Object Oriented Programming (OOP)

Lecture4: Inheritance & Polymorphis

Prepared by:
Mohamed Mabrouk

Those slides are based on slides by:
Dr. Sherin Mousa and Dr. Sally Saad

Lecture Outline

- Class reusability
- Inheritance
- Method overriding
- Abstract class/method
- Abstract classes and methods
- Polymorphism

Class Reusability

Class reusability

- Is simply reusing a class in another class
- Has two forms COMPOSITION and INHERITANCE
- Composition is also called *has-a* → placing a reference/object of a class in another class
- For instance, relation between class Employee and Department

Class reusability

```
public class Department {
   Employee[] employees;
}
```

OR

```
public class Employee {
   Department department;
}
```

Class reusability

- Inheritance is also called *is-a* → extending a class with another class
- For instance, relation between class Employee and Person

Class reusability

```
public class Employee extends Person {

}
```

Inheritance

Inheritance

- Means that a new class (called child class or subclass) inherits from an existing class (called parent or super class) → It inherits all its members and characteristics
- Can add/modify parent class functionality to fit its requirements
- One of the main pillars of OOP

Inheritance

```java
public class Person {
	private String name;
	String address;
	public Person(String name, String address) {
		this.name = name;
		this.address = address;
	}
}
```

```java
public class Employee extends Person {
	private float salary;
}
```

Subclass Constructors

- Subclass has to have a constructor similar to that of base class
- Subclass constructor has to call base class constructor directly or indirectly → constructor calls another constructor that in turn calls `super`
- `super()` can be used to call base class constructor → otherwise a compilation error

Subclass Constructors

- Call to `super` *<u>MUST BE THE FIRST STATEMENT</u>*
- Super class constructor must be called to ensure that all base class members are initialized
- Super class members are also members of subclass, so they have to be initialized first

Method Overriding

Method Overriding

- Access modifier for an overriding method can be same or more, but not less, access than the overridden method
- For instance, if base class method's access modifier is `protected` → the access modifier of child class's method can be `protected` or `public` but NOT `private`
- Any method defined in Java is ***OVERRIDABLE BY DEFAULT*** unless it defined as `final`

Method Overriding

- Second form of polymorphism (method overloading is the other form)
- Method of subclass has the exact same signature as that of the super class → same name, same parameters and same return type
- When a method is overridden the default behavior or base class can be adapted

Overloading versus Overriding

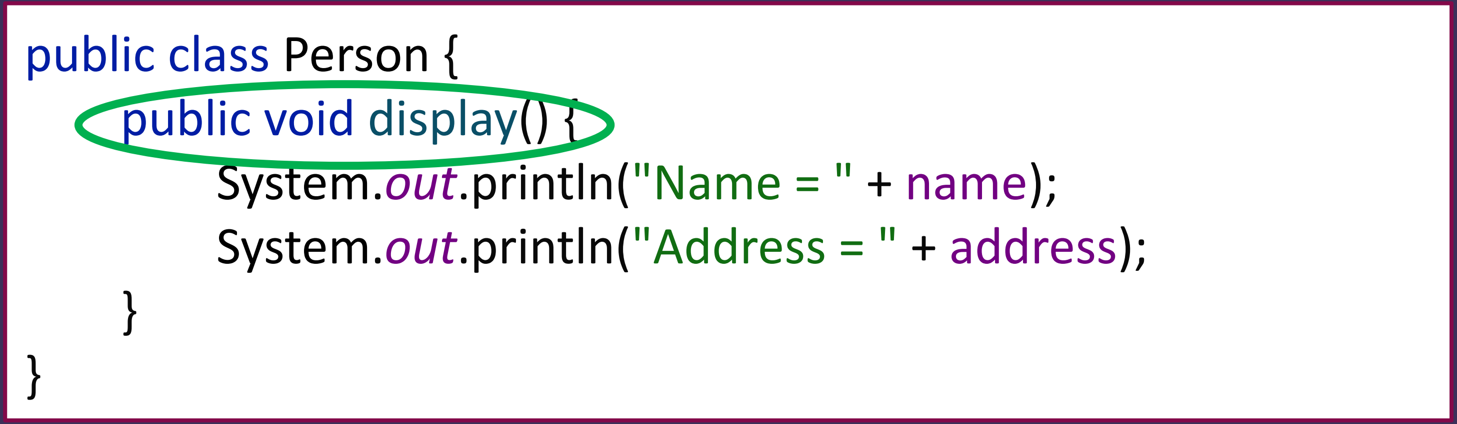
- Overloading is between methods of the same class
- Same method name but with different number of parameter or parameter types but not return type
- Overriding is between methods of a base and child classes
- It is the exact same method name and parameters and return type but with different behavior → parent class logic is modified/adapted

