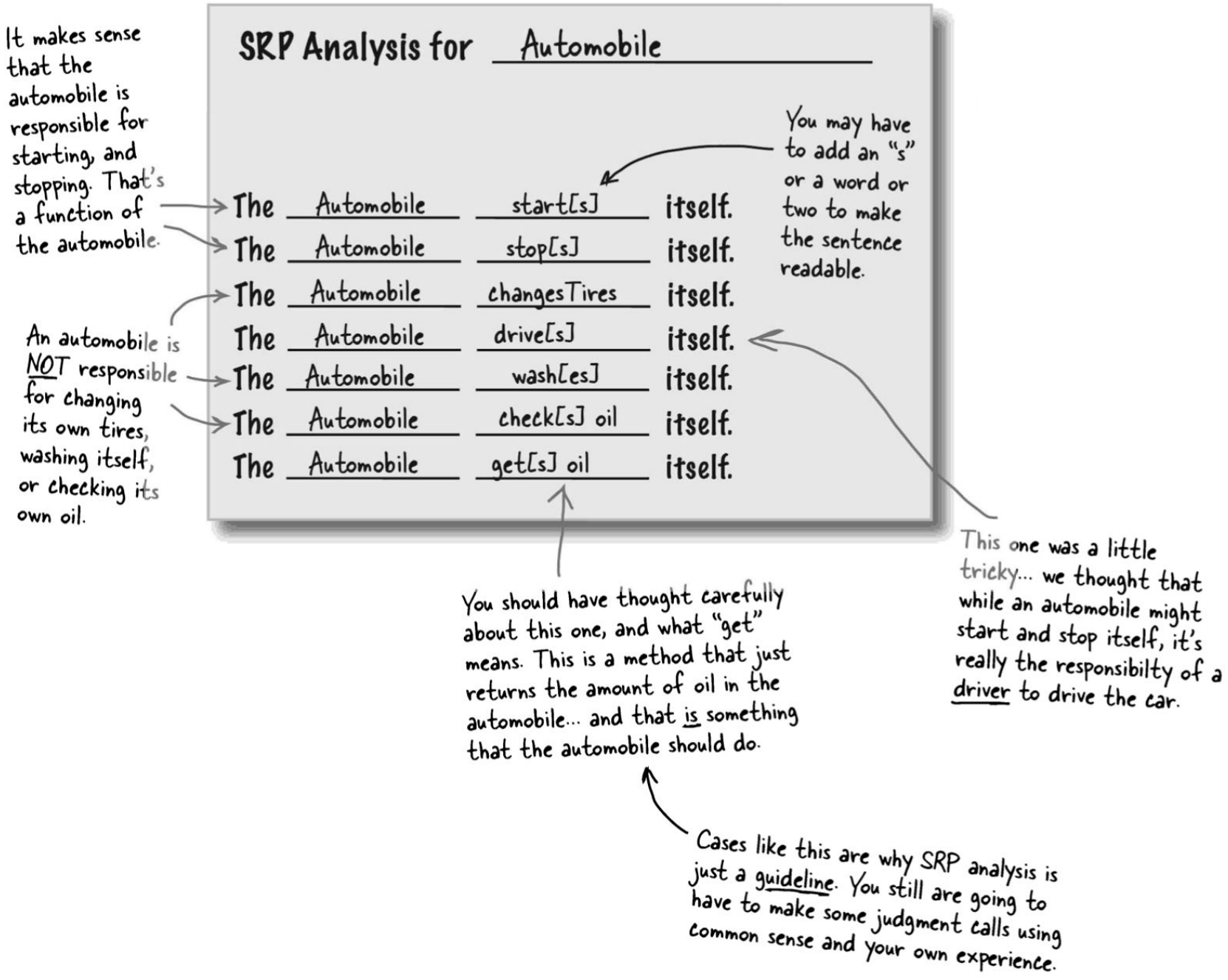
- The “collaborations” of the class:
  - What other classes does it interact with?
  - Collaborations should also consider the audience for this class.

