

A collage of vintage audio formats. In the background is a large, dark grey vinyl record. Overlaid on the record are several other items: a colorful CD with a rainbow-like reflection, a black cassette tape, and three clear plastic cassette tape cases. The cases are arranged in a cluster, with one case in the foreground showing its internal reels and another one partially open. The overall image has a soft, slightly faded appearance.

Module 1 Day 12

Polymorphism: Inheritance & Interfaces

Module 1 Day 12

Can you ... ?

1. Explain what polymorphism is and how it is used with inheritance and interfaces
2. Explain where & how inheritance can help us write polymorphic code
3. State the purpose of interfaces and how they are used
4. Implement polymorphism through inheritance (also see Day 11 Lecture Final)
5. Implement polymorphism through interfaces

Three Main Inheritance Scenarios

Recall from the previous discussion that there are three main ways inheritance can be implemented.

- A **concrete class** (all the classes we have seen so far) inheriting from another concrete class. (Day 11)
- A concrete class inheriting from an **Interface**. (Today!)
- A concrete class inheriting from an **abstract class**. (Tomorrow)

Today we will be working with Java interfaces.

Java Interfaces: A Contract of Behavior

An interface defines a contract of behavior that will be honored by any class that implements the interface.

Interface in the real world:

- **A fast food restaurant chain requires that each franchisee mount a giant corporate logo in the front of the building.**
- *While a Logo is required to exist, the franchisee is free to choose the contractors & workers it needs to actually place it on their building.*

Here, the existence of the Corporate Logo is the contract of behavior (interface); it is expected when management (application) calls to inspect. The contractors and their workers are the implementation (method) at the location (class).

Java Interfaces: The Obligations

- A class that implements an interface must define its own specific implementation of the interface abstract methods.
- The methods that the class needs to implement are defined in the **Interface** using **abstract methods**.
- An interface cannot be instantiated; they can only be “implemented” by other classes.
- An interface itself is an example of code that is “abstract in nature”, it *is not* an *Abstract Class* (more on that tomorrow)

Java Interfaces: Declaration & Implementation

- The declaration for an Interface is as follows:

```
public interface <<Name of the Interface>> {...}
```

- A class implementing an Interface must have the following convention:

```
public class <<Name of Class>> implements <<Name of Interface>> {...}
```

- The class implementing an interface is also called a **concrete class**.
- You cannot instantiate Interfaces, you can only instantiate the classes that implement an interface.

Java Interfaces: Abstract Methods

An abstract method is one that ***doesn't have an implementation***; the method has no body. Here is an example from a Vehicle Interface:

```
package te.mobility;  
  
public interface Vehicle {  
  
    public void honkHorn();  
    public void checkFuel();  
}
```

- The Interface Vehicle has two abstract methods:
honkHorn()
and
checkFuel()
- These abstract methods do not have method bodies. There are no {*//code blocks.*}, and they each **end with a semicolon.**

Java Interfaces: Abstract Methods

A class implementing the Vehicle interface ***must*** provide a concrete implementation of the two abstract methods.

```
package te.mobility;  
  
public interface Vehicle {  
  
    public void honkHorn();  
    public double checkFuel();  
  
}
```

honkHorn has
been
implemented

checkFuel has
been
implemented

```
package te.mobility;  
  
public class Car implements Vehicle {  
  
    private double fuelLeft;  
    private double tankCapacity;  
  
    @Override  
    public void honkHorn() {  
        System.out.println("beeeep?");  
    }  
  
    @Override  
    public double checkFuel() {  
        return (fuelLeft / tankCapacity) * 100;  
    }  
  
}
```


Java Interfaces: Abstract Method Rules

When implementing the interface abstract methods in a concrete class, the following rules are in effect:

- To fulfill the Interface's contract, the concrete class must implement the method with the exact return type, name, and arguments. In other words, the signatures ***must*** match.
- The access modifier on the implemented method cannot be more restrictive than that of that Interface.
 - For example - the concrete class cannot implement the method as private if the abstract class has marked it as public.
- All abstract methods in the interfaces definition are assumed to be public.

