Interface, 3 definitions used in this class

- (from cs2308): the mechanism that code outside the object uses to interact with the object; the object's public member functions.
- (graphical) user interface (sometimes shortened to “interface”): the means by which the user and a computer system interact, in particular the use of input devices and software.
- Java Interface: a reference type, similar to a class, that contains constants and/or method signatures (methods with empty bodies).

Goal: to separate the interface from the implementation

Example: The Icon interface in Java

- You can use javax.swing.JOptionPane to display message:
  ```java
  JOptionPane.showMessageDialog(null, "Hello, World!");
  ```
- Note the “i” icon on the left:
  ![Icon Icon](image)

- To specify an arbitrary image file:
  ```java
  JOptionPane.showMessageDialog(null, "Hello, World!", "Message", JOptionPane.INFORMATION_MESSAGE, new ImageIcon("globe.gif");
  ```

- What if we want to draw the image using library methods? Here is the declaration of the showMessageDialog method:
  ```java
  public static void showMessageDialog(
    Component parent,
    Object message,
    String title,
    int messageType,
    Icon anIcon);
  ```

- You can use any class that implements the javax.swing.Icon interface type:
  ```java
  public interface Icon {
    int getIconWidth();
    int getIconHeight();
    void paintIcon(Component c, Graphics g, int x, int y);
  }
  ```
Java Interfaces

• In the Java programming language, an Interface is a form or template for a class: the methods do not have implementations (they are like C++ prototypes).
• The methods are implicitly public.
• An interface may contain fields, but these are implicitly static and final (named constants).
• A class implements the interface type by providing an implements clause and supplying implementations for the methods that are declared in the interface type.
• An interface can be used as a type (for variables, parameters, etc)

Java permits an object instance of a class that implements an Interface to be assigned to a variable or parameter of that type.

Example: A new class that implements Icon

• The javax.swing.ImageIcon class implements Icon (see the api)
• Let’s design a class MarsIcon that implements the Icon interface type (see Horstmann for imports and detailed explanation):

```java
public class MarsIcon implements Icon {
    public MarsIcon(int aSize) {
        size = aSize;
    }
    public int getIconWidth() { return size; }
    public int getIconHeight() { return size; }
    public void paintIcon(Component c, Graphics g, int x, int y) {
        Graphics2D g2 = (Graphics2D) g;
        Ellipse2D.Double planet = new Ellipse2D.Double(x, y, size, size);
        g2.setColor(Color.RED);
        g2.fill(planet);
    }
    private int size;
}
```

Note it provides definitions for the three Icon methods

Example: Using MarsIcon in showMessageDialog

• This driver uses our MarsIcon class to make the dialog:

```java
import javax.swing.*;
public class IconTester {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hello, Mars!", "Message", JOptionPane.INFORMATION_MESSAGE, new MarsIcon(50));
        System.exit(0);
    }
}
```

I got this when I ran the code on my mac:

Class diagram

• the Icon interface type and the classes that implement it:
  ✦ A—|---> B means class A implements interface B
  ✦ A—|---> B means class A uses class/interface B
Polymorphism

- Upcasting:
  - Permitting an object of a class type to be treated as an object of any interface type it implements:

```
Icon x = new MarsIcon(50);
```

- Polymorphism:
  - The ability of objects belonging to different class types to respond to method calls of the same name, but with an appropriate type-specific behavior.
  - It allows many types (implementing the same Interface) to be treated as if they were one type, and a single piece of code to work on all those different types equally, yet getting type-specific behavior for each one.

Polymorphism Example (using an Interface):

```
public interface Instrument {
    void play(String n);
}

public class Wind implements Instrument {
    public void play(String n) {
        System.out.println("Wind.play() " + n);
    }
}

public class Stringed implements Instrument {
    public void play(String n) {
        System.out.println("Stringed.play() " + n);
    }
}

public class Percussion implements Instrument {
    public void play(String n) {
        System.out.println("Percussion.play() " + n);
    }
}
```

```
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute);  // upcasting to Instrument
        tune(violin); // upcasting to Instrument
    }
}
```

```
What if we didn’t have polymorphism?

- We could overload `tune` to work for each type of `Instrument`

```
public class Music {
    public static void tune(Wind i) {
        i.play("Middle C");
    }
    public static void tune(Stringed i) {
        i.play("Middle C");
    }
    public static void tune(Percussion i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute); // No upcasting necessary
        tune(violin);
    }
}
```

```
Output:

Wind.play() Middle C
Stringed.play() Middle C
```

Polymorphism Example continued

```
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute);  // upcasting to Instrument
tune(violin); // upcasting to Instrument
    }
}
```

```
What is output?