CRC and SRP

- CRC helps implements Single Responsibility Principle (SRP)
- SRP means: a class should only have one main responsibility.
- If you've got a class that seems to violates SRP,
  you can use the CRC to sort out which classes should be doing what.
- SRP is one of the SOLID principles (We will study the other principles later after the Mid-Term (UTS))

Here's what your SRP analysis sheet should look like.




It makes sense that the automobile is responsible for starting, and stopping. That's a function of the automobile.
SRP Analysis for Automobile
You may have to add an "s" or a word or two to make the sentence readable.
The Automobile start[s] itself.
The Automobile stop[s] itself.
The Automobile changesTires itself.
The Automobile drive[s] itself.
The Automobile wash[es] itself.
The Automobile check[s] oil itself.
The Automobile get[s] oil itself.
An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.
This one was a little tricky... we thought that while an automobile might start and stop itself, it's really the responsibilty of a driver to drive the car.
You should have thought carefully about this one, and what "get" means. This is a method that just returns the amount of oil in the automobile... and that is something that the automobile should do.
Cases like this are why SRP analysis is just a guideline. You still are going to have to make some judgment calls using common sense and your own experience.

SRP Analysis for Automobile

The	Automobile	start[s]	itself.
The	Automobile	stop[s]	itself.
The	Automobile	changesTires	itself.
The	Automobile	drive[s]	itself.
The	Automobile	wash[es]	itself.
The	Automobile	check[s] oil	itself.
The	Automobile	get[s] oil	itself.

If you've got a class that seems to violate the SRP, you can use a CRC card to sort out which classes should be doing what.

Class: Automobile

Description: This class represents a car and its related functionality

Responsibilities:

Name	Collaborator
Starts itself.	
Stops itself.	
Gets tires changed	Mechanic, Tire
Gets driven	Driver
Gets washed	CarWash, Attendant
Gets oil checked	Mechanic
Reports on oil levels	

Any time you see "Gets", it's probably not the responsibility of this class to do a certain task.

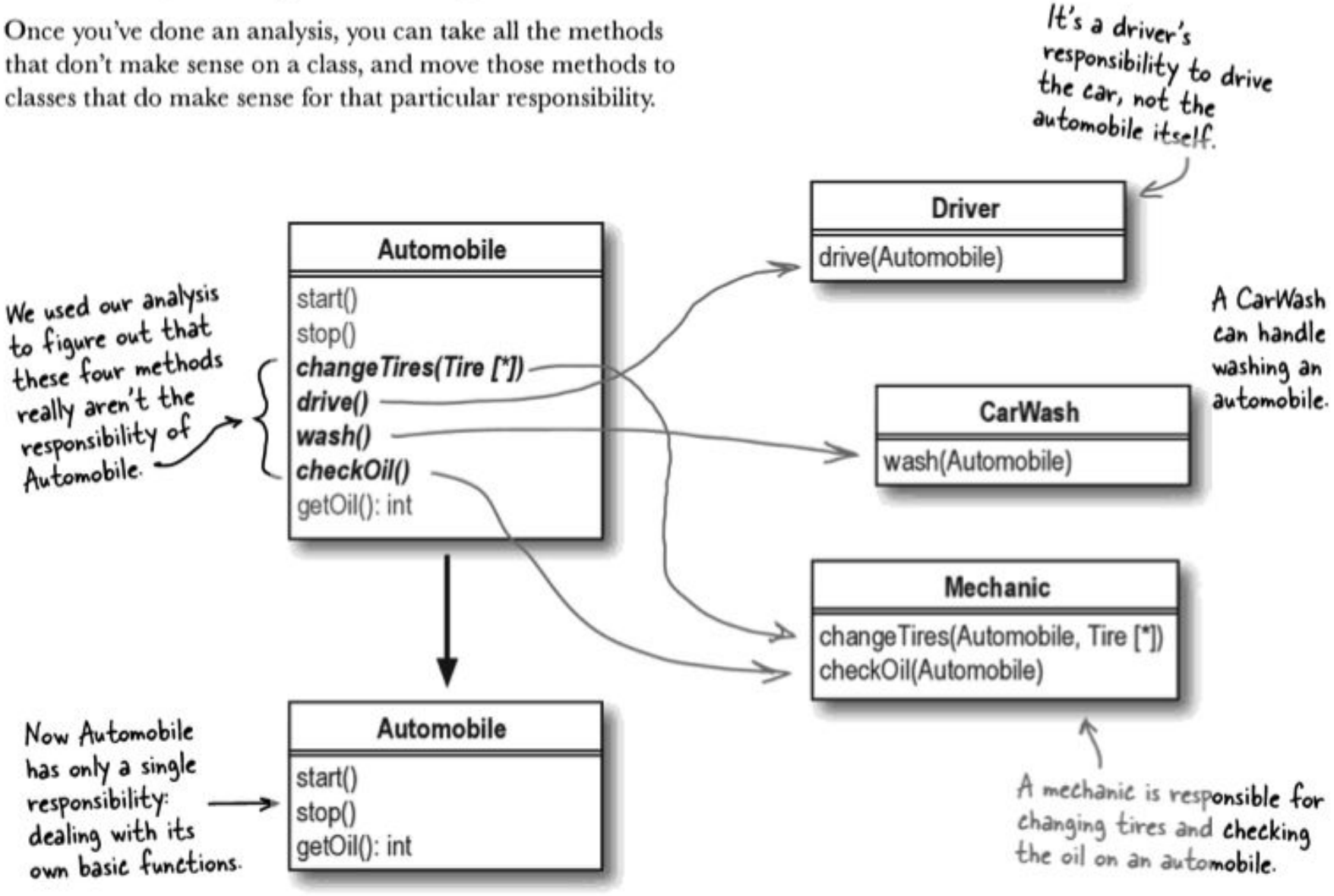
Technically, you don't need to list responsibilities that AREN'T this class's, but doing things this way can help you find tasks that really shouldn't be on this class.