Overriding Example

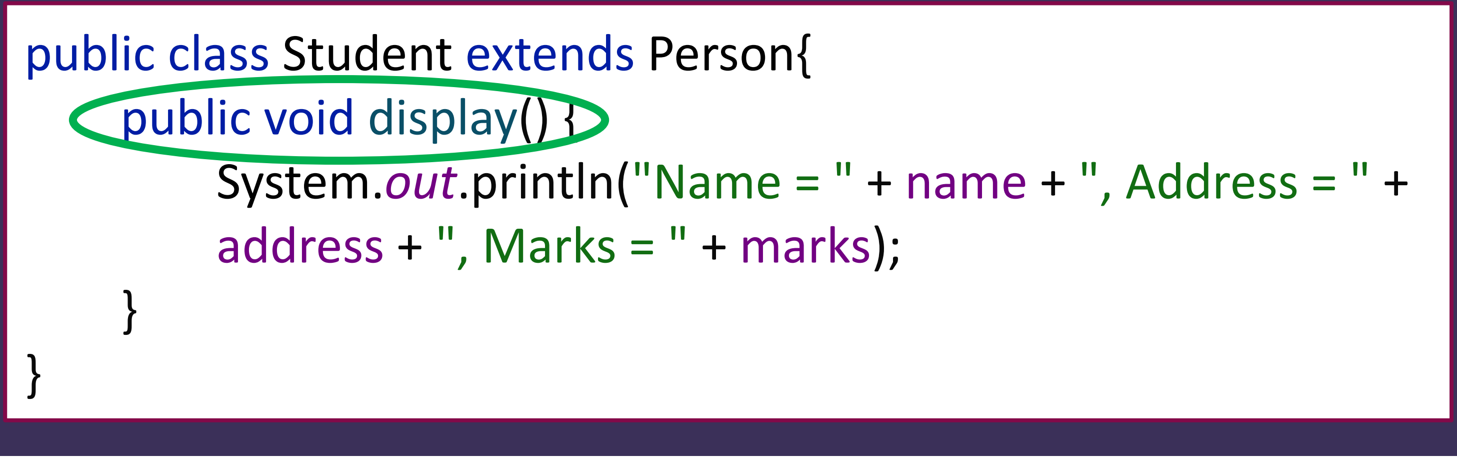
```
public class Person {
        public void display() {
                System.out.println("Name = " + name);
                System.out.println("Address = " + address);
        }
}
```

```
public class Student extends Person{
        public void display() {
                System.out.println("Name = " + name + ", Address = " +
                address + ", Marks = " + marks);
        }
}
```

Overriding Example



```
public class Person {
    public void display() {
        System.out.println("Name = " + name);
        System.out.println("Address = " + address);
    }
}
```



```
public class Student extends Person{
    public void display() {
        System.out.println("Name = " + name + ", Address = " +
        address + ", Marks = " + marks);
    }
}
```

Same method signature but different behavior

Inheritance Summary

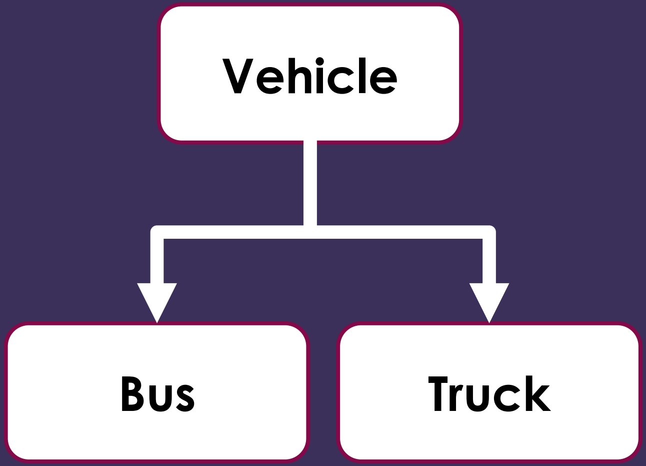
- Enables code reusability
- A class inherits (extends) another class which has similar but not exact behavior
- Subclass can add new functionalities and/or adapt existing ones
- It inherits all non-private members (fields and methods)
- A class can have *EXACTLY ONE PARENT* class

