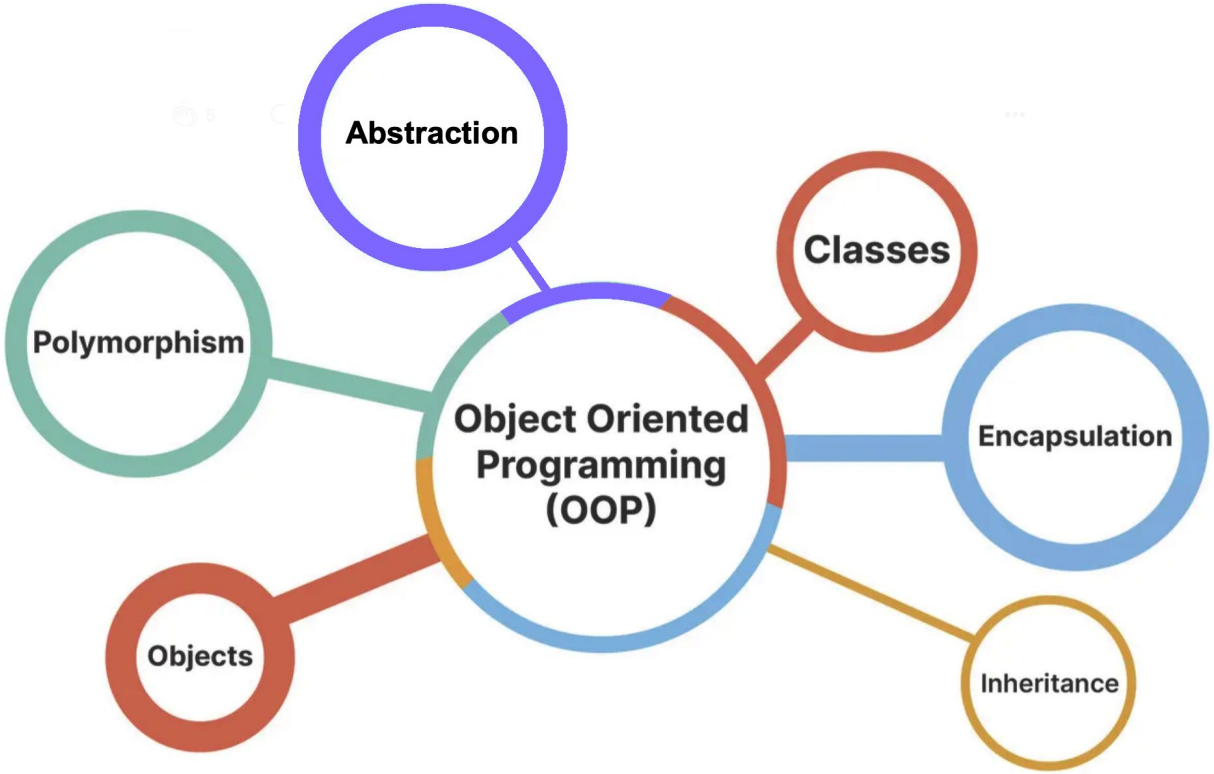
COSC 121: Computer Programming II



Today's Key Concepts

- Abstract classes are special classes that model generic concepts but never get instantiated
  - Keyword: abstract
- An interface defines the blueprint for a class, defining the set of methods that must be implemented without defining their specific implementation
  - User defined interfaces
    - Methods are all implicitly public and abstract
    - Java 8/9: Methods can be default, static, private
  - Java standard interfaces: Comparable and Cloneable
- Interfaces or abstract classes?

Abstract Class Example

- Syntax:
  abstract class className() { … }

```
public abstract class Animal
{
  protected int numLegs;
  protected int speed;
  public abstract void walks();
  public abstract void eats();
  public int runs()
  {
    // statements to define how fast it runs based on speed
  }
}
```

Abstract Class Example

```
public abstract class Animal
{
  protected int numLegs;
  protected int speed;
  public abstract void walks();
  public abstract void eats();
  public int runs()
  {
    // statements to define how fast it runs based on speed
  }
}
```

- Syntax:
  abstract void methodName();

- Abstract methods end with semicolon ;
- All subclasses ***must*** define walks() and eats() or declare them abstract too

Abstract Classes and Methods

- Abstract classes are usually used to define common features for their subclasses
  - Can***not*** instantiate objects of an abstract class
  - Abstract classes may contain abstract methods
  - Abstract methods are used to define common behavior for subclasses that have different implementations of those methods
- Classes that extend abstract classes must implement (override) abstract methods or declare them abstract as well

Can abstract classes have constructors?

