Sabancı Üniversitesi

Object Oriented Programming with Java

04 - Object Oriented Programming Concepts

Sabancı Üniversitesi

OOP Rule – Composition & Aggregation

- A class can have references to objects of other classes as members
- Sometimes referred to as a *has-a* relationship

```
public class Address{
        /*…local variables,setters/getters,constructor…*/
}

public class Student{
        private String name;
        private Address _address;
        public Student(String name){
                this.name = name;
        }
}
```

Sabancı Üniversitesi

Difference of Composition and Aggregation

- **Aggregation** implies a relationship where the child can exist independently of the parent.
  - Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
- **Composition** implies a relationship where the child cannot exist independent of the parent.
  - Example: House (parent) and Room (child). Rooms don't exist separate to a House.

Sabancı Üniversitesi

Enumerations

- Declared with an **`enum`** declaration
  - A comma-separated list of **`enum`** constants
  - Declares an **`enum`** class with the following restrictions:
    - **`enum`** types are implicitly **`final`**
    - **`enum`** constants are implicitly **`static`**
    - Attempting to create an object of an **`enum`** type with **`new`** is a compilation error
- **`enum`** constants can be used anywhere constants can
- **`enum`** constructor
  - Like class constructors, can specify parameters and be overloaded
- Can be declared as a member of a class or independently
- Declaration without a constructor:

```
public enum CoffeSizes{SMALL,MEDIUM,LARGE,GRANDE}
```

Enumerations with Constructors

```
public enum CoffeeSizes {

        MEDIUM("Medium",3),
        SMALL("Small",1),
        LARGE("Large",5);

        private String name;
        private double litres;

        private CoffeeSizes(String name,
                            double litres) {
                this.name = name;
                this.litres = litres;
        }

        public String getName() {
                return name;
        }

        public double getLitres() {
                return litres;
        }
}
```

```
public static void main(String[] args) {
for (CoffeeSizes sizes : CoffeeSizes.values()) {
          System.out.println("Size: " +
sizes.getName() + ", Capacity: " + sizes.getLitres());
          }
}
```

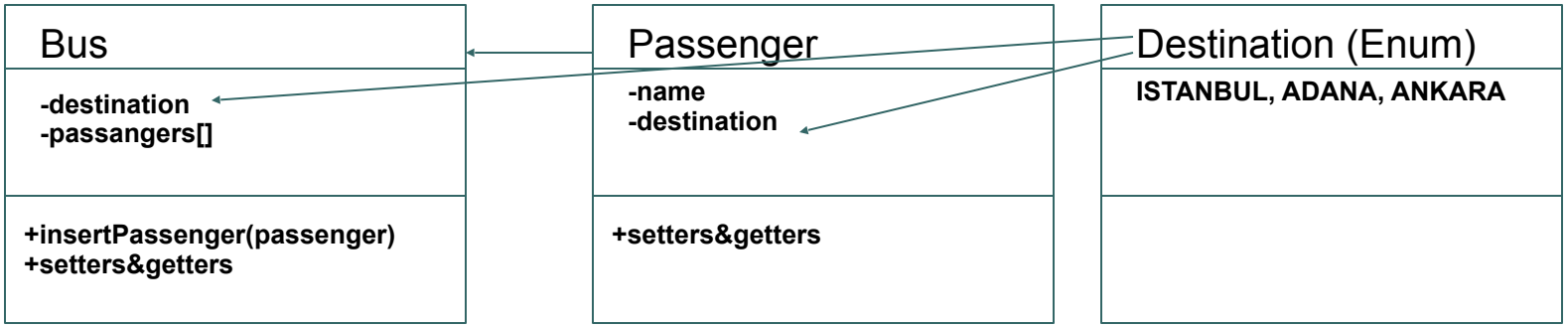
Usage

Definiton

It is a syntax error to declare enum constants after the enum type's constructors, fields and methods in the enum declaration

Lab: Bus Reservation

- Create a Bus and a Passenger object
- Create an enum called Destination
- In main method insert passengers into the Bus, if the capacity is full or destination is different give information



Main method content example:

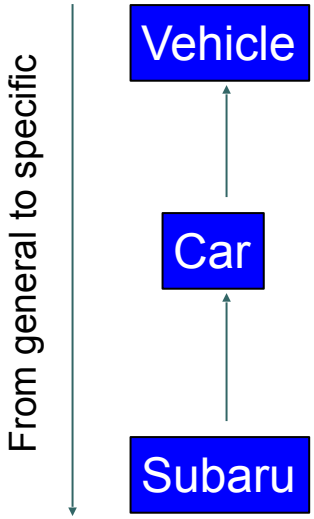
```
bus1.insertPassenger(new Passenger("Ali", Destination.ADANA));
bus1.insertPassenger(new Passenger("Veli", Destination.ANKARA));
bus1.insertPassenger(new Passenger("Mehmet", Destination.ADANA));
```