Abstract Class/Method

Abstact Class

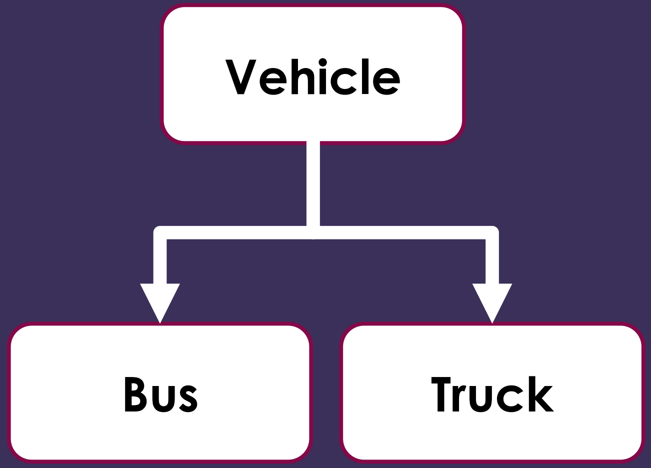
- A class that does not have any concrete functionality by itself
- It *MUST BE INHERITED* (extended) to have a meaning
- Is called abstract class
- *CAN NEVER BE INSTANTIATED*

Abstact Class Example



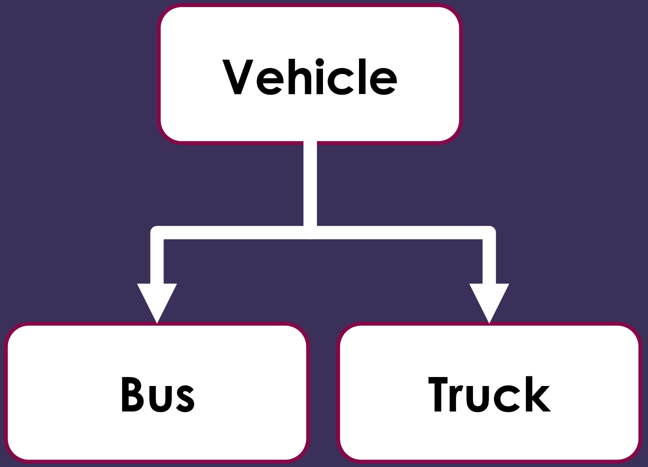
Abstact Class Example

- Vehicle class can be defined abstract



Abstact Class Example

- Vehicle class can be defined abstract
- It provides basic functionality of any moving vehicle