Abstract Classes and Methods

- Abstract classes are usually used to define common features for their subclasses
  - Can***not*** instantiate objects of an abstract class
  - Abstract classes may contain abstract methods
  - Abstract methods are used to define common behavior for subclasses that have different implementations of those methods
- Classes that extend abstract classes must implement (override) abstract methods or declare them abstract as well

Can abstract classes have constructors? Yes!
Why?

Abstract Classes and Methods

- Abstract classes are usually used to define common features for their subclasses
  - Can***not*** instantiate objects of an abstract class
  - Abstract classes may contain abstract methods
  - Abstract methods are used to define common behavior for subclasses that have different implementations of those methods
- Classes that extend abstract classes must implement (override) abstract methods or declare them abstract as well

Can abstract classes have constructors? Yes!
Why? Initialize attributes, also called by subclasses for attribute initialization

iClicker Question

Which code snippet defines an abstract class correctly?

A.
```
class A {
   abstract void method() { … }
}
```

B.
```
public class abstract A {
   abstract void method();
}
```

C.
```
class A {
   abstract void method();
}
```

D.
```
abstract class A {
   protected void method();
}
```

E.
```
abstract class A {
   abstract void method();
}
```

Final Notes about Abstract Classes

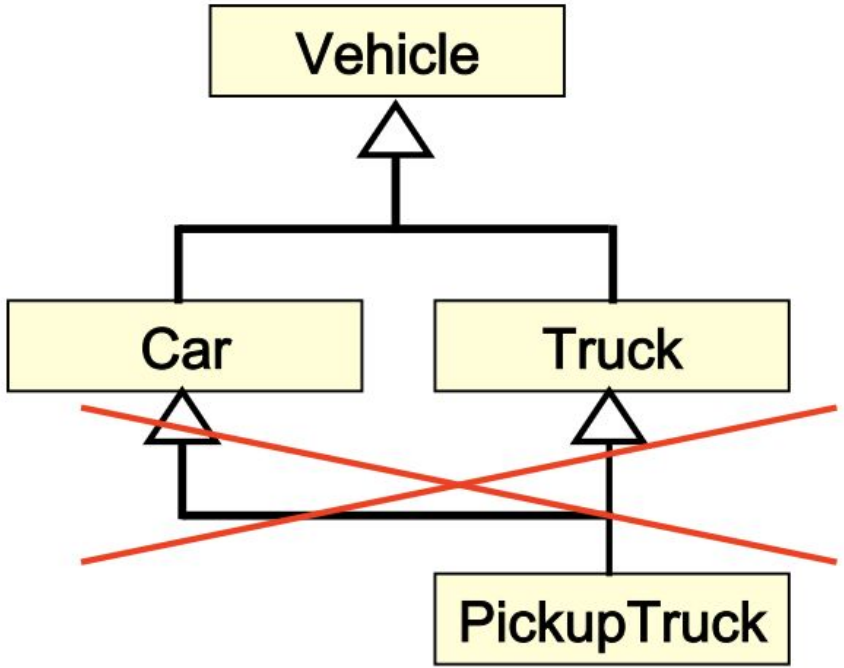
- Any class that contains abstract methods must be abstract
  - Abstract classes don't have to have abstract methods
- A subclass can be abstract even if its superclass is not abstract
  - Ex: Object class is "concrete", yet you can define abstract classes
- An abstract subclass can override a concrete method in its superclass and define it abstract instead
  - Ex: redefining default behavior for equals() in subsequent classes
- You can create **reference variables** of an abstract type!
  - Ex: Animal pet = new Dog();

Recall: Problem of Multiple Inheritance

- Java does not allow multiple inheritance
- Example:

Car inherits from Vehicle
Truck inherits from Vehicle

PickupTruck ***cannot*** inherit from more than one class



Origin of Interfaces

- Interface is an OOP technique that lets you "conform to" multiple classes
- Avoids the collision problem that arises in multiple inheritance
  - Interfaces don't have any attributes
  - A class can implement multiple interfaces
- Ex: a penguin is a bird that does not fly and has mammal behaviors

```
// Bird interface definition
public interface Bird {
    void layEggs(); // All birds lay eggs
    void move();    // Common movement method
}

// Mammal interface definition
public interface Mammal {
    void nurseYoung(); // Mammals nurse their young
    void move();       // Common movement method
}
```

No collision with move()

Penguin Example (cont.)

```
// Penguin class implements both Bird and Mammal interfaces
public class Penguin implements Bird, Mammal {

    // from Bird interface
    public void layEggs() {
        System.out.println( "lays one egg per year" );
    }

    // from Mammal interface
    public void nurseYoung() {
        System.out.println( "nurses its young with crop milk" );
    }

    // from both interfaces
    public void move() {
        System.out.println( "waddles on land and swims in water" );
    }
}
```

Penguin Example (cont.)

```
public class TestPenguin {
    public static void main( String[] args ) {
        Penguin myPenguin = new Penguin();

        System.out.println( "Penguin behaviors:" );
        myPenguin.move();
        myPenguin.layEggs();
        myPenguin.nurseYoung();









    }
}
```

Penguin Example (cont.)

```
public class TestPenguin {
    public static void main( String[] args ) {
        Penguin myPenguin = new Penguin();

        System.out.println( "Penguin behaviors:" );
        myPenguin.move();
        myPenguin.layEggs();
        myPenguin.nurseYoung();

        // The penguin object can be treated as either a Bird or a Mammal type
        Bird aBird = myPenguin;
        aBird.layEggs();

        Mammal aMammal = myPenguin;
        aMammal.nurseYoung();
    }
}
```

more later

Defining Interfaces

- A Java interface is a group of constants and abstract methods
- All variables declared in an interface are treated as constants
  - Implicitly public, static, and final
  - Must be initialized at the time of declaration
- All methods in an interface must be abstract

  (Note: Java 8 allows default and static; Java 9 allows private – out of scope)
- Ex:

```
public interface Shape {
  int NUM_DIMENSIONS = 2;
  double calcArea();
  double calcPerimeter();
}
```

← implicitly public, static, final

} implicitly public, abstract

Using Interfaces

- Can***not*** instantiate objects from an interface
- A class uses the implements keyword to adopt an interface
  - Must provide definition for all of the interface's abstract methods or declare itself abstract too

Can interfaces have constructors?

Using Interfaces

- Can***not*** instantiate objects from an interface
- A class uses the implements keyword to adopt an interface
  - Must provide definition for all of the interface's abstract methods or declare itself abstract too

Can interfaces have constructors? No!
Why?

Using Interfaces

- Can***not*** instantiate objects from an interface
- A class uses the implements keyword to adopt an interface
  - Must provide definition for all of the interface's abstract methods or declare itself abstract too

Can interfaces have constructors? No!
Why? Interfaces do not have attributes (used to keep track of states), only constants

Circle Example

```
public class Circle implements Shape {
  private double radius;
  public Circle( double r ) { radius = r; }

  // define abstract methods
  public double calcArea()
  {
      return Math.PI * radius * radius;
  }
  public double calcPerimeter()
  {
      return 2 * Math.PI * radius;
  }

  // example of using the constant
  public void displayDimension()
  {
      System.out.println( NUM_DIMENSIONS );
  }
}
```

Recall:

```
public interface Shape {
  int NUM_DIMENSIONS = 2;
  double calcArea();
  double calcPerimeter();
}
```

Memory Example

- Suppose you want to define different types of memory storage e.g., CD, USB key, hard drive, etc.
- Interface:

```
public interface MemoryInterface
{
  public void writeTo( int location, int value );
  public int readFrom( int location );
  public void loadMemory();
  public int size();
}
```

Memory Example (cont.)

- When using an interface, the class is just like a regular class:
  - Can have attributes
  - Has constructor

```
public class SmallDisk implements MemoryInterface
{
  int[] memArray;
  public SmallDisk( int size )
  {
    memArray = new int[size];
  }

}
```

Memory Example (cont.)