Console Output:

```
Passenger inserted
Destinations do not match
Bus is full
```

OOP Rule: Inheritance

- Software reusability
- Create new class from existing class
  - Absorb existing class's data and behaviors
  - Enhance with new capabilities
- Subclass extends superclass
  - Subclass
    - More specialized group of objects
    - Behaviors inherited from superclass
      - Can customize
    - Additional behaviors
- Referred as *is-a* relationship




```
public class Vehicle{/*Class body*/}
public class Car extends Vehicle {/*Class body*/}
public class Subaru extends Car{/*Class body*/}
```

Sabancı Üniversitesi

Class Hierarchy

- Direct superclass
  - Inherited explicitly (one level up hierarchy)
- Indirect superclass
  - Inherited two or more levels up hierarchy
- Single inheritance
  - Inherits from one superclass
- Multiple inheritance
  - Inherits from multiple superclasses
    - Java does not support multiple inheritance

Sabancı Üniversitesi

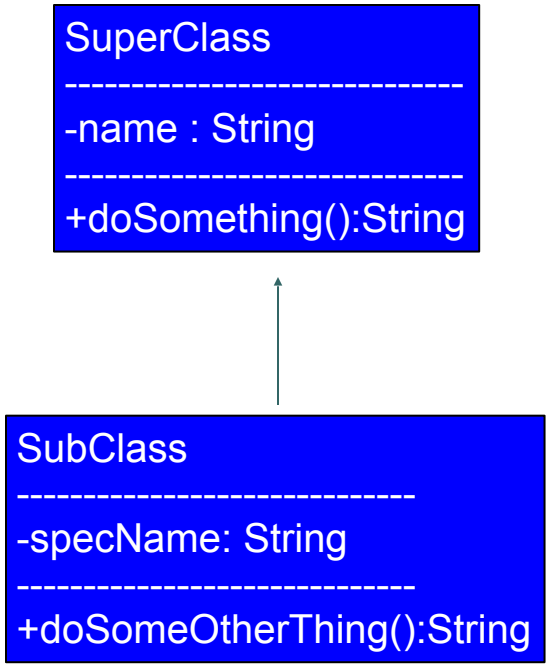
Superclasses & Subclasses

- Object of one class “is an” object of another class
  - Example: Rectangle is quadrilateral.
    - Class `Rectangle` inherits from class `Quadrilateral`
    - `Quadrilateral`: superclass
    - `Rectangle`: subclass
- Superclass typically represents larger set of objects than subclasses
  - Example:
    - superclass: `Vehicle`
      - Cars, trucks, boats, bicycles, …
    - subclass: `Car`
      - Smaller, more-specific subset of vehicles

Inheritance Hierarchy

- Inheritance relationships: tree-like hierarchy structure
- Each class becomes
  - superclass
    - Supply members to other classes

  OR
  - subclass
    - Inherit members from other classes
- `private` members aren't accessible from the subclasses
- `final` classes cannot be subclassed
- Every class is a subclass of `Object`



The subclass has all the public members of the superclass plus its own members

Sabancı Üniversitesi

`protected` & `default` access in Inheritance Relationship

- `protected` access
  - `protected` members of the superclass are accessible by subclass members even if the inheriting class isn't in the same package as the inherited class
  - Subclass access to superclass member
    - Keyword `super` and a dot (.)
- `default` access
  - members of the superclass are not accessible by subclass members if the inheriting class isn't in the same package as the inherited class

Sabancı Üniversitesi

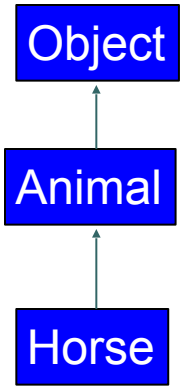
Constructors in Inheritance Relationship

- Constructors are not inherited in the subclasses
- First task of any subclass constructor is to call its direct superclass's constructor
- If the code does not include an explicit call to superclass constructor, Java implicitly calls the superclass's no-argument constructor
- To call superclass's constructor explicitly use `super()`
- To call one of the constructors in a class use `this()`
- Constructor call (`super()` or `this()`) should be the first statement in the constructor