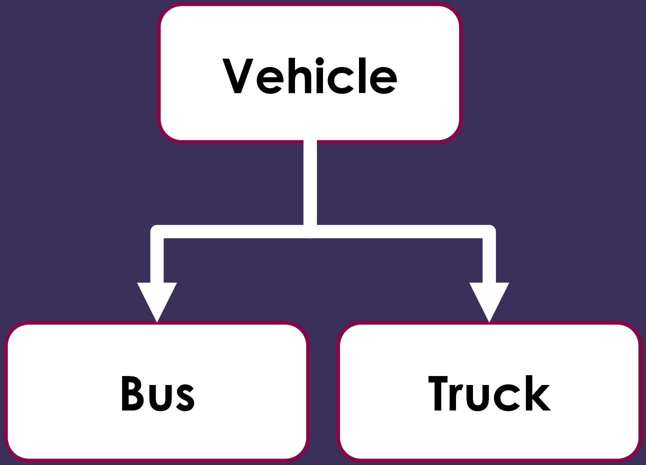

Vehicle

Bus

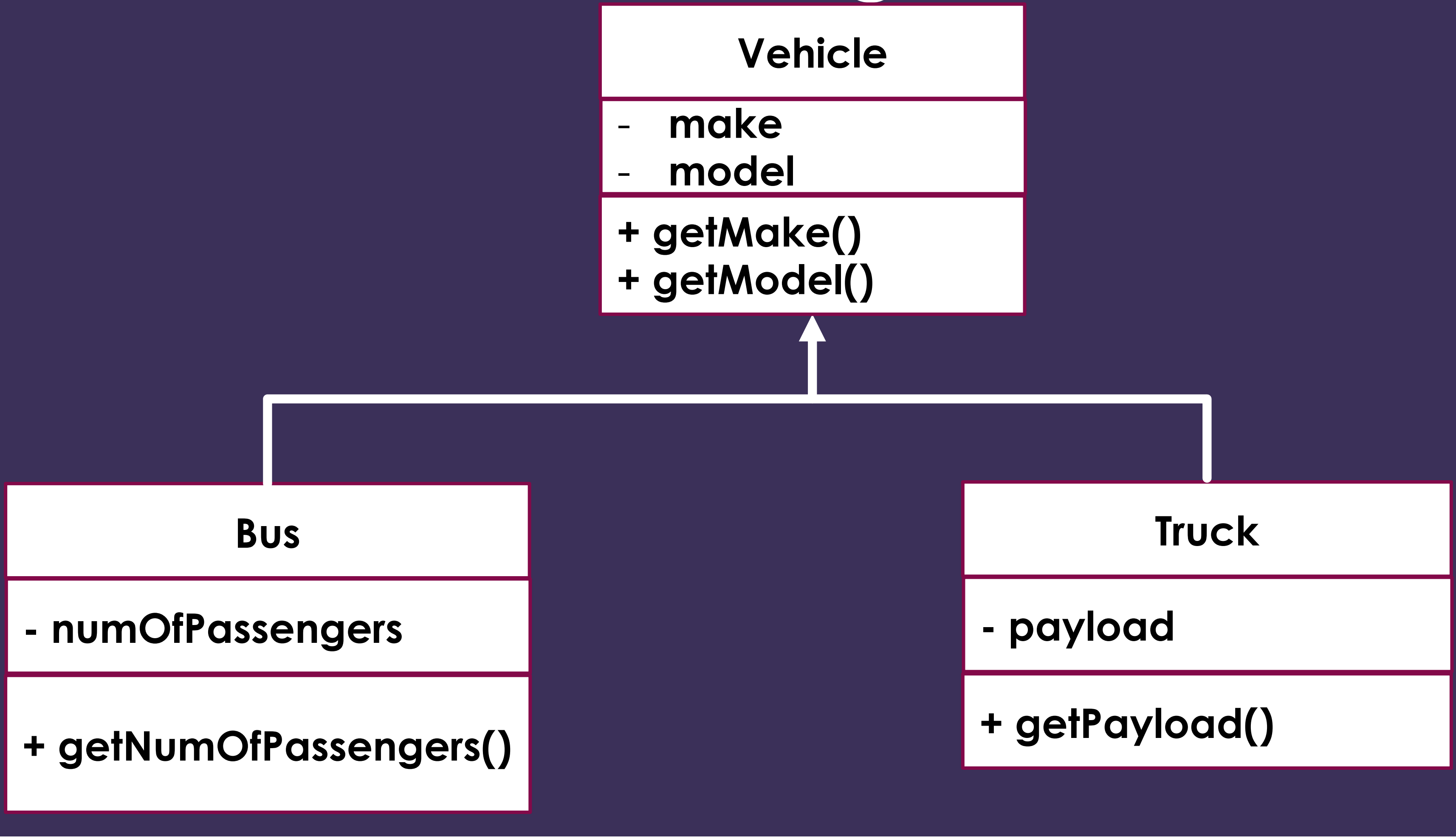
Truck

Abstact Class Example

- Vehicle class can be defined abstract
- It provides basic functionality of any moving vehicle
- Cannot be used by itself, rather one of its children can be used and instantiated



Abstact Class UML Diagram



How to Define an Abstract Class

```
public abstract class Vehicle {
	private String make;
	private String model;

	public Vehicle(String make, String model){
		this.make = make;
		this.model = model;
	}
	public String getMake(){
		return make;
	}
	 public String getModel() {
		return model;
	}
}
```

