CS5004, Spring 2024

Lab 5:ADTs. Introduction to Polymorphism

Therapon Skoteiniotis, Tamara Bonaci and Abi Evans

skotthe@ccs.neu.edu, tbonacin@northeastern.edu, ab.evans@northeastern.edu

1. Summary

In today's lab, we will explore object-oriented concept of polymorphism. We will focus on:

- Subtyping and subtype polymorphism
- Static and dynamic data types
- Casting
- Ad-hoc polymorphism
- Overriding and overloading of methods

Note 1: Labs are intended to help you get started, and give you some practice while the course staff is present and able to provide assistance. You are not required to finish all the questions during the lab, but you are expected to push your lab work to a designated repo on the Khoury GitHub.

2. ADTs and Data Collections

Problem 1

You are working with a group of marine scientists, going on an expedition around Juan de Fuca Strait. Their mission is to collect information about marine mammals living in the Strait. The scientists have already developed some code, to encode relevant information about a single marine mammal. The code is provided in the lecture-code repo.

The scientists have also designed an ADT for the `MarineMammalDirectory`, a linked data collection of all of the encountered marine mammals. The ADT for `MarineMammalDirectory` is given as follows:

- **createEmpty()**: `MarineMammalDirectory` - creates an empty `MarineMammalDirectory`

- **add(MarineMammal mammal)**: `MarineMammalDirectory` - adds `MarineMammal` in the directory. If `MarineMammal` is already in the directory, the directory remains unchanged.
- **isEmpty():** `Boolean` - returns true is `MarineMammalDirectory` is empty, and false otherwise
- **size()**: `Integer` - returns the number of `MarineMammals` in an non-empty `MarineMammalDirectory`, and 0 if the directory is empty.
- **contains(MarineMammal mammal)**: `Boolean` - returns true if `MarineMammal` exists in the directory, and false otherwise.
- **remove(MarineMammal mammal)**: `MarineMammalDirectory` -returns `MarineMammalDirectory` with `MarineMammal` removed. If `MarineMammal` was not a part of the directory, returns the unchanged directory.
- **getMammal()**: `MarineMammal` - returns a random `MarineMammal`.

Please help the marine scientists, and provide the UML class diagram for the MarineMammalDirectory ADT.

3. Subtype Polymorphism

In the programming language theory, **subtype polymorphism** is a form of polymorphism in which a subtype (a subclass) is related to another data type (supertype, or superclass) in such a way that a subtype inherits the states and the behavior of the supertype, and therefore a **subtype can be viewed as its supertype**.

2.1. Static and Dynamic Data Types

Binding is what happens when a method invocation is bound to an implementation. It involves method lookup in the class, or one or its parents, and during the lookup, both method names and input arguments are being checked.

Binding can happen at two different times:

- **Static binding** happens during the compile time (method calls bound to their implementation immediately)
- **Dynamic binding** happens during run time

2.2. Dynamic Binding and Subtype Polymorphism

Subtype polymorphism allows us to treat instances of a subclass as if they are an instance of a parent class. That, in turns, means that a compiler will not know what the type of the instance is, only its base type.

Dynamic binding allows us to deal with this problem – it is the JVM, and not a compiler that binds a method call to its implementation.

Practice Exercise

For practice, let's take a look at some examples, and determine the compile-time types, and the run-time types, as well as the methods that actually get called.

- **Example 1:**

```
//class definition
public class MotorBike {
        public void revEngine() {…}
}

//usage
MotorBike bike = new MotorBike();
motorbike.revEngine();
```

Static time data type of variable bike is: _____
Dynamic time data type of variable bike is: _____

- **Example 2:**

```
public class MotorVehicle {
  public void start() {…}
  public void stop() {…}
}

public class MotorBike extends MotorVehicle()
{
  //overridden
  public void start() {…}
  public void revEngine() {…}
}

//usage
MotorBike bike = new MotorBike();
motorbike.start();
```

Static time data type of variable bike is: _____

Dynamic time data type of variable bike is: _____

- **Example 3:**

```
//reference is to base class
MotorVehicle vehicle = new MotorBike();
vehicle.start();

Object object = new MotorBike();
object.toString();
```

