LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Polymorphism

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Polymorphism
- Collections
- Functional Programming
- Stream
- Exceptions
- Enums

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon

- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

TOPIC DESCRIPTION

Polymorphism: Imagine a general instruction like "start a vehicle." Whether the vehicle is a car, bike, or truck, they all can "start," but the mechanism differs based on the type of vehicle. Polymorphism works similarly, enabling a common action (like *start*) to be implemented for specific types.

Benefits of Composition:

Code Reusability: Same contract handles different data types or object behaviors.
Extensibility: Easier to add new functionality without altering existing interfaces.
Dynamic Behavior: Allows decisions to be made at runtime, increasing flexibility.

TOPIC DESCRIPTION

Polymorphism: one of the key principles of object-oriented programming (OOP). It allows objects to take on multiple forms depending on their use or context. Enables a single interface to represent different underlying forms (data types).

Two Types:

- Compile Time Polymorphism (Method Overloading): Achieved by having multiple methods in the same class with the same name but different parameter lists (different types, numbers, or orders of parameters).
- Run-Time Polymorphism (Method Overriding): Achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.

RUN TIME POLYMORPHISM

Consider Ver 1.0 of our code base. We have a Car and Truck classes

```
public class Car {  3 usages  new *

    public void start(){  1 usage  new *
        System.out.println("Car start");
    }
}
```

```
public class Truck {  no usages  new *
    public void start(){  no usages  new *
        System.out.println("Truck start");
    }
}
```

CLUNKY DRIVER EXAMPLE

Complex Code

This code must know what the types are and know to call a start() method.

```
public class ClunklyDriver {  new *


    public static void main(String[] args) {  new *

        ClunklyDriver driver = new ClunklyDriver();
        Car car = new Car();                // Car has a start method
        Truck truck = new Truck();          // Truck has a start method i.e. similar functionality to Car's
        // Are these not both vehicles?
        // What happens when I add a Boat?

        driver.clunkyStart(car, truck); // This interface is brittle

    }

    public void clunkyStart(Car car, Truck truck) {  1 usage  new *
        car.start();     // Call the Car's start method
        truck.start();   // Call the Truck's start method

    }
}
```

```java
public interface Startable {  1 usage  3 impleme

    void start();  3 implementations  new *
}
```

```java
public class Vehicle implements Startable{  7 usages

    @Override  2 overrides  new *
    public void start(){
        System.out.println("Vehicle started");
    }
}
```

```java
public class PolyCar extends Vehicle{  1 usage

    @Override  new *
    public void start(){
        System.out.println("Car start");
    }
}
```

```java
public class PolyTruck extends Vehicle {  2 usages

    @Override  new *
    public void start(){
        System.out.println("Truck start");
    }
}
```

BETTER EXAMPLE

We have a `Startable` interface that a new parent class implements. When we inherit from `Vehicle` the class also get the relationship to the `Startable` interface.

Note the `@Override` annotation

UPDATED DRIVER MAIN

```
    Vehicle car = new PolyCar();              // Car has a start method from the Startable interface
    Vehicle truck = new PolyTruck();          // Truck has a start method also from same interface
    Vehicle[] vehicles = {car, truck};
    driver.startVehicles(vehicles); // This signature and interface is flexible
}

public void startVehicles(Vehicle[] vehicles) {  1 usage  new *

    for (Vehicle vehicle : vehicles) {
        vehicle.start(); // LOOK Code does not know if this is a PolyCar or PolyTruck!
```

The new method `startVehicle` for version 2.0 method take an array of `Vehicles[]`. When we inherit from `Vehicle` the child class also gets the relationship to the `Startable` interface, and the method does not know which type it is at compile time.

What happens when we add a `Boat`?