CRC and UML

- Once you've come up with a set of CRC cards, you may want to create a more formal description of your design using UML.
- You don't have to use UML, but it can be helpful, especially if you want to put up a diagram on the wall for everyone to ponder, which is a very good idea.
- UML has become a standard language in software engineering design.

Going from multiple responsibilities to a single responsibility

Once you've done an analysis, you can take all the methods that don't make sense on a class, and move those methods to classes that do make sense for that particular responsibility.



Diskusi Kelompok

- Buat kelompok 4 orang
- Pikirkan hal terkait dengan sistem SIAK, meliputi aspek: mahasiswa, dosen, asdos, kelas, mata kuliah, jadwal, program studi, orang tua, alamat, nilai, sks, kurikulum dan lain-lain yang terpikirkan.
- Diskusikan bersama bagaimana hubungan antara masing-masing aspek tersebut.
- Masing-masing individu, harap membuat dua *CRC Cards* yang berbeda.
- Kemudian gabungkan seluruh CRC, perhatikan apakah setiap class sudah memenuhi SRP atau tidak. Buat *SRP analysis* –nya.
- Buat Class Diagram –nya. Masing-masing peserta, dipersilahkan menyalin keseluruhan rancangan untuk kebutuhan implementasi pribadinya.

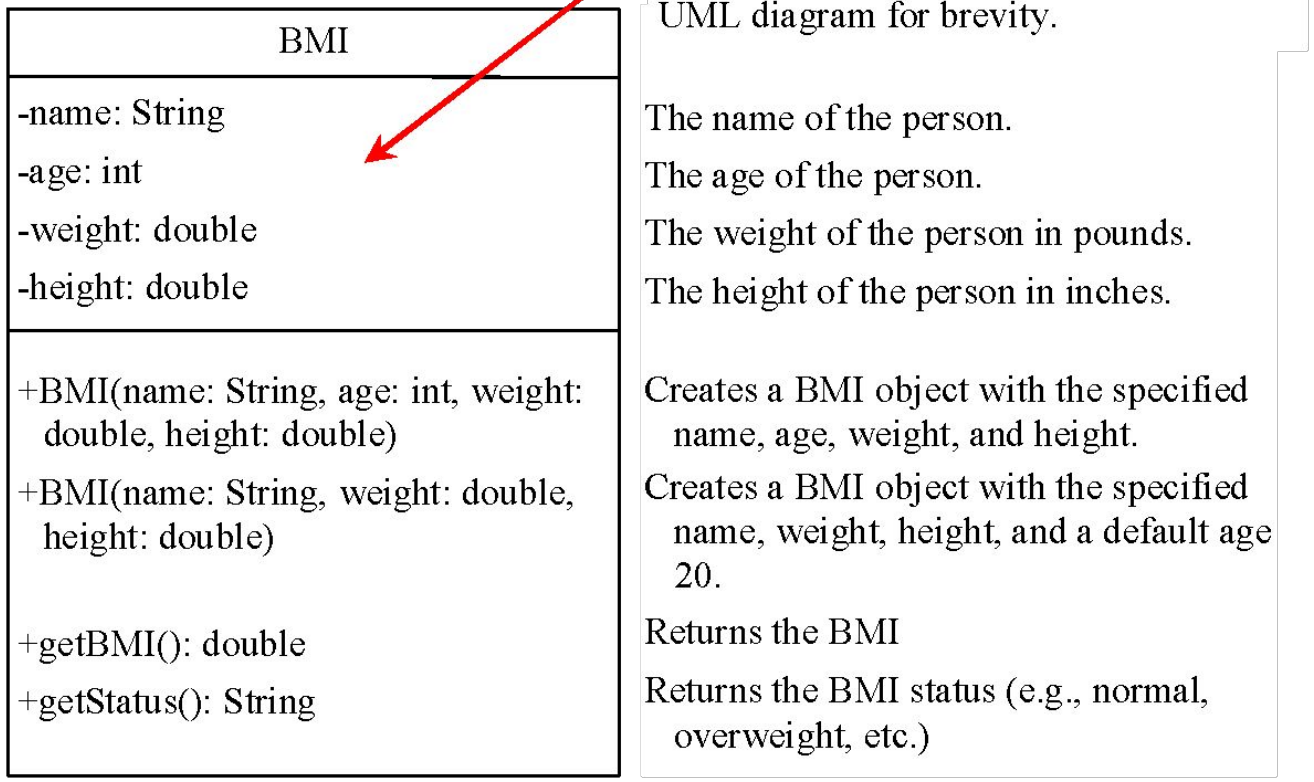
UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Ada Pertanyaan?

- ❑ CRC
- ❑ SRP
- ❑ How to Build Class Diagram
- ❑ How to apply the correct association

The BMI Class (Body Mass Index)



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.