Java Interfaces: Polymorphism

Polymorphic objects are those that can assume many forms. In other words, they can pass more than one “Is-A” test.

- A child object from a parent class that implements an interface:
 - is a member of the child class
 - and a member of the parent class
 - And a member of interface “class”.
- Yesterday we instantiated a ReserveAuction with
`ReserveAuction classicCar = new ReserveAuction ()`
classicCar Is-A ReserveAuction **and** Auction.

Polymorphism is the ability to leverage these relationships in order to write more compact and reusable code.

Java Interfaces: Polymorphism References

Interfaces allow us to create references based on the interface, but instantiate an instance of the concrete class instead.

```
Vehicle fastCar = new Car();
```

The diagram illustrates the components of the Java code snippet `Vehicle fastCar = new Car();`. A central grey box contains the full line of code. Two arrows point from this box to two separate light blue boxes below it. The left arrow points to the text 'To the left of the equal sign is the reference, note it is of the interface type.', which explains the `Vehicle` part of the code. The right arrow points to the text 'To the right of the equal sign is the instantiation of the object, note is of the concrete class.', which explains the `new Car()` part of the code.

To the left of the equal sign is the reference, note it is of the interface type.

To the right of the equal sign is the instantiation of the object, note is of the concrete class.

Java Interfaces: Polymorphism in use

Assuming that Car and Truck implements vehicle, consider a new class called RepairShop. Just like the real world, this RepairShop class is able to repair more than one type of vehicle.

```
package te.main;


import te.mobility.Car;

public class RepairShop {

    public void repairVehicle(Car damagedCar) {
        System.out.println("repairing");
    }

}
```

This is an issue, the RepairShop only accepts objects of class Car! Hmm...



Java Interfaces: Polymorphism in use

We can rely on interfaces to make the `repairVehicle` method much more flexible by defining it so that it accepts any `Vehicle` (interface) type.

```
package te.main;

import te.mobility.Car;

public class RepairShop {

    public void repairVehicle(Vehicle damagedVehicle) {
        System.out.println("repairing");
    }

}
```

The method will now accept objects of class `Car`, and objects of class `Truck`.

Java Interfaces: Polymorphism Example

In a Garage class, we can use interfaces to create **vehicles** of any type and supply the **RepairShop** repairVehicle method the **Vehicle**(s) it is expecting.

Both of these Declarations and Assignments are ok to make because Cars and Trucks are concrete classes implementing the Vehicle interface.

```
package te.main;
import te.mobility.Car;
import te.mobility.Truck;
import te.mobility.Vehicle;

public class Garage {

    public static void main(String[] args) {

        Vehicle fastCar = new Car();
        Vehicle bigTruck = new Truck();
        RepairShop repairShop = new RepairShop();

        repairShop.repairVehicle(fastCar);
        repairShop.repairVehicle(bigTruck);

    }
}
```



Java Interfaces Beyond the Basics: Default Methods

Looking at our Vehicle interface, we could define a default method as follows:

```
package te.mobility;  
  
public interface Vehicle {  
  
    public double checkFuel(String units);  
  
    default void honkHorn() {  
        System.out.println("beep");  
    }  
}
```

An instance of a concrete class that implements Vehicle can just call honkHorn now through the instantiated object, i.e. myCar.honkHorn();

Java Interfaces Beyond the Basics: Default Methods

A concrete class can override the default method by implementing its own version of the method:

```
package te.mobility;

public interface Vehicle {

    public double checkFuel(String units);

    default void honkHorn() {
        System.out.println("interface");
    }
}
```

For an instance of car, if honkHorn is invoked, this concrete method takes priority. The output from Car.honkHorn() will be “concrete.”

```
package te.mobility;

public class Car implements Vehicle {

    private double fuelLeft;
    private double tankCapacity;

    public void honkHorn() {
        System.out.println("concrete");
    }

    @Override
    public double checkFuel(String units) {
        return (fuelLeft / tankCapacity) * 100;
    }
}
```


Java Interfaces Beyond the Basics: Data Members

It is possible for interfaces to have data members, if they do, they are assumed to be public, static, and final... Constants.