Wind.play() Middle C
Stringed.play() Middle C
```

Polymorphism: in tune, i is an Instrument, but it calls the play method based on the specific type of the object it receives.
But we do have upcasting and polymorphism:

- We can get the same effect with just one tune method

```java
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Percussion snaredrum = new Percussion();
        tune(flute); // upcasting
        tune(violin);
        tune(snaredrum);
    }
}
```

Output: polymorphism

Wind.play() Middle C
Stringed.play() Middle C
Percussion.play() Middle C

Polymorphism in JOptionPane.showMessageDialog

- Consider implementing the showMessageDialog method:

  ```java
  public static void showMessageDialog( . . . Icon anIcon);
  ```

- The width of the dialog box depends on the width of anIcon.
- But anIcon could refer to a MarsIcon or to an ImageIcon, how do we call the proper method?
- Since the type of anIcon must be a class that implements Icon, we know it must have a getIconWidth() method that returns the width of the Icon, so we can use that: anIcon.getIconWidth()
- During run-time, the Java interpreter determines the class type of the object anIcon is referring to, and uses the implementation of getIconWidth from that class.

Implementing the Java Comparable Interface

- Assume you want to sort an ArrayList of custom objects (instances of some class you created).
- The following static method is available in the Java API:

  ```java
  void Collections.sort(List<T> list)    // for ArrayLists
  ```

- All elements in the ArrayList must implement the java.lang.Comparable<T> interface:

  ```java
  int compareTo(T o);     //T is your custom class
  ```

The call object1.compareTo(object2) is expected to return a negative number if object1 should come before object2, zero if the objects are equal, and a positive number otherwise

Sorting with Comparable, example

```java
import java.util.*;
public class Student implements Comparable<Student> {
    private String name;
    private String major;
    private int idNumber;
    private float gpa;
    public Student(String name, String major, int idNumber, float gpa) {
        this.name = name; this.major = major;
        this.idNumber = idNumber; this.gpa = gpa;
    }
    public String getName() { return name; }
    public float getGpa() { return gpa; }
    public String toString() {
        return "Student: " + name + " " + major + " " + idNumber + " " + gpa;
    }
    public int compareTo(Student rhs) {
        return name.compareTo(rhs.name);
    }
}
```

This will sort by name. compareTo is already defined in String, so we can reuse it.
Implementing the Java **Comparator** Interface

- Assume you want to sort the ArrayList of students by gpa, but you don’t want to reimplement `compareTo`.

- The following static method is available in the Java API:
  ```java
  void Collections.sort(List<T> list, Comparator<T> c)
  ```

- The `java.lang.Comparator<T>` interface:

  ```java
  int compare(T obj1, T obj2); // T is your custom class
  ```

  Compares `obj1` to `obj2` for order. Returns a negative number, zero, or a positive number depending on whether `obj1` is less than, equal to, or greater than `obj2` in the particular sort order.

To sort by gpa, define a new class that implements Comparator as follows:

```java
public class StudentByGpa implements Comparator<Student> {
    public int compare(Student lhs, Student rhs) {
        float lhsGpa = lhs.getGpa();
        float rhsGpa = rhs.getGpa();
        if (lhsGpa < rhsGpa) return -1;
        if (lhsGpa == rhsGpa) return 0;
        return 1;
    }
}
```

To sort by name, define another Comparator as follows:

```java
public class StudentByName implements Comparator<Student> {
    public int compare(Student lhs, Student rhs) {
        return lhs.getName().compareTo(rhs.getName());
    }
}
```

Output:

- **Before:**
  - Student: Doe, J Math 1234 3.6
  - Student: Carr, M CS 1000 2.7
  - Student: Ames, D Business 2233 3.7

- **After:**
  - Student: Ames, D Business 2233 3.7
  - Student: Carr, M CS 1000 2.7
  - Student: Doe, J Math 1234 3.6
Anonymous objects and classes

- **Anonymous objects**: no need to name an object used only once:
  ```java
  Collections.sort(a, new StudentByGpa());
  ```

- **Anonymous classes**: no need to name a class used only once:
  ```java
  Comparator<Student> comp = new Comparator<Student>() {
      public int compare(Student lhs, Student rhs) {
          return lhs.getName().compareTo(rhs.getName());
      }
  };
  ```

  The right-hand side expression defines a temporary class with no name that implements `Comparator<Student>`, and constructs one object of that class.

Anonymous classes

- **Anonymous classes** can be returned by a function:
  ```java
  public class Student {
      ...
      public static Comparator<Student> compByName() {
          return new Comparator<Student>() {
              public int compare(Student lhs, Student rhs) {
                  return lhs.getName().compareTo(rhs.getName());
              }
          };
      }
      public static Comparator<Student> compByGpa() {
          return new Comparator<Student>() {
              public int compare(Student lhs, Student rhs) {
                  return Math.round(lhs.getGpa() - rhs.getGpa());
              }
          };
      }
  }
  ```

  ```java
  Collections.sort(a, Student.compByGpa());
  ```

Inheritance

- A way to reuse code from existing classes by extending an existing class with new fields and methods
- Classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, or parent classes. The resulting classes are known as derived classes, subclasses or child classes.
- The relationships of classes through inheritance gives rise to a hierarchy.
- In Java, each class has exactly one superclass. If none are specified, then `java.lang.Object` is the superclass.
- Note: In Java, constructors are NOT inherited.

Simple Example of Inheritance

```java
public class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }
}
public class CleanserTester {
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        System.out.println(x);
    }
}
```
Simple Example of Inheritance

```java
public class Detergent extends Cleanser {
    // Change (override) a method:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Call superclass version
    }
    public void foam() { append(" foam()"); } // Added method
}
public class DetergentTester {
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute(); x.apply(); x.scrub(); x.foam();
        System.out.println(x);
        CleanserTester.main(args);
    }
}
```

Output:
```
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Cleanser dilute() apply() scrub()
```

Invoking Superclass Fields and Methods

- Cannot access superclass fields if they are private:
  ```java
  public class Detergent extends Cleanser {
      public String toString() { return "Detergent: " + s; }
      //ERROR: s is private
  }
  ```

- But be careful when calling superclass method:
  ```java
  public class Detergent extends Cleanser {
      public String toString() {return "Detergent: " + toString();}
      //ERROR: recursive call!!
  }
  ```

- Correct:
  ```java
  public class Detergent extends Cleanser {
      public String toString() {
          return "Detergent: " + super.toString();
      }
  }
  ```

General convention

- Fields are private
  - Not even subclasses should access these directly
- Methods are public
  - This is so other classes, including subclasses can access them.
- Overriding a method:
  - Writing a new instance method in the subclass that has the same signature as the one in the superclass.
  - Any instance of the subclass will use the method from the subclass
  - Any instance of the superclass will use the method from the superclass
  - The subclass can call the superclass method using "super.method()"

Initialization

- Java automatically inserts calls to the (default) superclass constructor at the beginning of the subclass constructor.

```java
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}
class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}
class Cartoon extends Drawing {
    public Cartoon() {
        System.out.println("Cartoon constructor");
    }
}
public class CartoonTester {
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

Output:
```
Art constructor
Drawing constructor
Cartoon constructor
```

So constructors are not inherited, they are called from the constructors of the subclass.
Initialization

• If your class doesn’t have default (no arg) constructors, or if you want to call a superclass constructor that has an argument, you must explicitly write the calls to the superclass constructor using the super keyword and the appropriate argument list.

```java
class Game {
   int x;
   Game(int i) {
      x = i;
      System.out.println("Game constructor");
   }
}
class BoardGame extends Game {
   BoardGame(int i) {
      super(i);
      System.out.println("BoardGame constructor");
   }
}
public class Chess extends BoardGame {
   Chess() {
      super(11);
      System.out.println("Chess constructor");
   }
}
```

Access specifiers

• keywords that control access to the definitions they modify
  ✦ public: accessible to all other classes
  ✦ private: accessible only from within the class in which it is defined
  ✦ package (unspecified, default): accessible only to other classes in the same package
  ✦ protected: accessible to all classes derived from (subclasses of) the class containing this definition, even if the class is in another package.
  Note: protected also provides package access!!!