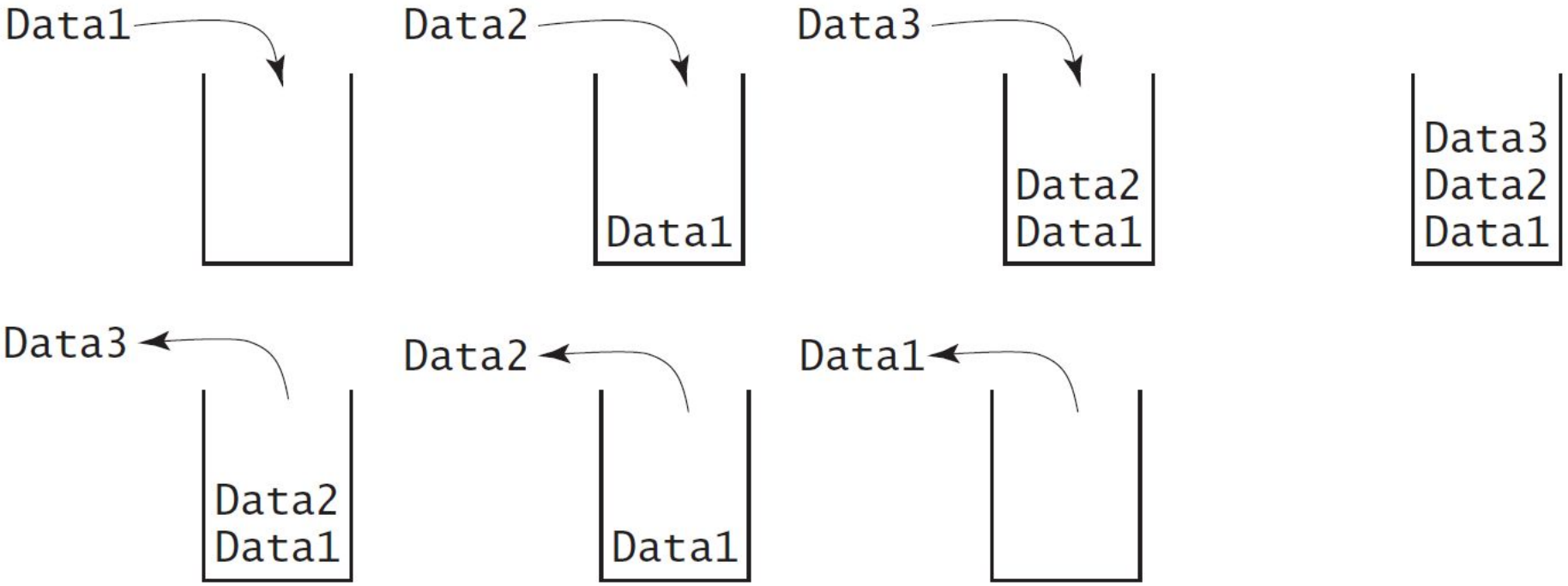
Returns the BMI

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

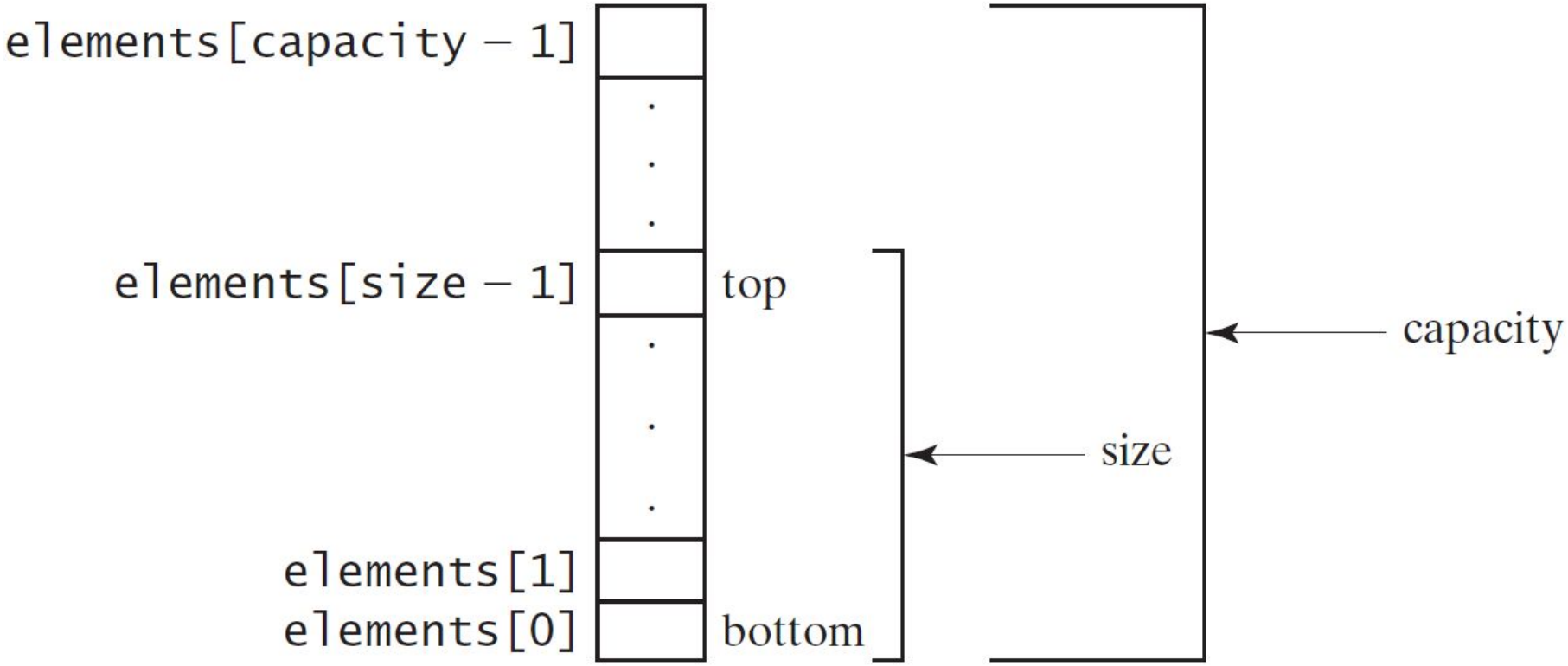
Example: The `StackOfIntegers` Class

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.

Designing the StackOfIntegers Class



Implementing `StackOfIntegers` Class



Wrapper Classes

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

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.

The `Integer` and `Double` Classes

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

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