Static time data type of variable vehicle is: _____
Dynamic time data type of variable vehicle is: _____

Static time data type of variable object is: _____
Dynamic time data type of variable object is: _____

- **Example 4:**

```
public interface ElectricalAppliance {

 public void turnOn();
 public void turnOff();
}

public class RemoteControl() {

  public static void turnApplianceOn(ElectricalAppliance appliance)
  {
    appliance.turnOn();
  }
}
public class HairDryer implements ElectricalAppliance {
}
public class Light implements ElectricalAppliance {
}

ElectricalAppliance appliance = new HairDryer();
RemoteControl.turnApplianceOn(appliance);
```

Static time data type of variable appliance is: _____
Dynamic time data type of variable appliance is: _____

```
appliance = new Light();
RemoteControl.turnApplianceOn(appliance);
```

Static time data type of variable appliance is: _____
Dynamic time data type of variable appliance is: _____

- **Example 5:**

```
public class HairDryer implements ElectricalAppliance {
 //other methods
 public void adjustTemperature();
}

ElectricalAppliance appliance = new HairDryer();
//following won't compile
appliance.adjustTemperature();
```

What is the reason the commented line of code does not compile?

Problem 2 (Transit Vehicle Management System)

Please implement a basic **transit vehicle management system**. The system needs to track three types of Vehicle:

- Bus
- Train
- Boat

Each vehicle has the following fields:

- ID, represented as a String
- Average speed, represented as a Float
- Max speed, represented as a Float

The system also tracks another data type, a TripReport, which is used to store information about a trip made by a particular vehicle. TripReport tracks the following:

- The Vehicle that took the trip
- Speed in miles per minute, a Float
- Distance traveled in miles, a Float
- Trip duration in minutes, an Integer

Implement class FleetManager that contains a single method with the following signature:

```
TripReport drive(float distance, Vehicle vehicle);
```

The drive method uses **subtype polymorphism** to create a TripReport for any type of Vehicle. To populate the fields of a TripReport, use the vehicle and distance passed as method parameters, the vehicle's average speed, and calculate the trip duration as distance divided by speed.

4. Ad Hoc Polymorphism and Method Overloading

Overloading (not be confused with overriding) allows us to create methods that share the **same method name** but **differ in their signature**. Overloading can be used for any method (constructor, static or non-static methods).

3.1. Calling an Overloaded Method

Calling an overloaded method is the same as calling any method. The Java compiler and runtime will call the appropriate method based on the **signature** that your method call matches.

3.2. Overloading and Subtypes

The Java compiler is responsible for figuring out which overloaded method will be called at runtime. Therefore, when calling an overloaded method whose arguments are reference type, the compiler will resolve (figure out which method to run) by using the **compile-time** type of the arguments!

Problem 3 (Transit Vehicle Management System and Overloaded Methods)

This problem builds on Problem 1, where you implemented a basic **transit vehicle management system**. As a reminder, the system needed to track three types of Vehicle:

- Bus
- Train
- Boat

The system also tracked another data type, a TripReport, which was used to store information about a trip made by a particular vehicle.

You implemented class FleetManager that contains a single method with the following signature:

TripReport drive(float distance, Vehicle vehicle);

The drive method used **subtype polymorphism** to create a TripReport for any type of Vehicle. To populate the fields of a TripReport, you used the vehicle and distance passed as method parameters, the vehicle's average speed, and you were asked to calculate the trip duration as distance divided by speed.

In this problem, we will use **method overloading**. Your task is to create three additional versions of the **drive method in the FleetManager** class, to allow client code to choose how to use the implementation.

Remember that with overloading, no two method signatures can share the same number of parameters AND the same types for those parameters.

If a drive method that takes speed as a parameter is passed a speed that exceeds the vehicle's max speed, use the vehicle's max speed to calculate the TripReport information.

Write tests to confirm that your overloaded methods work as expected.

