INTE2512 Object-Oriented Programming

OOP 1

Quang Tran

RMIT
UNIVERSITY

Outline

- Classes vs Objects
- Attributes
- Methods
- Constructors
- Modifiers
- Encapsulation
- Packages
- Class Relationships
- Dependency
- Association
- Aggregation
- Composition

Classes vs Objects

- Classes and objects are the core concepts in OOP
- Examples:

Class	Objects
Fruit	Apple Banana Mango

Class	Objects
Car	Toyota Volvo Audi

- A class is a template for objects, and an object is an instance of a class
- An object inherits the attributes and methods from the class that is used to create the object

Attributes

- Attributes are the data of a class
- They are variables defined in a class

```
// File: Point.java
public class Point {
    double x;
    double y;
}
```

Attributes

- Create objects and access attributes

```
// File: Main.java
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point();  // create a new object
        Point p2 = new Point();
        p1.x = 1; p1.y = 2;
        p2.x = 3; p2.y = 4;
        System.out.println("[" + p1.x + ", " + p1.y + "]");
        System.out.println("[" + p2.x + ", " + p2.y + "]");
    }
}
```

Methods

- A method is a block of code that is used to perform a specific task
- Methods are defined in a class

```java
// File: Point.java
public class Point {
    double x, y;
    void showPoint() {
        System.out.println("[" + x + ", " + y + "]");
    }
}
```

Methods

- Methods can be called as follow:

```
// File: Main.java
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point();
        Point p2 = new Point();
        p1.x = 1; p1.y = 2;
        p2.x = 3; p2.y = 4;
        p1.showPoint();
        p2.showPoint();
    }
}
```

Constructors

- A constructor is a **special method** that is used to initialize the values of object attributes
- **this** is a **reference** to the current object

```
public class Point {
    double x, y;
    public Point(double x, double y) {
        this.x = x;       // this.x is the attribute x
        this.y = y        // where x is the local variable x
    }
    void showPoint() {
        System.out.println("[" + x + ", " + y + "]");
    }
}
```

Constructors

- Now we can create objects with the constructor :

```java
// File: Main.java
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(3, 4);
        p1.showPoint();
        p2.showPoint();
    }
}
```

Modifiers

- There are two group of modifiers
  - Access modifiers
  - Non-access modifiers

Access Modifiers

- For classes, you can use either **public** or *default* (no modifier):

Modifier	Description
public	The class is accessible by any other class
default	The class is only accessible by classes in the same package

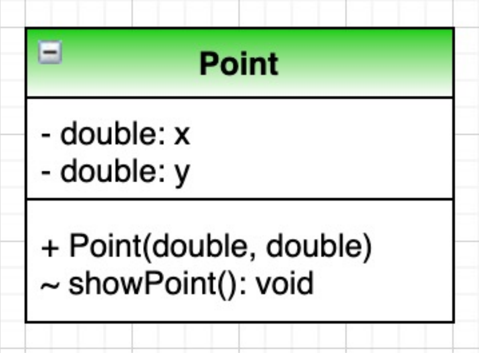
- For attributes, methods and constructors, you can use:

Modifier	Description
public	The code is accessible for all classes
protected	The code is accessible by subclasses, or classes in the same package
default	The code is only accessible by classes in the same package
private	The code is only accessible within the class it is declared

Access Modifiers

- Let’s apply access modifiers now

```
// File: Point.java
public class Point {
    private double x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y
    }
    void showPoint() {          // default access modifier
        System.out.println("[" + x + ", " + y + "]");
    }
}
```



Point

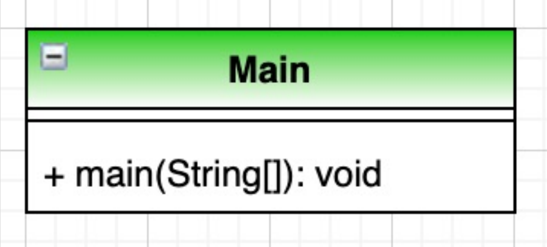
- double: x
- double: y

+ Point(double, double)
~ showPoint(): void

Access Modifiers

- Now we can create objects with the constructor :

```
// File: Main.java in the same directory with Point.java
public class Main {
    public static void main(String[] args) {
        Point p = new Point(1, 2);
        p.showPoint();
        System.out.println(p.x + ", " + p.y); // error
    }
}
```



Non-Access Modifiers

- For classes, you can use either **final** or **abstract**:

Modifier	Description
final	The class cannot be inherited by other classes
abstract	The class cannot be used to create objects

- For attributes, methods and constructors, you can use one of the following:

Modifier	Description
final	Attributes and methods cannot be overridden/modified
static	Attributes and methods belongs to the class, rather than an object
abstract	Can only be used on methods in an abstract class
transient	Attributes and methods are skipped when serializing the object containing them
synchronized	Methods can only be accessed by one thread at a time
volatile	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Encapsulation

- Encapsulation is to make sure that the "sensitive" data of a class is hidden from other classes
- To achieve this, you must:
    - Declare the attributes as **private**
    - Provide public **get** and **set** methods, *when needed only*, to access and update the value of a **private** attribute

Encapsulation

```
public class Person {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name };
}
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "John";              // error
        System.out.println(person.name);   // error
    }
}
```

Encapsulation

```
public class Person {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name };
}
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("John");               // access
        System.out.println(person.getName()); // update
    }
}
```

Packages

- Package is a mechanism to group related classes together
- You can think of a package as a **directory** containing related classes
- Packages allow to structure better maintainable code and to avoid name conflicts
- There are two group of packages:
    - Built-in packages (from the Java API)
    - User-defined packages (your own-created packages)

Packages

- To use a class or a package, use the **import** keyword

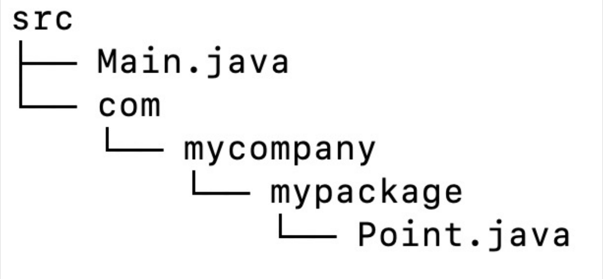
```
import package.name.ClassName;
import package.name.*;

import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter your name");
        String name = scan.nextLine();
        System.out.println("Hello " + name);
    }
}
```

Packages

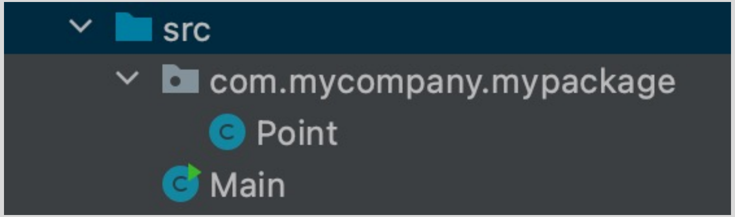
- Java uses a directory to store the Java files in a package
- To create a package, use the **package** keyword

```
package com.mycompany.mypackage;
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void showPoint() {
        System.out.println("[" + x + ", " + y + "]");
    }
}
```



```
src
├── Main.java
└── com
    └── mycompany
        └── mypackage
            └── Point.java
```

Packages



src

com.mycompany.mypackage

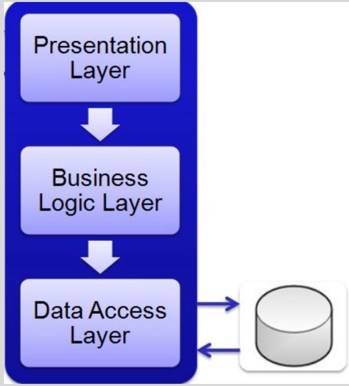
Point

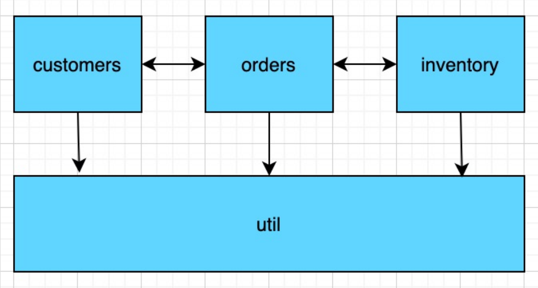
Main

```
import com.mycompany.mypackage.Point;
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(3, 4);
        p1.showPoint();
        p2.showPoint();
    }
}
```

Packages

- Two common approaches to organize your classes into packages
  - By layers: organize your classes based on the app layers
    - Example: presentation, business logic, data access
  - By functionality: organize your classes based on the app functionalities
    - Example: customers, orders, inventory, util
- You can also combine these two approaches