- When using an interface, the class is just like a regular class:
  - Can have attributes
  - Has constructor

- This example class defines all abstract methods
- Can define additional attributes and methods

```
public class SmallDisk implements MemoryInterface
{
  int[] memArray;
  public SmallDisk( int size )
  {
    memArray = new int[size];
  }
  public void writeTo( int location, int value )
  {
    memArray[location] = value;
  }
  public int readFrom( int location )
  {
    return memArray[location];
  }
  public void loadMemory()
  {
    // re-initialize memArray to default values
    memArray = new int[100];
    memArray[0] = 799;
    memArray[1] = 798;
    // etc.
  }
  public int size()
  {
    return memArray.length;
  }
}
```

With Multiple Interfaces …

- A class can implement several interfaces
- An interface can extend other interfaces
  - Ex: public interface A extends B { … }
  - Ex: public interface A extends B, C { … }
  - Promotes code reuse and create an "is-a" relationship between interfaces
  - Inherits all constants and abstract methods
  - Effectively creating a larger contract

Can an interface extend a class?

With Multiple Interfaces …

- A class can implement several interfaces
- An interface can extend other interfaces
  - Ex: public interface A extends B { … }
  - Ex: public interface A extends B, C { … }
  - Promotes code reuse and create an "is-a" relationship between interfaces
  - Inherits all constants and abstract methods
  - Effectively creating a larger contract

Can an interface extend a class? No!

Why?

With Multiple Interfaces …

- A class can implement several interfaces
- An interface can extend other interfaces
  - Ex: public interface A extends B { … }
  - Ex: public interface A extends B, C { … }
  - Promotes code reuse and create an "is-a" relationship between interfaces
  - Inherits all constants and abstract methods
  - Effectively creating a larger contract

Can an interface extend a class? No!

Why? Because we don't want the attributes (state) and implementation details

Access Constants in Interfaces

- A constant defined in an interface can be accessed using the dot operator
  - Syntax:
    InterfaceName.CONSTANT_NAME
    or
    ImplementingClassName.CONSTANT_NAME
  - Ex: recall Shape interface, you can write in the test class:

```
System.out.println( Shape.NUM_DIMENSIONS );
System.out.println( Circle.NUM_DIMENSIONS );
```

- Similar to how Math.PI is accessed from the Math class

iClicker Question

Suppose Employee and Student are classes and Comparable and Serializable are interfaces. Which of the following is valid?

A. class WorkingStudent extends Employee, Student
B. class WorkingStudent extends Employee, Student implements Comparable
C. class WorkingStudent extends Comparable
D. class WorkingStudent extends Employee implements Comparable, Serializable
E. class WorkingStudent extends Employee implements Student

iClicker Question

Suppose Employee and Student are classes and Comparable and Serializable are interfaces. Which of the following is valid?

A. class WorkingStudent implements Student
B. public interface Shape extends Comparable, Serializable
C. public interface WorkingStudent extends Student
D. Comparable student1 = new Comparable();

When to Use Which?

- Abstract class:

- Interface:

When to Use Which?

- Abstract class:
  - Related classes for modeling a family of objects (Animal, Dog, etc.)
  - Sharing code with subclasses, possibly allow partial implementation
  - Need for object state within abstract class's methods
  - Need for non-public visibility modifiers
- Interface:

When to Use Which?

- Abstract class:
  - Related classes for modeling a family of objects (Animal, Dog, etc.)
  - Sharing code with subclasses, possibly allow partial implementation
  - Need for object state within abstract class's methods
  - Need for non-public visibility modifiers
- Interface:
  - Unrelated classes with common behavior (Bird and Airplane)
  - Mimic multiple inheritance behavior
  - Defining a contract of what a class must do (not "how")
  - Define shared constants across multiple classes