Constructors in Inheritance Relationship

- Constructor Chaining

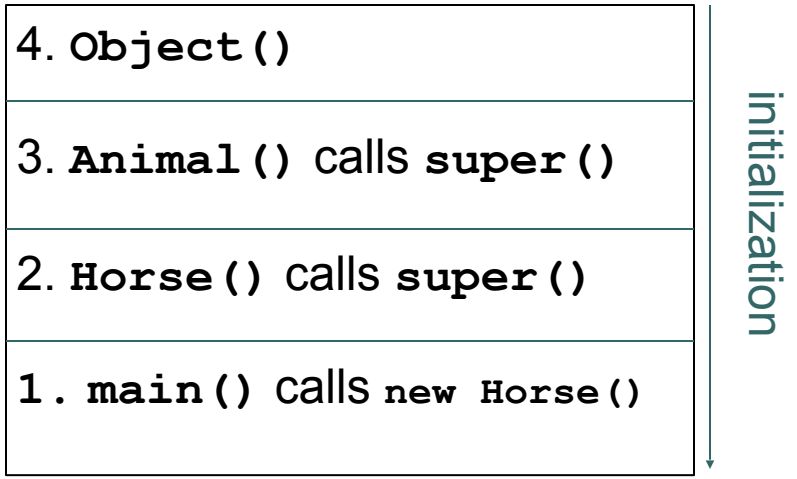
Object Hierarchy:



Before execution every constructor calls its superclass's constructor implicitly or explicitly

Main method calls `new Horse()`

Constructors on the call stack:



Sabancı Üniversitesi

Method Overriding

- The behavior of a method in the superclass can be modified in the subclass
- Example:

```
public class Animal{
        public String makeNoise(){
                return "No Sound";
        }
}

public class Lion extends Animal{
        public String makeNoise(){
        return "Roarrr!";
        }
}
```

Sabancı Üniversitesi

Method Overriding

- Rules for overriding methods:
  - The argument list must exactly match that of the overridden method
  - The return type must be the same as, or a subtype of, the return type in the original method
  - Access level cannot be more restrictive (but can be less restrictive)
  - `private`, `final` and `static` methods cannot be overridden *(but can be redefined)*

Sabancı Üniversitesi

OOP Rule: Polymorphism

- A reference variable can refer to any object of the same type as the declared reference, or -this is the big one! - *it can refer to any subtype of the declared type*.
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing
- A reference variable can be declared as a class type or an interface type (*more on interfaces later*)


```
Animal myDog   =    new Dog();
```

OOP Rule: Polymorphism

- When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
- The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked

```
class Animal{
        public void makeSound(){
                System.out.println("No Sound");
        }
}

class Dog extends Animal{
        public void makeSound(){
                System.out.println("Dog sound");
        }
}

class TestClass{
        public static void main(String[] args){
                Animal myDog = new Dog();
                myDog.makeSound();
        }
}
```

What will be the output?

Sabancı Üniversitesi

Casting Objects

- Allows us to tell the compiler to use a different type within the object's class hierarchy
  - Casting to a more generalized type (upcast)

```
//Employee extends Person
Employee e = new Employee(); //Subclass
Person p; //superclass
p = e; // implicit cast occurs, same as Person p = (Person)e;
```

  - The cast from a supertype to subtype is valid as long as it is made towards the right type

```
void someMethod(Person person){
        Employee emp = (Employee)person;
}
```

If the person reference was referring to another subtype instead of employee at runtime the cast would fail

Sabancı Üniversitesi

Using `instanceof` operator for Downcasting

- If a downcast is not valid, a `ClassCastException` will be thrown at runtime

```
Animal animal = new Cat();
Dog myDog = (Dog)animal; //compiler allows this but will throw an exception at runtime
```

- The `instanceof` operator tests the type of the object referenced (not the declared type of the reference variable itself)

```
Animal animal = new Cat();
if(animal instanceof Dog){ //will return false, no runtime exception
Dog myDog = (Dog)animal; //will not execute as animal isn't an instance of Dog
}
```

Sabancı Üniversitesi

The `Object` Class

- Every class that is defined in Java is a subclass of **`Object`** **`(java.lang.Object)`**
- Well designed classes should always override
  - **`equals`** → allows class to test instance equality
  - **`toString`** → allows class to convert itself to a **`String`**

Sabancı Üniversitesi

Separating Capability from Implementation

- An object has an interface and implementation
  - The programming interface is the set of visible methods
  - The implementation is hidden within those methods
- An object can safely change its implementation provided it does not change its programming interface
- Java provides a formal way of representing these programming interface contracts
  1. *Interface*
  2. *Abstract class*

Sabancı Üniversitesi

Abstract Classes

- An abstract class is a class with one or more abstract methods
- An abstract method is a method with no body
- Abstract methods should be implemented in the first concrete subclass
- An abstract class cannot be instantiated
- If a class has at least one abstract method then the class should be abstract too

Sabancı Üniversitesi

Abstract Class Example

```
public abstract class Shape{
        private int startX;
        private int startY;

        /.. setters & getters /

        public abstract void draw();

}

public class Circle extends Shape{

        public void draw(){
                …
        }
}
```

No Method Body!