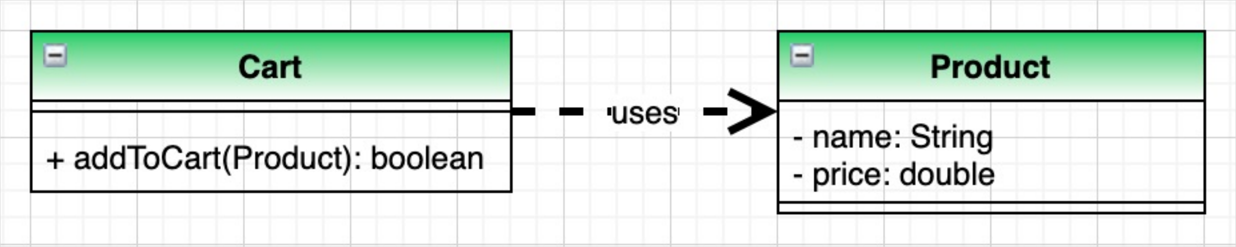


Class Relationships

- To construct a program, classes are designed to relate to each other in one of the following relationships:
  1. Dependency
  2. Association
  3. Aggregation
  4. Composition
  5. Inheritance (Generalization)
  6. Implementation (Realization)

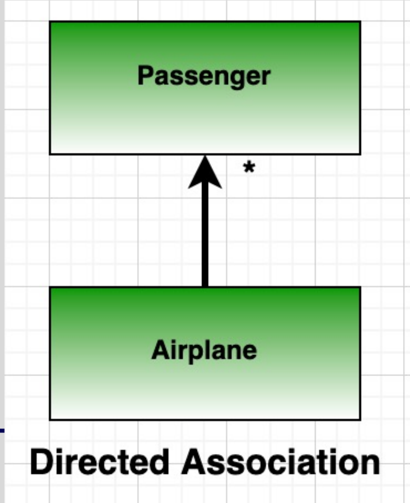
Dependency

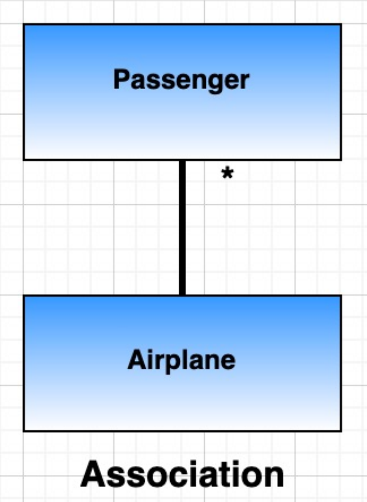
- Dependency represents a "uses" relationship in which a client class "uses" a supplier class
- Dependency indicates that the client class may:
  - use the supplier class as a parameter / local variable / return value in one or more methods
  - call one or more methods of the supplier class



Association

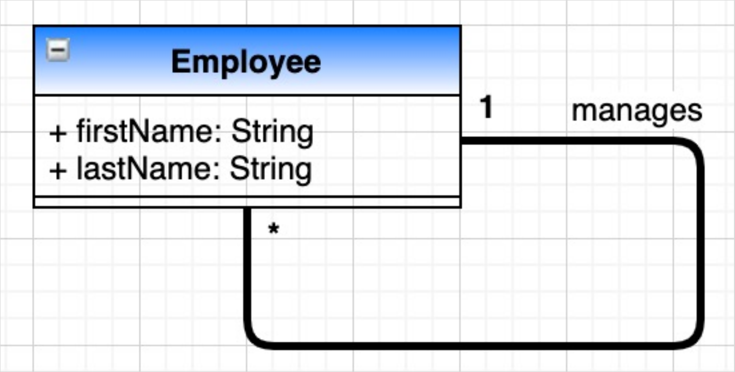
- Association can represent any relationship between two classes
- Association indicates that a class may:
  - use the other class as one or more attributes
  - call one or more methods of the other class





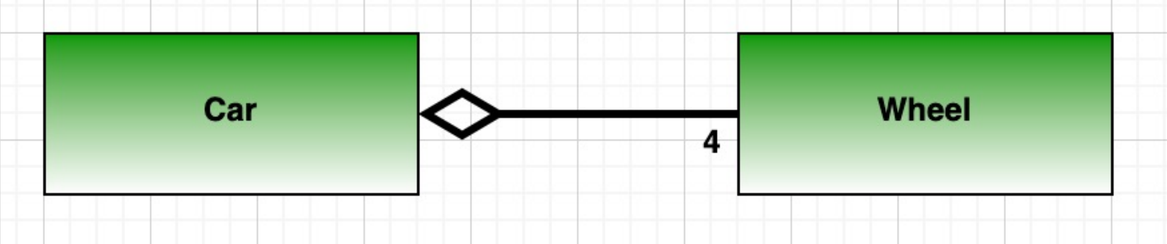
Reflexive Association

- Reflexive association is used when a class is related to itself
- In this example, an Employee object has a role Manager thus manages many other Employee objects