Optional Hard(er) Practice Exercise

Consider the following example code:

```
public class Espresso extends Coffee {

    public void method1() {
        System.out.println("E 1");
        super.method2();
    }

    public void method2() {
        System.out.println("E 2");
    }

    @Override
    public String toString() {
        return "Espresso:";
    }
}
```

```
public class Beverage {

    public void method1() {
        System.out.println("B 1");
    }

    public void method2() {
        System.out.println("B 2");
    }

    public void method3() {
        System.out.println("B 3");
    }

    public void tastesAs(Beverage drink) {
        System.out.println("Tastes as Beverage");
    }

    @Override
    public String toString() {
        return "Beverage:";
    }
}
```

```
public class DoubleEspresso extends Espresso {

    public void method2() {
        System.out.println("D 2");
    }

    @Override
    public String toString() {
        return "DoubleEspresso:";
    }
}
```

```
public class Coffee extends Beverage {

    public void method3() {
        System.out.println("C 3");
        method2();
    }

    public void tastesAs(Coffee drink) {
        System.out.println("Tastes as Coffee");
    }

    @Override
    public String toString() {
        return "Coffee:";
    }
}
```

Write down the output you expect to see in the Console when `testDrinks1.java` (below) is executed.

```
public class testDrinks1{
    public static void main (String [] arg){
        Beverage[] drinks = {new DoubleEspresso(), new Beverage(), new Coffee(), new Espresso()};

        for (int i=0; i<drinks.length; i++){
            System.out.println(drinks[i]);
            drinks[i].method1();
            System.out.println("*****");
            drinks[i].method2();
            System.out.println("*****");
            drinks[i].method3();
            System.out.println("*****");
        }

    }

}
```

Solutions, with object names omitted:

```
E 1
B 2
*****
D 2
*****
C 3
D 2
*****
B 1
*****
B 2
*****
B 3
*****
B 1
*****
B 2
*****
C 3
B 2
*****
E 1
B 2
*****
E 2
*****
C 3
E 2
*****
```

Write down the output you expect to see in the Console for lines 1 - 5 from `testDrinks2.java` (below).

If you think a certain line will not compile, write down **compile-time error**.
If you think a certain line will cause Runtime Exception, write down **ClassCastException.**

```
public class testDrinks2 {
    public static void main (String [] arg){

        Beverage cup1 = new Espresso();
        Object cup2   = new DoubleEspresso();

        ((Coffee)cup1).method1(); // line 1
        ((DoubleEspresso)cup1).method1();  // line 2
        ((Beverage)cup2).method2(); // line 3
        ((Coffee)cup2).method2(); // line 4
    //    cup2.method3();  //line 5
    }
}
```

Solution: Key point to remember: **Casting does NOT change the runtime (dynamic) type of an object .**

- line 1 - E 1 B 2
- line 2 - `ClassCastException`
- line 3 - D 2
- line 4 - D 2
- line 5 - `compile-time error`

A diligent student recalled that Java resolves overloading at compile time using the static types, while dynamic dispatch selects methods to execute at run time using the dynamic types.

However, this student was puzzled how overloading works with dynamic dispatch.
Help this student to figure out which line or lines from testDrinks3.java will output "Tastes as Coffee". Circle **each** line from the below code that will print out "Tastes as Coffee" (that may be one line or several lines).

```
public class testDrinks3 {
    public static void main (String [] args) {

        Beverage bb = new Beverage();
        Beverage bc = new Coffee();
        Coffee cc = new Coffee();

        bb.tastesAs(bb); // line 1
        bb.tastesAs(bc); // line 2
        bc.tastesAs(bb); // line 3
        bc.tastesAs(bc); // line 4
        cc.tastesAs(bb); // line 5
        cc.tastesAs(bc); // line 6
        bb.tastesAs(cc); // line 7
        bc.tastesAs(cc); // line 8
        cc.tastesAs(cc); // line 9
    }
}
```

Solution:

- Tastes as Beverage
- Tastes as Beverage

- Tastes as Beverage
- Tastes as Beverage
- Tastes as Beverage
- Tastes as Beverage
- Tastes as Beverage
- Tastes as Beverage
- Tastes as Coffee