How to Define an Abstract Class

```
public abstract class Vehicle {
    private String make;
    private String model;

    public Vehicle(String make, String model){
        this.make = make;
        this.model = model;
    }
    public String getMake(){
        return make;
    }
     public String getModel() {
        return model;
    }
}
```

Subclass of an Abstract Class

```
public class Truck extends Vehicle {
    private float payload;

    public Truck(String make, String model) {
        super(make, model);
    }
    public Truck(String make, String model, float payload) {
        this(make, model);
        this.payload= payload;
    }

    public float getPayload() {
         return payload;
    }
}
```

Abstact Method

- A method declared in base class with full signature but *HAS NO BODY*
- It has to be overridden in subclasses
- If a class has one or abstract methods → The class *MUST ALSO BE ABSTRACT*
- Abstract classes can contain both concrete (non-abstract) and abstract methods

Abstact Method Example

- Can you think of an abstract method to be added to our vehicle hierarchy???

Abstact Method Example

- What if we add a method called “clear” that clears the vehicle?

Abstact Method Example

- What if we add a method called “clear” that clears the vehicle?
- Does it depend on the type of car, i.e. does it differ in truck from that of bus?

Abstact Method Example

- What if we add a method called “clear” that clears the vehicle?
- Does it depend on the type of car, i.e. does it differ in truck from that of bus?
- Yes, in truck you have to clear payload, i.e. set it to 0, whereas in bus you have to set numOfPassengers to 0

Abstract Method Example

```
public abstract class Vehicle {
	public abstract void clear();
}
```

Abstract Method Example

```
public abstract class Vehicle {
	public abstract void clear();
}
```

```
public class Bus extends Vehicle{
	public void clear(){
		this.numOfPassengers = 0;
	}
}
```

Abstract Method Example

```
public abstract class Vehicle {
        public abstract void clear();
}
```

```
public class Bus extends Vehicle{
        public void clear(){
                this.numOfPassengers = 0;
        }
}
```

```
public class Truck extends Vehicle{
        public void clear(){
                this.payloda = 0;
        }
}
```

Abstact Method

- Constructors and static methods cannot be declared abstract → why???

Abstact Method

- Constructors and static methods cannot be declared abstract → why???
- Constructors are not inherited

Abstact Method

- Constructors and static methods cannot be declared abstract → why???
- Constructors are not inherited → we should call `super` for them to get invoked

Abstact Method

- Constructors and static methods cannot be declared abstract → why???
- Constructors are not inherited → we should call `super` for them to get invoked
- Static methods are for the whole class

Abstact Method

- Constructors and static methods cannot be declared abstract → why???
- Constructors are not inherited → we should call `super` for them to get invoked
- Static methods are for the whole class → cannot be overridden and their implementations have to be provided when defined

Abstact Method

- Can an abstract class have a constructor ???

Abstact Method

- Can an abstract class have a constructor ???
→ yes, but if you try to call it you get a compilation error

Abstact Class/Method Summary

- Abstract class provides basic implementation that has to be extended to make it complete
- Abstract class may or may not contain and abstract method
- If there at least one abstract method → class must be abstract
- An abstract method does not have a body rather has signature only

Abstact Class/Method Summary

- An abstract class variable can be instantiated with a reference to any of its subclasses (if they are not abstract)
- For instance:

```
Vehicle v = new Truck();
Vehicle v = new Bus();
```

- This is also called upcasting

Abstact Class/Method Summary

- An abstract class variable can be instantiated with a reference to any of its subclasses (if they are not abstract)
- For instance:

```
Vehicle v = new Truck();
Vehicle v = new Bus();
```

- This is also called upcasting → why is this legal???

Abstact Class/Method Summary

- An abstract class variable can be instantiated with a reference to any of its subclasses (if they are not abstract)
- For instance:

```
Vehicle v = new Truck();
Vehicle v = new Bus();
```

- This is also called upcasting → why is this legal???
- Simply because a Truck or a Bus is also a vehicle

Abstact Class/Method Summary

- Is the other way around also legal???
- For instance: can we say something like

```
Truck t = new Vehicle();
Bus b = new Vehicle();
```

Abstact Class/Method Summary

- Is the other way around also legal???
- For instance: can we say something like

```
Truck t = new Vehicle();
Bus b = new Vehicle();
```