ADDING A NEW TYPE

```
public class PolyBoat extends Vehicle {  no usages  new *
}
```

```
    Vehicle car = new PolyCar();                // Car has a start method from the Startable interface
    Vehicle truck = new PolyTruck();           // Truck has a start method also from same interface
    Vehicle boat = new PolyBoat();
    Vehicle[] vehicles = {car, truck, boat};
    driver.startVehicles(vehicles); // This signature and interface is flexible
}
```

```
/**
 * Code does NOT Change!
 * @param vehicles Array of Vehicles
 */
public void startVehicles(Vehicle[] vehicles) {  1 usage  new *
    for (Vehicle vehicle : vehicles) {
        vehicle.start(); // LOOK Code does not know if this is a PolyCar,PolyTruck or PolyBoat!
```

INHERITANCE FEATURES

Limitations:

- Java does not support multiple inheritance with classes to avoid ambiguity (the "diamond problem").
- A class can only extend one superclass but can implement multiple interfaces.

Inheritance is a powerful feature in Java that enhances code organization, promotes reuse, and helps in building extensible systems.

INHERITANCE INITIALIZATION

- The intialization happens from the top to the bottom
- The base class[es] each initialize themselves
- The child class can pass it constructed values "up" the chain with a special keyword called `super`.
- The keyword `super` must be the first line in the constructor
  - not counting comments

INHERITANCE INITIALIZATION EXAMPLE

```
public class Vehicle implements Startable{  14 usages  4 inheritors  new *

    protected int numWheels = 4;  4 usages

    private final Engine engine = new Engine( name: "Gas");  1 usage

    public Vehicle(){  3 usages  new *

        System.out.println("Vehicle Constructor");
        System.out.println("I Have " + numWheels + " Wheels");
    }

    public PolyTruck() {  3 usages  new *
        super();
        System.out.println("PolyTruck Constructor");
    }

    public class BigTruck extends PolyTruck {  no usages
        public BigTruck() {  no usages  new *
            super();
            System.out.println("I am Big Truck");
        }
```

- When creating a `BigTruck` instance the compiler will start with the highest base class (`Engine` in the `Vehicle` class and initialize the values from top to the bottom
  - numWheels is set to 4
  - The Engine is created

INHERITANCE INITIALIZATION EXAMPLE

```java
public class Vehicle implements Startable{  14 usages  4 inheritors  new *

    int numWheels = 4;  1 usage

    private final Engine engine = new Engine( name: "Gas");  1 usage

    public Vehicle(){  3 usages  new *
        System.out.println("Vehicle Constructor");
        System.out.println("I Have " + numWheels + " Wheels");
    }
public class PolyTruck extends Vehicle {  5 usages  1 inheritor

    public PolyTruck() {  3 usages  new *
        super();
        System.out.println("PolyTruck Constructor");
    }
public class BigTruck extends PolyTruck {  no usages
    public BigTruck() {  no usages  new *
        super();
        System.out.println("I am Big Truck");
    }
```

```java
public static void main(String[] args) {
    BigTruck bt = new BigTruck();
```

1. Engine Name: Gas
2. Vehicle Constructor
3. I Have 4 Wheels
4. PolyTruck Constructor
5. I am Big Truck Constructor
6. Start Truck
7. Start a Big Truck

BEST PRACTICE

```
public class Vehicle implements Startable{  14 usages  4 inheritors  new *

    int numWheels = 4;  1 usage

    private final Engine engine = new Engine( name: "Gas");  1 usage

    public Vehicle(){  3 usages  new *
        System.out.println("Vehicle Constructor");
        System.out.println("I Have " + numWheels + " Wheels");
    }
public class PolyTruck extends Vehicle {  5 usages  1 inheritor

    public PolyTruck() {  3 usages  new *
        super();
        System.out.println("PolyTruck Constructor");
    }
public class BigTruck extends PolyTruck {  no usages
    public BigTruck() {  no usages  new *
        super();
        System.out.println("I am Big Truck");
    }
```

Minimize the steps to set the object into a valid state

Don't call any other methods in this class from the constructor

The only safe methods to call inside a constructor are those that are final in the base class

DOWNCASTING

Downcasting is used when working with polymorphism, where methods operate on superclass references, but you need to access subclass-specific functionality.

Sometimes you need to know the child's type and not just the base type or interface it implements.

Its better to design your code to minimize the need for downcasting by using interfaces, abstract classes, or method overriding.