Concrete class

Abstract Method implementation

- We can create an object reference to an abstract class:

```
Shape circle = new Circle();
circle.draw();
```

Polymorphism

Sabancı Üniversitesi

Lab: The Pen Revisited

- Think about how to shorten the code previously written (The Pen Lab project) using polymorphism and casting
- Hint-1: Create a shape class and extend Circle&Rectangle from it
- Hint-2: You have several choices:
  - Use a single drawShape() method in Pen class and use instanceof operator
  - Use a single drawShape() method in Pen class and put draw() method in Shape class

Sabancı Üniversitesi

Interfaces

- Similar to abstract class
- All interfaces are implicitly `public` and `abstract`
- All the methods of an interface are `public` and `abstract`
- All interface variables are implicitly `public`, `static` and `final`

```
public interface Drawable {
        public void draw();
        public void clear();
}
```

Sabancı Üniversitesi

Implementing an Interface

- A class implementing an interface is said to be a subtype of its interface
- The subtype class must implement all of the interface methods
- `implements` keyword is used, and a class can implement more than one interface and extend from “a” class

```
public class Circle implements Drawable{
       public void draw(){ … }
       public void clear(){ … }

}
```

Sabancı Üniversitesi

More on Interfaces

- Interfaces can *extend* from other interfaces (more than once) but not from classes

```
public interface extends Interface1, Interface2{}
```

- A reference of an object can be an interface type (polymorphism)

```
Shape circle = new Circle();
Shape square = new Square();

drawShape(circle);
drawShape(square);

void drawShape(Shape shape){
       shape.draw();
}
```

Sabancı Üniversitesi

`default` Methods in Interfaces

- Java 8 has added default methods as a new feature

```
public interface SalesCalcs {
... // A number of lines omitted
  public default void printItemReport(){
    System.out.println("--" + this.getName() + " Report--");
    System.out.println("Sales Price: " + this.calcSalesPrice());
    System.out.println("Cost: " + this.calcCost());
    System.out.println("Profit: " + this.calcProfit());
  }
}
```

- `default` methods:
  - Are declared by using the keyword “default”
  - Are fully implemented methods within an interface
  - Provide useful inheritance mechanics

Sabancı Üniversitesi

`default` Method: Example

```
SalesCalcs[] itemList = new SalesCalcs[5];

    itemList[0] = new CrushedRock(12, 10, 50);
    itemList[1] = new CrushedRock(8, 6, 10);
    itemList[2] = new RedPaint(10, 8, 25);
    itemList[3] = new Widget(6, 5, 10);
    itemList[4] = new Widget(14, 12, 20);

    System.out.println("==Sales Report==");
    for(SalesCalcs item:itemList){
      item.printItemReport();
    }
```

Sabancı Üniversitesi

`static` Methods in Interfaces

- Java 8 allows `static` methods in an interface. So it is possible to create helper methods like the following:

```
public interface SalesCalcs {
... // A number of lines omitted
  public static void printItemArray(SalesCalcs[] items){
    System.out.println(reportTitle);
      for(SalesCalcs item:items){
        System.out.println("--" + item.getName() + " Report--");
        System.out.println("Sales Price: " + item.calcSalesPrice());
        System.out.println("Cost: " + item.calcCost());
        System.out.println("Profit: " + item.calcProfit());
      }
}
```

Sabancı Üniversitesi

Garbage Collection and Method `finalize`

- Garbage collection
  - JVM marks an object for garbage collection when there are no more references to that object
  - JVM's garbage collector will retrieve those objects memory so it can be used for other objects
- `finalize` method
  - All classes in Java have the `finalize` method
    - Inherited from the `Object` class
  - `finalize` is called by the garbage collector when it performs termination housekeeping
  - `finalize` takes no parameters and has return type `void`
- `System.gc()` method can be called for garbage collection to collect the *null* references

Sabancı Üniversitesi

Modifiers & Code Elements Summary

	Class	Interface	Constructor	Method	Field	Inner Class	Nested Interface	Floating Block
abstract	✔	✔	x	✔	x	✔	✔	x
final	✔	x	x	✔	✔	✔	x	x
native	x	x	x	✔	x	x	x	x
private	x	x	✔	✔	✔	✔	✔	x
protected	x	x	✔	✔	✔	✔	✔	x
public	✔	✔	✔	✔	✔	✔	✔	x
static	x	x	x	✔	✔	✔	✔	✔
synchronized	x	x	x	✔	x	x	x	✔
transient	x	x	x	x	✔	x	x	x
volatile	x	x	x	x	✔	x	x	x
strictfp	✔	✔	x	✔	x	✔	✔	x