جامعة بيرزيت
BIRZEIT UNIVERSITY

Thinking in Objects

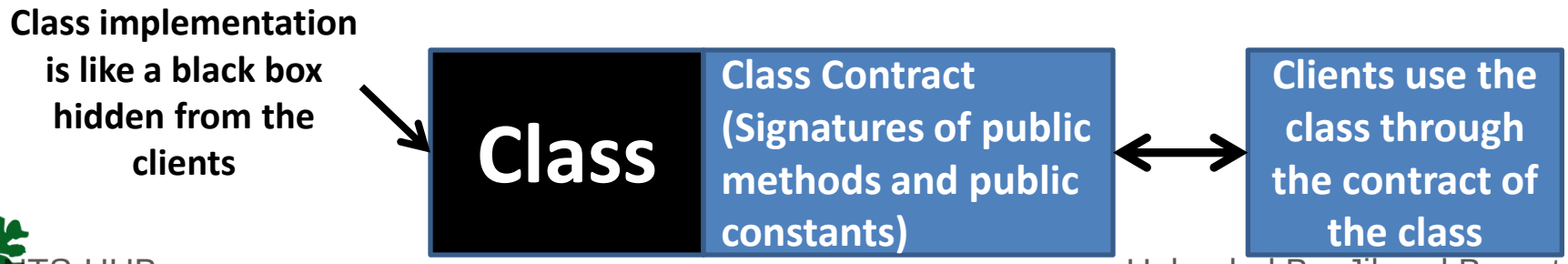
Class Abstraction and Encapsulation

❖ Class ***abstraction*** means to separate class implementation from the use of the class.

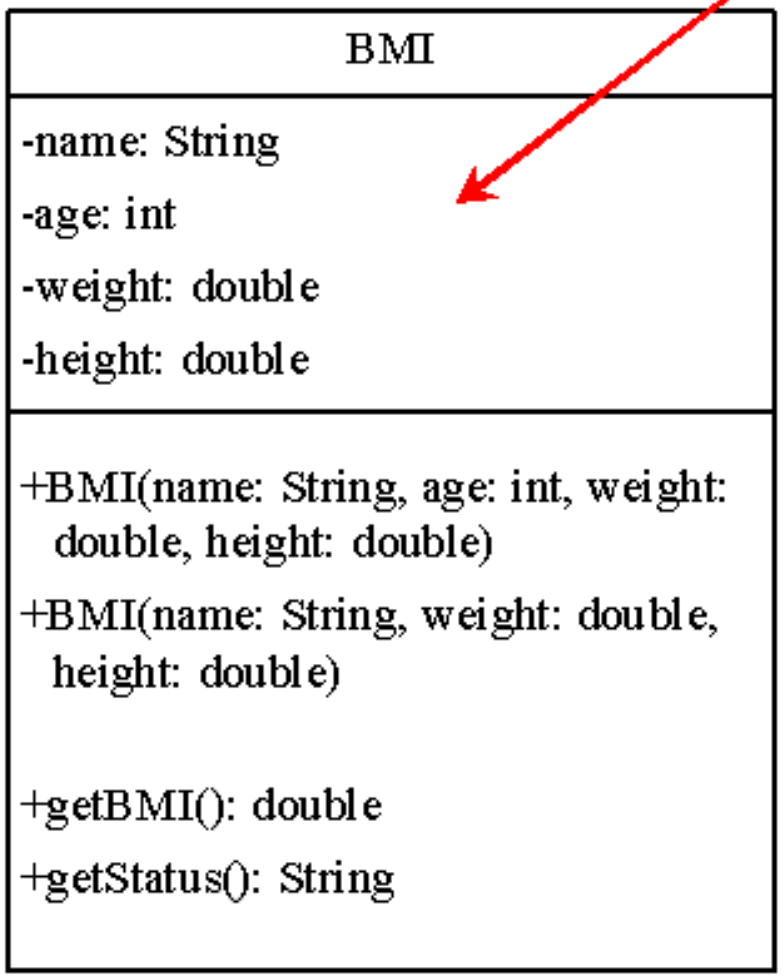
❖ The creator of the class provides a description of the class and let the user know how the class can be used.

❖ 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.



Case Study: The BMI Class



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.