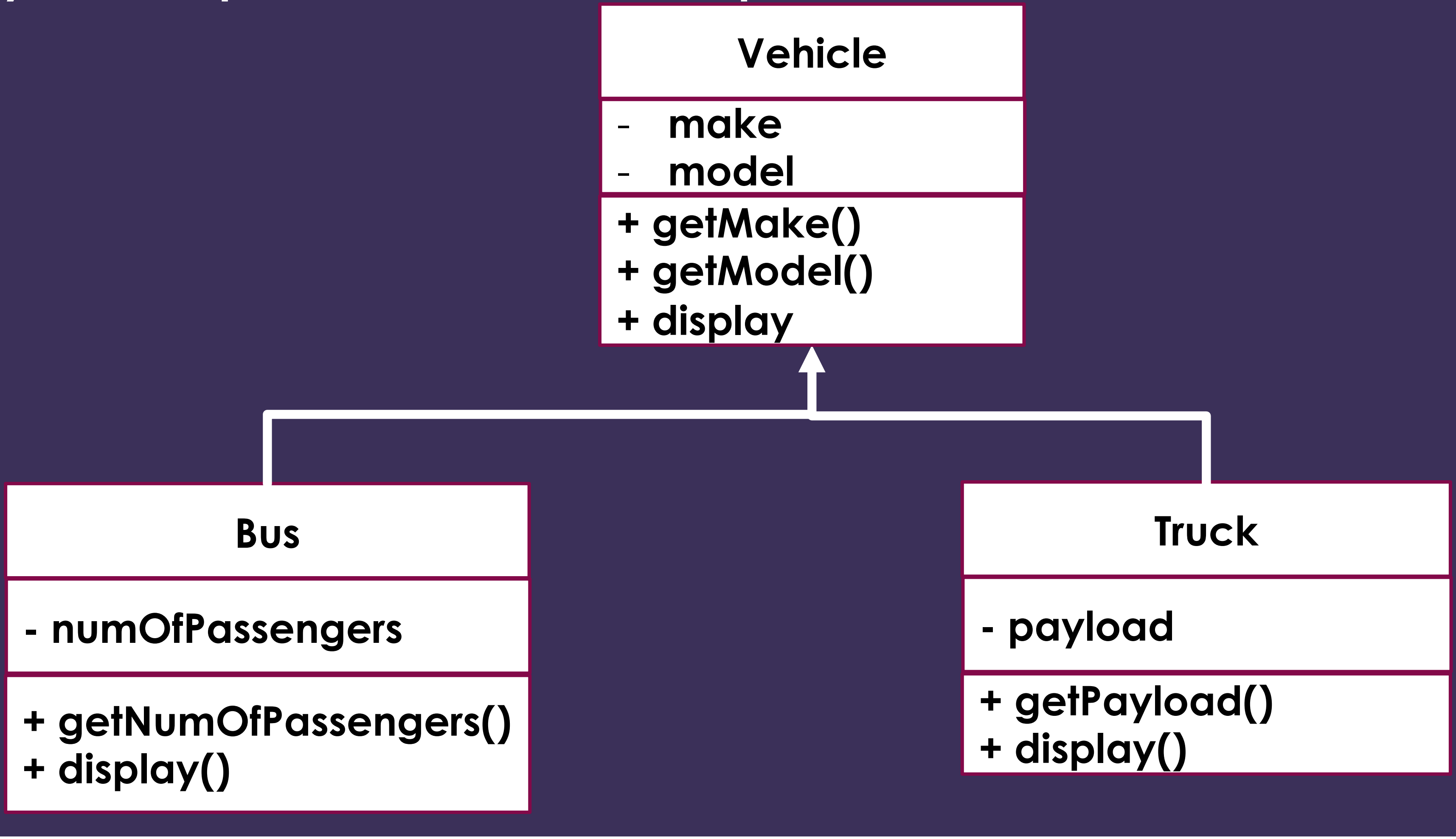
- Try it out yourself

Polymorphism

Polymorphism

- Polymorphism means many/several forms
- It has two forms; method overloading and method overriding
- Refers to the dynamic binding mechanism that determines which method definition will be called in case of overriding

Polymorphism Example



Vehicle

- make
- model

+ getMake()
+ getModel()
+ display

Bus

- numOfPassengers

+ getNumOfPassengers()
+ display()

Truck

- payload

+ getPayload()
+ display()