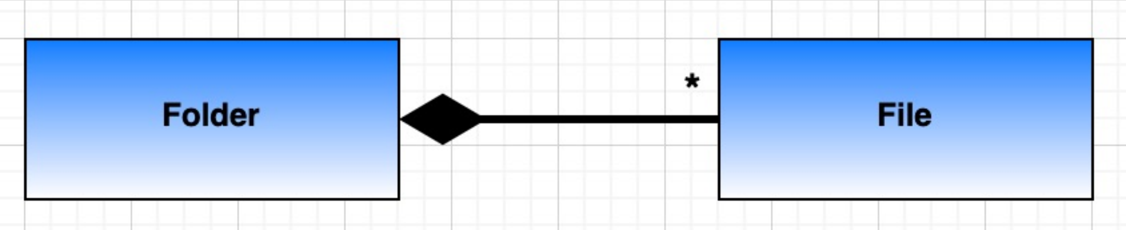
Aggregation

- Aggregation is a **strong** association to represent a "has-a" relationship
- In aggregation, if the container object is deleted, the contained objects can stay

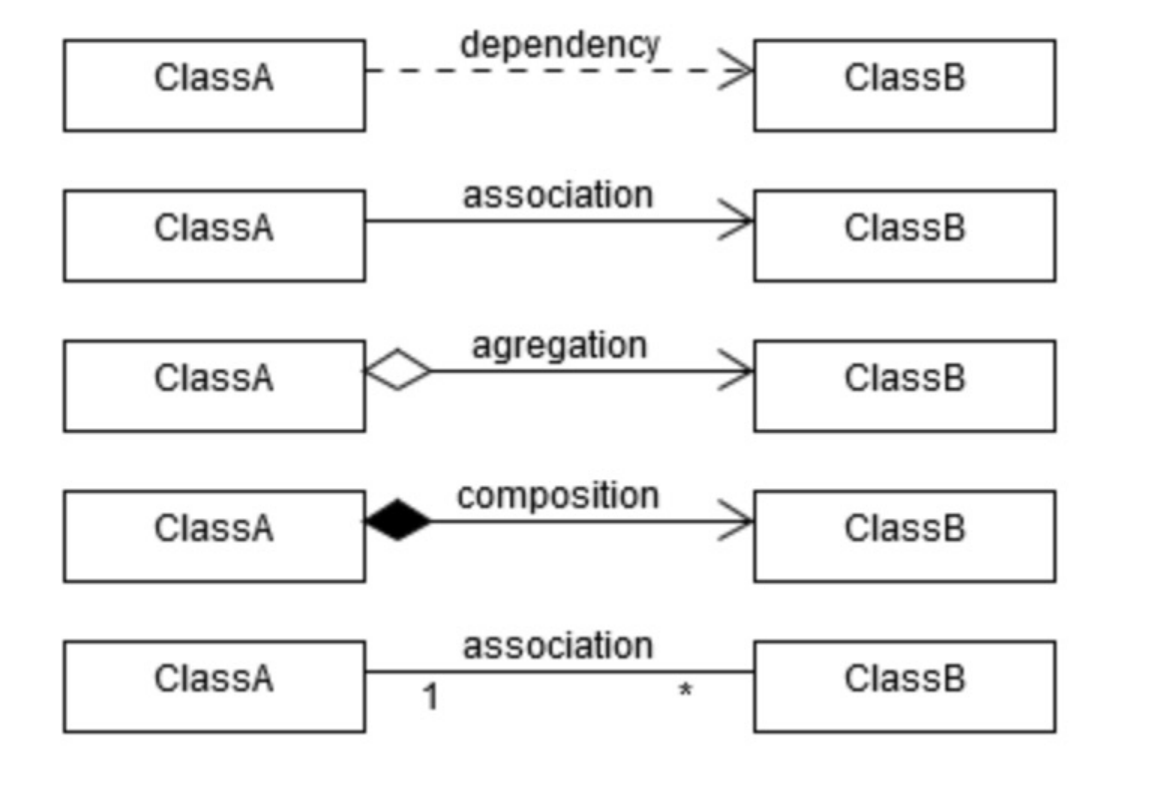


Composition

- Composition is a **strong** aggregation to represent a "contains" relationship
- In composition, if the container object is deleted, the contained objects must be deleted as well



Relationship Summary



References

1. D. Y. Liang, Intro to Java Programming, 10th edition, chapter 1-5, 2015.
2. W3Schools, Java Tutorial, 2021.
3. TutorialsPoint, Java Tutorial, 2021.
4. Jenkov, Java Tutorial, page 1-20, 2021.
5. Oracle Corporation, Java 11 API Specification, 2019.
6. IBM Corporation, The class diagram: an introduction to structure diagrams in UML 2, 2004.