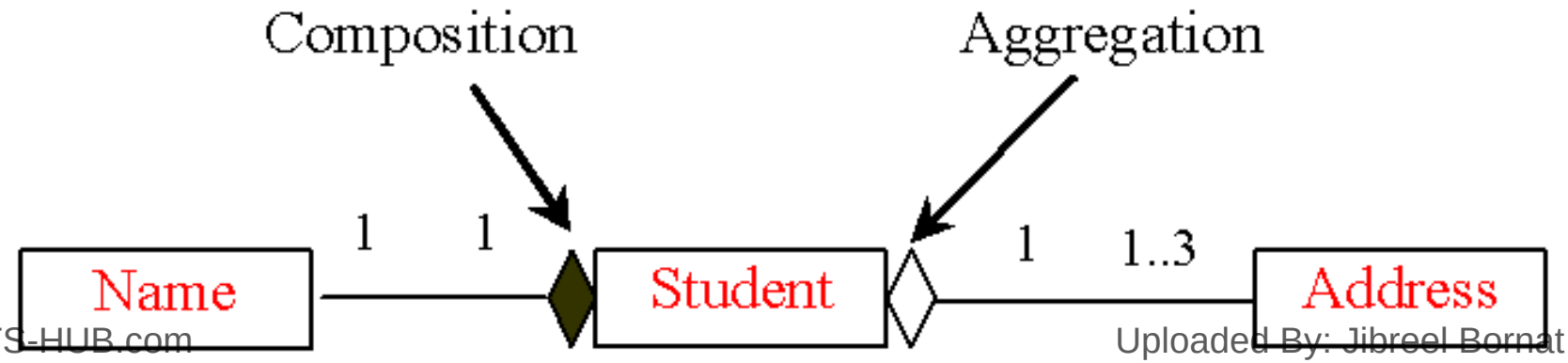
Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

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

Object Composition

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



Class Representation

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

❖ For example, the relationship in the previous Figure can be represented as follows:

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

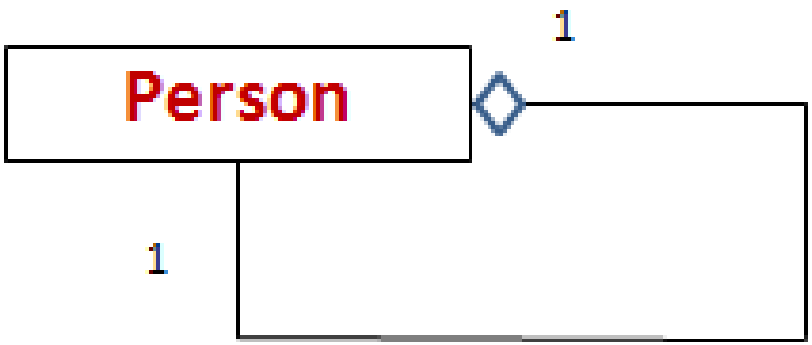
```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

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

Aggregation Between Same Class

❖ Aggregation may exist between objects of the same class.

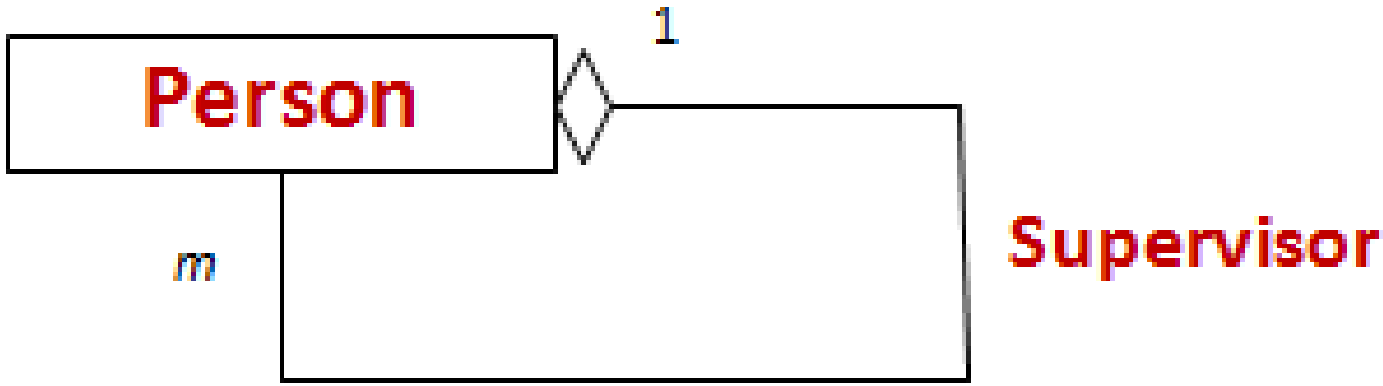
❖ For example, a **person** may have a **supervisor**:




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

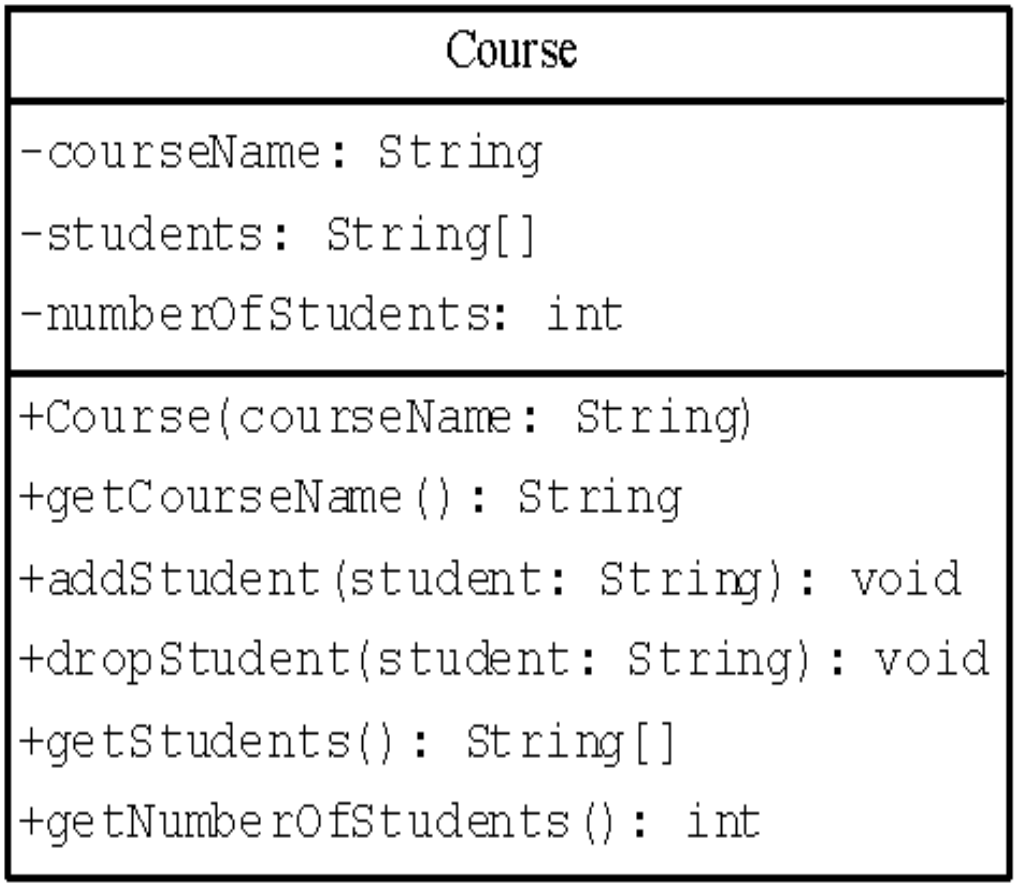
Aggregation Between Same Class

❖ What happens if a person has several supervisors?




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

Example: The Course Class



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

Designing a Class

- (**Coherence**) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.

- You can use a class for **students**, for example, but you should not combine **students** and **staff** in the same class, because students and staff have different entities.

Designing a Class cont.

- (**Separating responsibilities**) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- Example: the classes **String**, **StringBuilder**, and **StringBuffer** all deal with strings, for example, but have different responsibilities:
  - **String** class deals with immutable strings.
  - **StringBuilder** class is for creating mutable strings.
  - **StringBuffer** class is similar to **StringBuilder** except that **StringBuffer** contains synchronized methods for updating strings.

Designing a Class cont.

❖ Classes are designed for **reuse**.

❖ Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it:

- Design the **properties** to ensure that the user can set properties in any order, with any combination of values.
- Design **methods** to function independently of their order of occurrence.

Designing a Class cont.

- **Follow standard Java programming style and naming conventions:**
  - Choose **informative names** for classes, data fields, and methods.
  - Always place the data declaration before the constructor, and place constructors before methods.
  - Always provide a constructor and **initialize** variables to avoid programming errors.

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

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.

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");
```

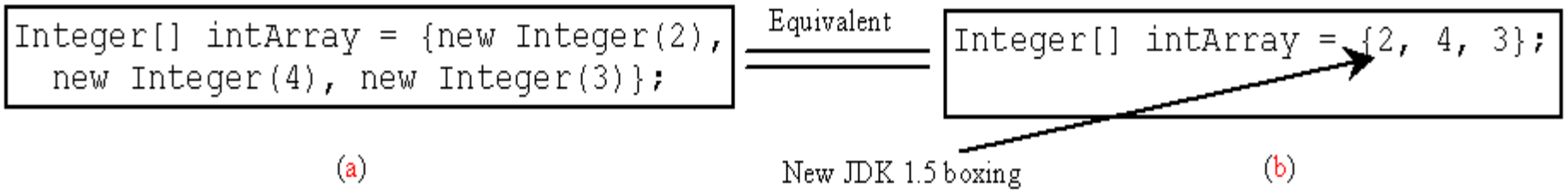
The Methods for Parsing Strings into Numbers

❖ 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[] arr = {1, 2, 3};

System.out.println(arr[0] + arr[1] + arr[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***.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("92233720368547758O7");
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);
```