# **Interface**

OO Programming involves encapsulation/data hiding, inheritance, and polymorphism

*- inheritance*

- classes are created from existing classes (**extends** keyword)

- implicit **extends Object** in the absence of an explicit extends

- inherit attributes and behaviors of an existing class (superclass)

- superclass should provide functionality useful in many subclasses

- embellish these with new/different capabilities in new classes (subclass)

- *inheritance of interface* (**implements**) / *inheritance of implementation* (**extends**)

- *is-a* relationship

Java does not support multiple inheritance of implementation (only single inheritance of implementation)

indirect superclass class Indirect\_Superclass

single inheritance

superclass class Superclass extends Indirect\_Superclass

single inheritance

subclass class Subclass extends Superclass

- Java does support multiple inheritance of *interfaces*

- achieves many of the advantages of multiple inheritance without the associated problems

- pure *inheritance of interface*

superclass interface1 interface2 ...

subclass class Subclass extends Superclass

implements Interface1, Interface2

Every object of a subclass type *is* also *an* object of that subclass’s superclass type (**interface** types also)

- “subclass-object-*is*-*a*-superclass-object” relationship

- contrast with *composition* — *has*-*a* relationship

- converse is not true — superclass objects are not objects of that superclass’s subclasses

- superclass members become members of the subclass

**Interface**

Java allows a variable of an interface type to refer to several different types of objects that implement the interface (one form of *polymorphism*; *inheritance of interface*)

Java interface provides a description of the public methods that objects of that type need to provide

- describes the type's public interface (what)

- specifies a contract that any class that implements the interface must satisfy (how)

Interfaces contain no method definitions, no constructors, and no instance variables (no state) [pre-Java SE 8]

- may contain public static final constants

- these constants become part of any class that implements the interface

Classes may implement multiple interfaces

- multiple *inheritance of interface*

public class Circle implements Resizable, Movable, Colorable

A polymorphic method can take arguments of different types through the use of interfaces

public Resizable getInShape( Resizable shape ) { ... }

Extending Interfaces

public interface SubInterface extends SuperInterface1, SuperInterface2

Java SE 8 Interface Enhancements

default interface methods - default implementation if the implementing class does not provide an

overriding implementation

- keyword default instead of abstract

public **default** void superSizeMe( double size ) { /\* default code \*/ }

- allows you to evolve existing interfaces by adding new methods to old interfaces without breaking

code that uses them

- arguably more flexible than abstract classes (more on this later)

static interface methods - static helper methods for working with objects that implement the interface

- static methods belong to the interface type, not the implementing type

- different than public static final constants defined in an interface

Beware: Some (many/most?) OOP purists think static interface methods (and to a lesser extent default interface methods) are an abomination

Program to an Interface, Not an Implementation

- Implementation inheritance (extends) is best for small numbers of tightly coupled classes

- Interface inheritance (implements) is best for flexibility

public **interface** OperateCar {

  // constant declarations, if any

public static final int numWheels = 4;

// method signatures. **Can omit public abstract**

  public abstract moveForward(double speed);

  int turn(Direction direction, double radius, double startSpeed, double endSpeed);

  int changeLanes(Direction direction, double startSpeed, double endSpeed);

  int signalTurn(Direction direction, boolean signalOn);

  int getRadarFront(double distanceToCar, double speedOfCar);

//default method.

  default int getRadarRear(double distanceToCar, double speedOfCar){

return distanceToCar / speedOfCar;

}

         ......

   // more method signatures or default methods

}

By default, all interface method headers are public and abstract

An interface provides only the method headers (not the bodies) – name, parameters, return type – followed by ;

- classes that implement an interface are required to provide the method bodies for all these headers

Associate an interface with a class with the implements clause in the class header.

public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --

    // for example:

    int signalTurn(Direction direction, boolean signalOn) {

       // code to turn BMW's LEFT turn indicator lights on

       // code to turn BMW's LEFT turn indicator lights off

       // code to turn BMW's RIGHT turn indicator lights on

       // code to turn BMW's RIGHT turn indicator lights off

    }

    // other members, as needed -- for example, helper classes not

    // visible to clients of the interface

}

What if we did not provide method bodies (implementations) for every method header in the interface?

In some other class we can say:

*OperateCar*

OperateBMW760i

OperateToyota

Type Hierarchy

private OperateCar driver;

private OperateToyota ref1 = new OperateToyota( ... );

private OperateBMW760i ref2 = new OperateBMW760i( ... );

driver = ref1; // or driver = new OperateToyota( ... );

shape.signalTurn(Direction.LEFT, true );

driver = ref2; // or driver = new OperateBMW760i( ... );

shape.signalTurn(Direction.LEFT, true );

A variable or parameter or return value whose type is an interface may reference objects from any class that implements that interface

Static Compile Time Method Invocation Check

- the only thing the compiler knows when looking at your program is the type of the reference

- compiler emits code to call the method with the signature it finds in that type at compile time

- possible argument coercion to match method signature if no exact match

ref.method( … );

Dynamic Run Time Method Invocation

- which method code is executed at run time is determined by which object the message is being sent to

- could be either a OperateToyota object or a OperateBMW760i object or any object that implements OperateCar

- can only send a message to an object through an interface type if the message is part of the interface

**Exercise: Design a system such that all the ATMs from these banks will work with some sort uniform functionality.**