Dynamic Binding Example

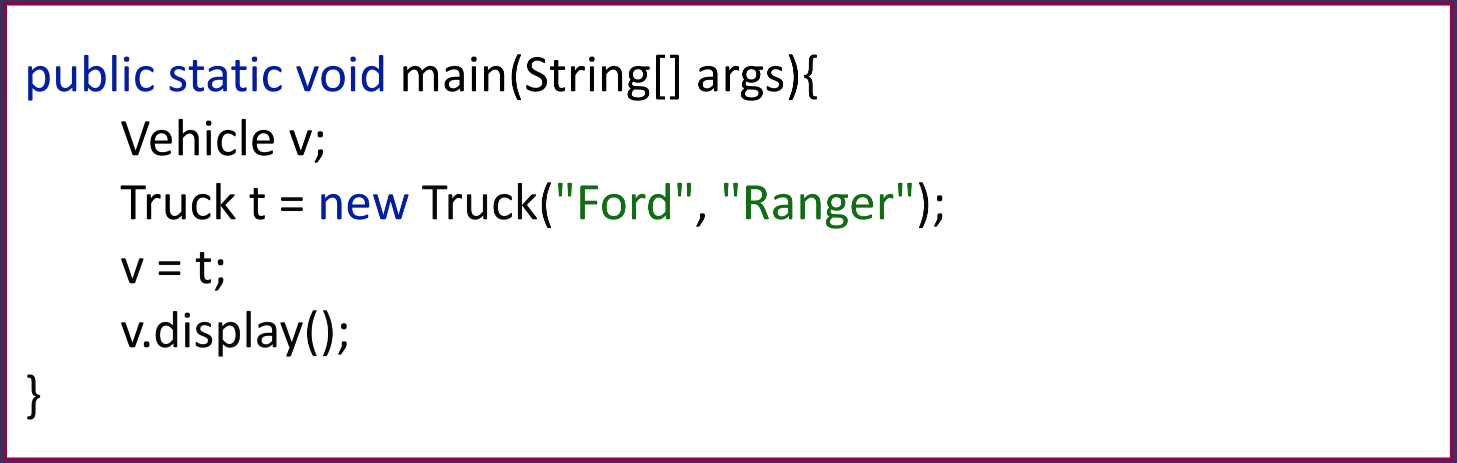
```java
public static void main(String[] args){
        Vehicle v = new Truck("Ford", "Ranger");
        v.display();
}
```

Dynamic Binding Example

```
public static void main(String[] args){
	Vehicle v = new Truck("Ford", "Ranger");
	v.display();

	v = new Bus("Toyota", "Coaster");
	v.display();
}
```

Dynamic Binding Example



```java
public static void main(String[] args){
	Vehicle v;
	Truck t = new Truck("Ford", "Ranger");
	v = t;
	v.display();
}
```

Dynamic Binding Example

```
public static void main(String[] args){
    Vehicle v;
    Truck t = new Truck("Ford", "Ranger");
    v = t;
    v.display();
    Bus b = new Bus("Toyota", "Coaster");
    v = b;
    v.display();
}
```

Why Polymorphism

- Allows you to define general methods in super classes and leave implantation details for sub classes
- Promotes software extensibility → At the time of implementation you are not aware of the new classes that will be defined but you are sure that they should implement certain method

Dynamic Binding

- When a method is overridden in a subclass and you define an object of base type → method of subclass is still called
- For instance:

```
Person p = new Employee();
```

- This means that p internally refers to an Employee, however you can only reference methods defined in Person (at compile time)

Dynamic Binding

- Via inheritance, a variable of superclass can point to an object of the class itself or any of its subclasses
- However, *YOU CANNOT DIRECTLY MAKE A VARIABLE VARIABLE OF A SUBCLASS TYPE AND POINT TO OBJECT OF ITS SUPERCLASS*.
- The actual type of the instance *AT RUNTIME* determines which method will get invoked

Upcasting and Downcasting

- Upcasting is converting an object of a subclass to it superclass → Done implicitly
- Upcasting example:

```
Person p;
Employee emp = new Employee();
p = emp;
```

Upcasting and Downcasting

- Downcasting is converting an object of a superclass to one of its subclasses → Must be done explicitly
- Downcasting example:

```
Person p = new Employee();
Employee emp;
emp = p;
```