KEY POINTS

```
public void startVehicles(Vehicle[] vehicles) {  1 usage   new *
    for (Vehicle vehicle : vehicles) {
        vehicle.start(); // LOOK Code does not know if this is a
        // PolyCar, PolyTruck or PolyBoat!

        // If you must know the type you interrogate the type
        // Be careful this is generally a code smell!
        if (vehicle instanceof PolyTruck truck){
            truck.dropCargo();
        }
    }
```

Upcasting is automatic:
Assigning a subclass object to a superclass reference happens implicitly

```
Vehicle v = new PolyTruck();

v.start()
```

```java
/**
 * Truck Specific method
 */
public void dropCargo() { System.out.println("Drop cargo"); }
```

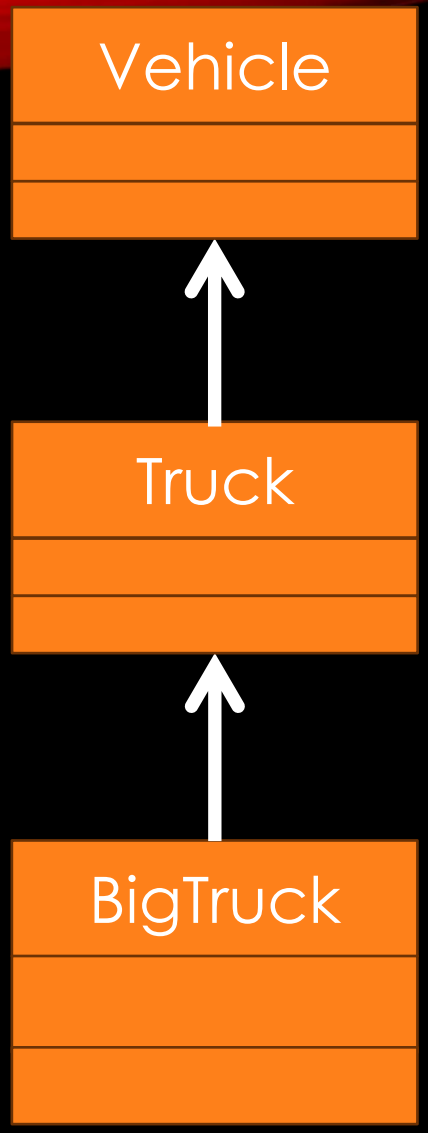
```java
public void startVehicles(Vehicle[] vehicles) {  1 usage  new *

    for (Vehicle vehicle : vehicles) {
        vehicle.start(); // LOOK Code does not know if this is a
        // PolyCar, PolyTruck or PolyBoat!

        // If you must know the type you interrogate the type
        // Be careful this is generally a code smell!
        if (vehicle instanceof PolyTruck truck){
            truck.dropCargo();
        }
    }
```

KEY POINTS

Converting a superclass reference back to a subclass reference needs explicit casting

```
PolyTruck p = (PolyTruck)vehicle
```

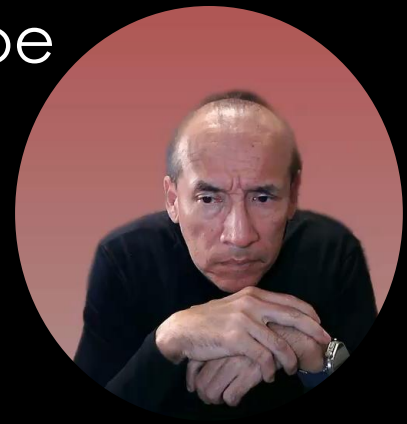
Runtime Checks: Subject to runtime type checks. If the object being cast is not an instance of the subclass, JVM throws `ClassCastException`.



TOPIC SUMMARY - INHERITANCE

Inheritance: a fundamental object-oriented programming (OOP) principle allowing a new class (a subclass) to inherit properties and behaviors (methods) from an existing class (called a superclass or base class).

Referred to as an "is-a" relationship, where the subclass is a more specific version of the superclass. E.g. A square “is-a” rectangle, which in turn “is-a” Shape



TOPIC SUMMARY

Polymorphism means many bodies in Latin

The concept only works in the context of Inheritance and Interfaces

The start method only needed what's in the Vehicle interface

Perh
con
an
e