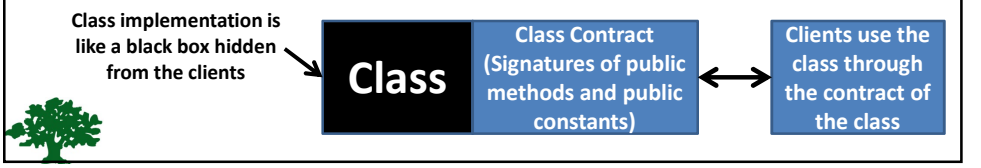
BIRZEIT UNIVERSITY

Thinking in Objects

By: Mamoun Nawahdah (PhD)
2022/2023

Class Abstraction and Encapsulation

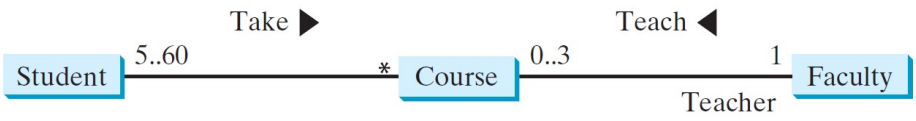
- Class ***abstraction*** means to *separate class implementation from the use of the class*.
- The creator of the class provides a description of the class and let the user know how the class can be used.
- The user of the class does not need to know how the class is implemented.
- The detail of implementation is **encapsulated** and hidden from the user.



Class Relationships

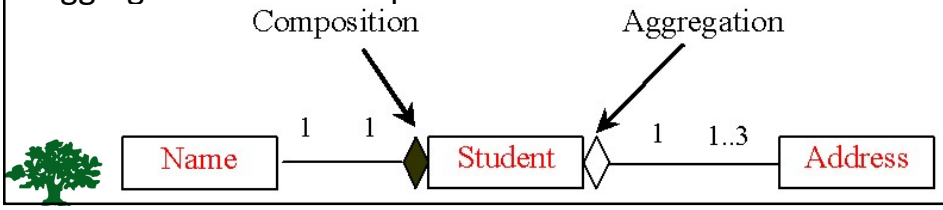
- **Association**
- **Aggregation**
- **Composition**
- **Inheritance** (Next Chapter)

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



Aggregation

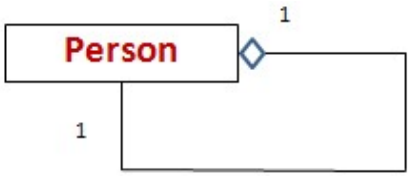
- **Aggregation** models ***has-a*** relationships and represents an **ownership** relationship between two objects.
- The **owner object** is called an ***aggregating object*** and its class an ***aggregating class***.
- The **subject object** is called an ***aggregated object*** and its class an ***aggregated class***.
- **Composition** is actually a special case of the aggregation relationship.



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;
  …
}
```

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.

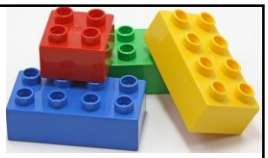
6

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.

7

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.

8

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.

9

Wrapper (غلاف) Classes

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

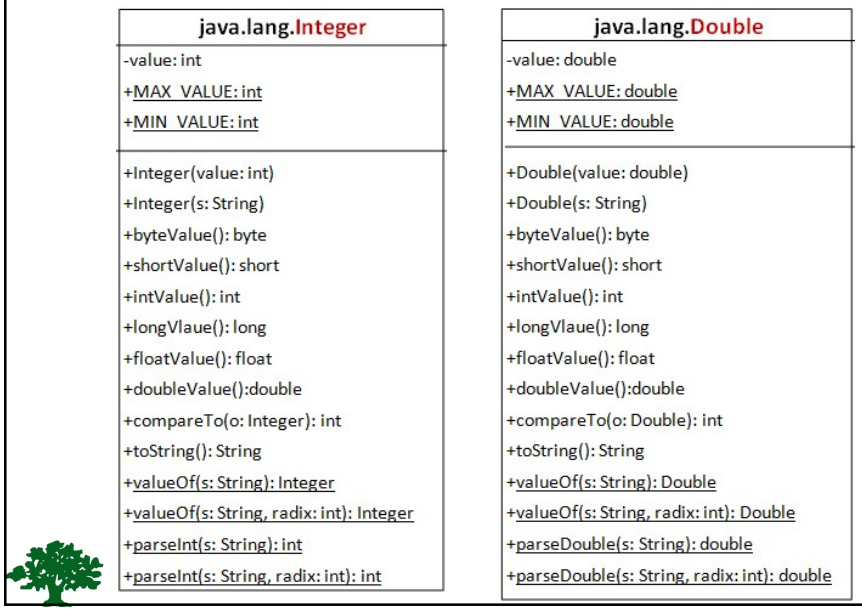
NOTE:

(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.

10

The Integer and Double Classes



11

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***.

```
import java.math.BigInteger;
```

12

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);
```

13