Upcasting and Downcasting

- Downcasting is converting an object of a superclass to one of its subclasses → Must be done explicitly
- Downcasting example:

```
Person p = new Employee();
Employee emp;
emp = (Employee) p;
```

Upcasting and Downcasting

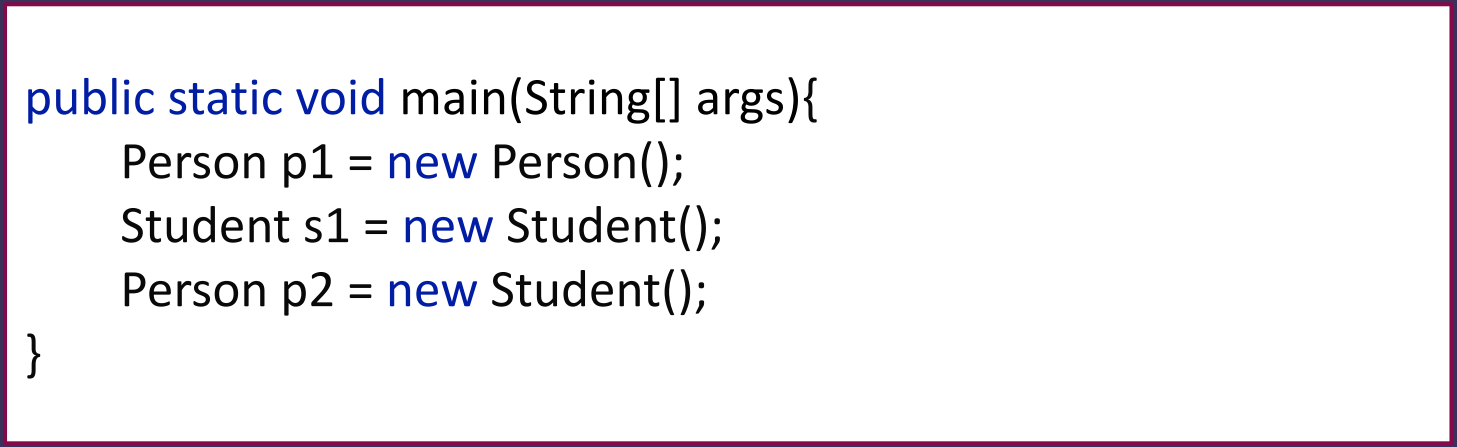
- Downcasting is converting an object of a superclass to one of its subclasses → Must be done explicitly
- Downcasting example:

```
Person p = new Employee();
Employee emp;
emp = (Employee) p;
```

- Can throw an exception if p is not actually of type Employee, but of type Student for instance

instanceof Operator

- `instanceof` operator can be used to check whether an object is a certain class type or not



```
public static void main(String[] args){
        Person p1 = new Person();
        Student s1 = new Student();
        Person p2 = new Student();
}
```

instanceof Operator Exmple

```
public static void main(String[] args){
        if(p1 instanceof Person){}

        if(p1 instanceof Student){}

        if(s1 instanceof Student){}

        if(s1 instanceof Person){}

        if(p2 instanceof Person){}

        if(p2 instanceof Student){}
}
```

instanceof Operator Exmple

```
public static void main(String[] args){
      if(p1 instanceof Person){}          ✓

      if(p1 instanceof Student){}         ✗

      if(s1 instanceof Student){}         ✓

      if(s1 instanceof Person){}          ✓

      if(p2 instanceof Person){}          ✓

      if(p2 instanceof Student){}         ✓
}
```

Upcasting and Downcasting Summary

- Assign a superclass variable to superclass object? ✓
- Assign a subclass variable to subclass object? ✓
- Assign a superclass variable to a subclass object? ✓
- Assign a subclass variable to a superclass object? → Must be done via explicit casting ✗

Exercise on Up/Down casting

- Define an abstract class called Employee with field “baseSalary” and abstract method called “calcSalary”
- Define 3 types of employees Normal, Manager, and Trainee all are subclasses of Employee
- For normal employee, net salary equals base salary * 1.2
- For manager, net salary equals base salary * 1.5
- For trainee, net salary equals base salary * 1
- Define an array of length 3 and place one of each type in that array

Thank You!