The `Integer` Class and the `Double` Class

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

Numeric Wrapper Class Constructors

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)
```

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};
```

New JDK 1.5 boxing

(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("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);
```

String in Java

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";
```

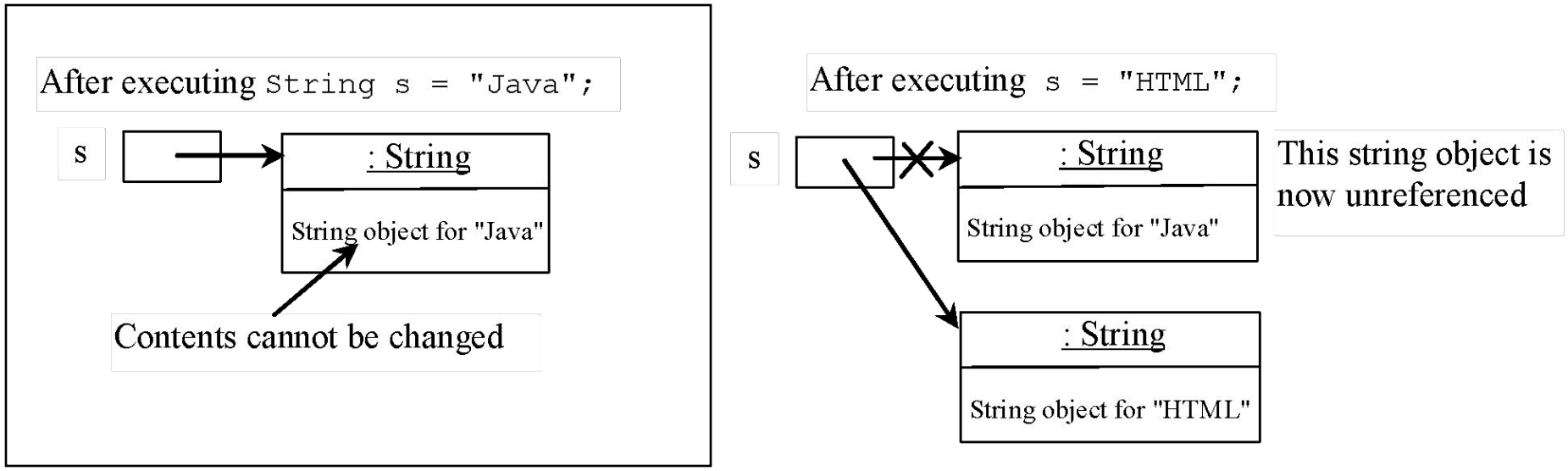
Strings Are Immutable

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

```
String s = "Java";

s = "HTML";
```