More on Method Modifiers

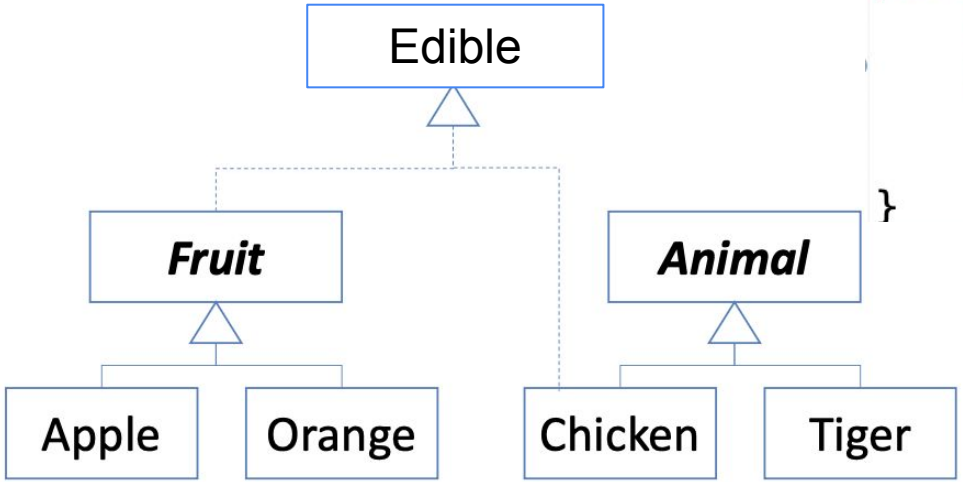
- Interfaces have zero or more of the following:

```
public interface InterfaceName
{
    constants
    abstract methods
    default methods                 // introduced in Java 8
    static methods                  // introduced in Java 8
    private methods                 // introduced in Java 9
    private static methods          // introduced in Java 9
}
```

allows some methods to be implemented

default Methods

- Provide default implementation in an interface
- Mixing abstract and default methods gives developers more flexibility
- Example:




```
public interface Edible {
    void howToEat();
    public default void print() {
        System.out.println( "edible" );
    }
}
```

Fruit and Chicken don't need to implement print()
(can choose to override it if desired)

Edible Example

```
public interface Edible {
    void howToEat();
    public default void print() {
        System.out.println( "edible" );
    }
}

public class Chicken implements Edible {
    public void howToEat()  {
        System.out.println( "chew and swallow" );
    }
}

public class Fruit implements Edible {
    public void howToEat()  {
        System.out.println( "peel, cut, bite" );
    }
    public void print() {
        System.out.println( "yummy" );
    }
}
```

```
public class TestEdible {
    public static void main( String[] args ) {
        Chicken dinner = new Chicken();
        dinner.howToEat();
        dinner.print();
        Fruit dessert = new Fruit();
        dessert.howToEat();
        dessert.print();
    }
}
```

Output:

```
chew and swallow
edible
peel, cut, bite
yummy
```

static Methods

- Provide implementation for utility methods in user-defined interfaces
- Ex:

```
public class TestInterface {
    public static void main(String[] args) {
        // calls using the interface name
        int result = Calculator.add(5, 3);
        System.out.println("The sum is: " + result);
    }
}

interface Calculator {
    // Static method
    static int add(int a, int b) {
        return a + b;
    }
}
```

private Methods

- Build helper methods used internally by other default or static methods within the interface
- Ex:

```
public interface Edible {
    void howToEat();
    private void cut() {
        System.out.println( "cut" );
    }
    public default void print() {
        cut();
        System.out.println( "edible" );
    }
}
```

```
public class Chicken implements Edible {
    public void howToEat()  {
        System.out.println( "chew and swallow" );
    }
}
```

```
public class TestEdible {
    public static void main( String[] args ) {
        Chicken dinner = new Chicken();
        dinner.howToEat();
        dinner.print();
    }
}
```

Output:

```
chew and swallow
cut
edible
```

private static Methods

- Is a utility method used internally by other static or default methods within the same interface
- Ex:

```
// Private static method for common validation logic
private static boolean validateAmount(double amount) {
    if (amount <= 0)
        return false;
    return true;
}
```

in another method of the same interface:

```
static boolean isAmountPositive(double amount) {
    if (!validateAmount(amount))
        return false;
    System.out.println("Amount validation passed");
    return true;
}
```

code in test class:

```
System.out.println("Is 500 valid? " + PaySys.isAmountPositive(500.0));
```

iClicker Question

Which of the following is invalid in Java 9+?

A.
```
interface F {
    default void displayHello(){ System.out.println("Hello"); }
}
```
B.
```
interface F {}
```
C.
```
interface F { int MAX = 200; }
```
D.
```
interface F {
    final int MAX = 200;
    String text;
    void display();
}
```
E.
```
interface F { boolean display(int a); }
```