• Classes can only be public or unspecified (which is package)

Java.lang.Object

• some commonly used and/or overridden methods:
  ✦ toString: Returns a string representation of the object. You should override this if you want a displayable version of the objects of your class.
  ✦ equals: Indicates whether some other object is "equal to" this one. For your class, it will use ==, unless you override it.
  ✦ clone: Creates and returns a copy of this object. Make your class implement Cloneable to use a default version of this method.

Polymorphism

• Upcasting:
  ✦ Permitting an object of a subclass type to be treated as an object of any superclass type.

```java
Cleanser x = new Detergent();
```

• Polymorphism:
  ✦ The ability of objects belonging to different types to respond to method calls of the same name, each one according to an appropriate type-specific behavior.
  ✦ It allows many types (derived from the same superclass) to be treated as if they were one type, and a single piece of code to work on all those different types equally, yet getting type-specific behavior for each one.

Very similar to polymorphism with Interfaces
Polymorphism Example (using Inheritance):

- Wind, Stringed and Percussion are Instruments

```java
public class Instrument {
    void play(String n) {
        System.out.println("Instrument.play() " + n);
    }
}
public class Wind extends Instrument {
    void play(String n) {
        System.out.println("Wind.play() " + n);
    }
}
public class Stringed extends Instrument {
    void play(String n) {
        System.out.println("Stringed.play() " + n);
    }
}
public class Percussion extends Instrument {
    void play(String n) {
        System.out.println("Percussion.play() " + n);
    }
}
```

Example continued

```java
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute); // upcasting to Instrument
        tune(violin); // upcasting to Instrument
    }
}
```

What is output?

```
Wind.play() Middle C
Stringed.play() Middle C
```

What is output?

```
Instrument.play() Middle C
Instrument.play() Middle C
```

Polymorphism: in tune, i is an Instrument, but it calls the play method based on the specific type of the object it receives.

Dynamic (run-time) binding

- Given the definition of tune, how does the compiler know which definition of the play method to call? Instrument? Wind? Stringed?

```java
public static void tune(Instrument i) {
    i.play("Middle C");
}
```

- It will differ depending on the specific type of each argument passed to i.
- This cannot be determined at compile time.

- Binding: connecting the method call to a method definition.

  - Static binding: done at compile time (play binds to Instrument.play)
  - Dynamic binding: at run-time, the JVM determines the actual type of i and uses its play() definition. It can vary for each invocation of tune.
  - If the actual type of i does not define "play()", the JVM looks for the nearest definition in its superclass hierarchy.

Abstract methods and classes

- An abstract class is a class that cannot be instantiated, but it can be subclassed
- It may or may not include abstract methods:
- An abstract method is a method that is declared in a class without a method body, like this:

```java
abstract void f(int x);
```
- If a class contains an abstract method, it must be declared to be an abstract class.
Abstract methods and classes, example

- Any class that inherits from an abstract class must provide method definitions for all the abstract methods in the base class.
  ✦ Unless the derived class is also declared to be abstract
- The Instrument class can be made abstract:
  ✦ No longer need “dummy” definitions for abstract methods
  ✦ Common code (shared by subclasses) can be put in the abstract superclass

```java
abstract class Instrument {
    private int i; // Storage allocated in each subclass
    abstract void play(String n); // subclass must define
    String what() {
        return "Instrument";    // when would this be executed?
    }
    abstract void adjust();    // subclass must define
}
```

Interface or Abstract class?

- Interface
  ✦ Pro: can be implemented by any number of classes
  ✦ Con: each class must have its own code for the methods, common method implementations must be duplicated in each class
- Abstract Class
  ✦ Pro: subclasses do not have to repeat common method implementations, common code is in the abstract superclass
  ✦ Con: Cannot be multiply inherited.