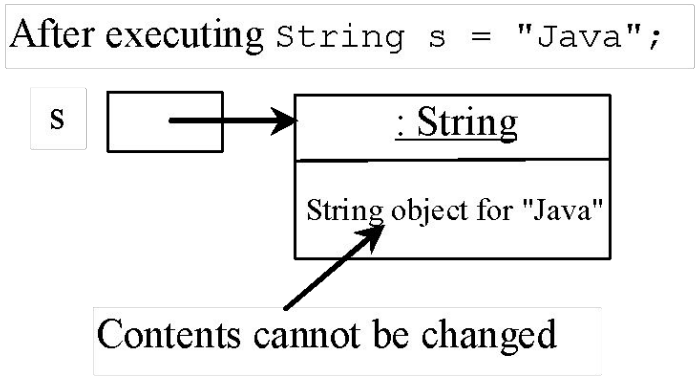
Trace Code

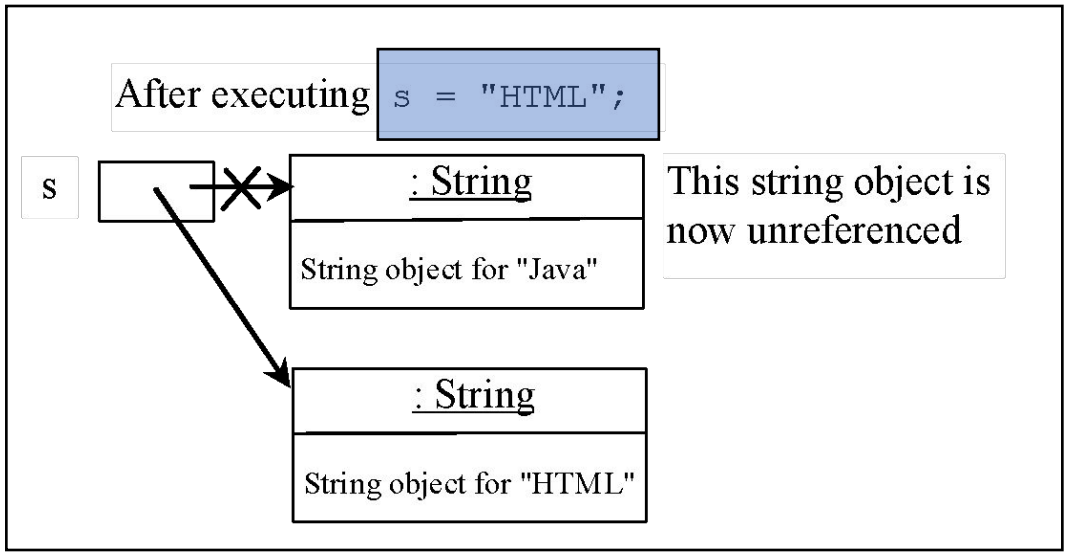
```
String s = "Java";
s = "HTML";
```



Trace Code

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





Interned Strings

Since strings are immutable and are frequently used,

to improve efficiency and save memory,

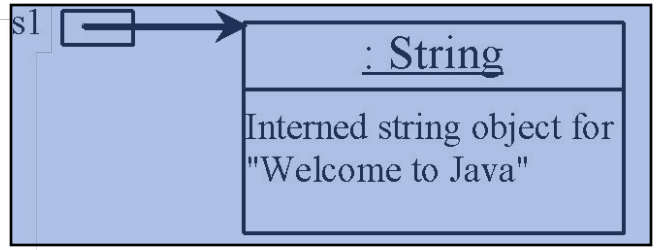
the JVM uses a unique instance for string literals with the same character sequence.

Such an instance is called *interned*.

For example, the following statements:

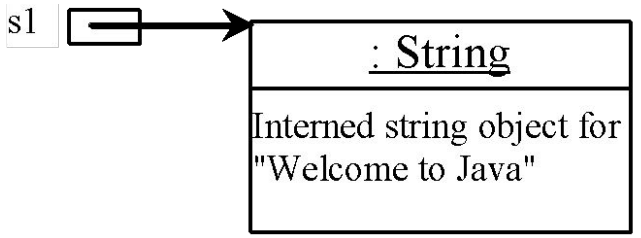
Trace Code

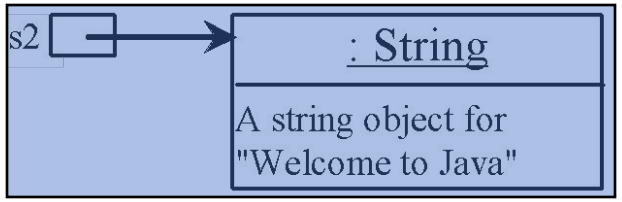
```
String s1 = "Welcome to Java";

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

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



Trace Code

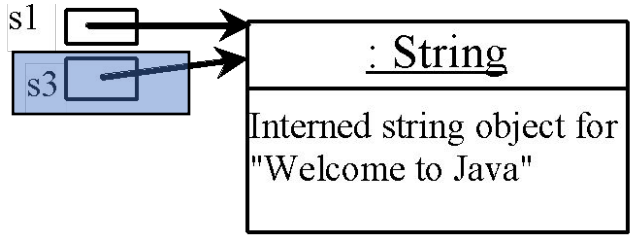
```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```

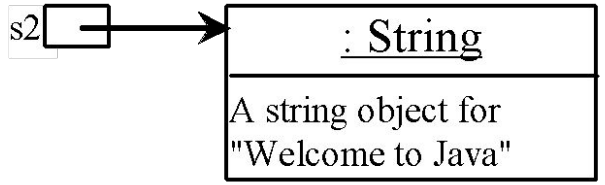




Trace Code

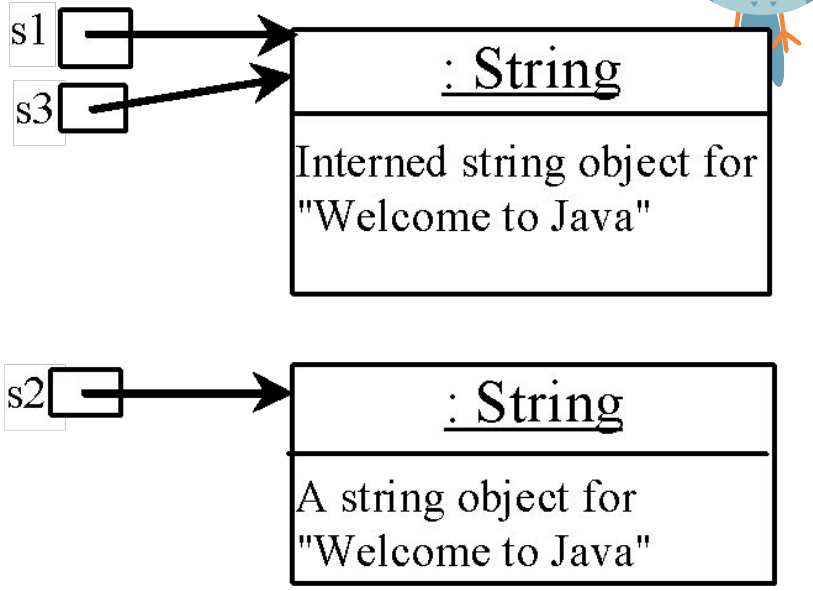
```
String s1 = "Welcome to Java";

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

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





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 == 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. (***Interned String***)

Silahkan cari tahu tentang *interned string*!

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".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] + " ");
```

displays

Java HTML Perl

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***.

This topic will be studied deeper on the next courses.

In this section, two simple patterns are used.

Please refer to Supplement III.F, "Regular Expressions," for further studies or other resources.

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

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

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]);
```

Silahkan coba run, berapa token.length?

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'.

`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

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

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.

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.

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.

Bisakah anda buat versi yang lebih efisien?
Bisakah anda membuat nya lebih OO?

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

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 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") ;

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

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

Silahkan coba run, berapa ss.length?

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Ada Pertanyaan?

- ❑ Class Diagram
- ❑ Relasi Antar Class
- ❑ Memetakan Class menjadi java source code
- ❑ Wrapping Class
- ❑ Berbagai macam class terkait String
- ❑ Regular Expression

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS
ILMU
KOMPUTER

Selamat Berlatih!

Perhatikan lagi List Objective yang perlu dikuasai pekan ini.
Baca buku acuan dan berlatih!
Bila masih belum yakin tanyakan ke dosen, tutor atau Kak Burhan.

Semangat !

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS
ILMU
KOMPUTER

Pekan :

Topik :