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!");
  ```

- To specify an arbitrary image file:

  ```java
  JOptionPane.showMessageDialog(null, "Hello, World!",
  "Message",
  JOptionPane.INFORMATION_MESSAGE,
  new ImageIcon("globe.gif"));
  ```

Example: The Icon interface in Java

- 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 (a) providing an implements clause and (b) 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);
    }
}
```

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:
    
    ```java
    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):

- Wind, Stringed and Percussion are Instruments with a play(String) method.

```java
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
    }
}
```

Output:

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

What if we didn’t have polymorphism?

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

```java
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
```
But we do have upcasting and polymorphism:

- We can get the same effect with just one tune method.
- Add a snaredrum Percussion object and call tune on it.

```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`

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);
    }
}
```
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

Sorting with Comparator, sort by gpa

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

- **Anonymous objects**: no need to name an object used only once:
  
  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 (1) defines a temporary class with no name that implements Comparator<Student>, and (2) constructs one object of that class (note keyword “new”).

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());
            }
        };
      }
  }
  ```

  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; }
}
```

```java
public class CleanserTester {
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        System.out.println(x);
    }
}
```

**Output:**

Cleanser dilute() apply() scrub()
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);
    }
}
```

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

Invoking Superclass Fields and Methods

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

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

- Correct:
```
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.

```
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");
    }
}
```

Output:
```
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.
    - You do not need to override the clone method, but the documentation recommends that you do ( you can just call super.clone() )

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 inherit from 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

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?
- 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.

Collections in Java

• A collection is a data structure for holding elements
• java.util.Collection<T> is an interface implemented by many classes in Java. It has 3 extended interfaces:
  ✦ List<T> implemented by ArrayList<T> and LinkedList<T>, etc.
  ✦ Set<T> implemented by HashSet<T> and others
  ✦ Queue<T> implemented by PriorityQueue<T> and others

• Some methods in the Collection interface:
  ✦ isEmpty(), contains(e), add(e), remove(e), iterator()

Maps in Java

• A map is an object that associates keys with values.
• A map cannot contain duplicate keys; each key can map to at most one value.
• java.util.Map<K,V> is an interface implemented by many classes in Java
  ✦ HashMap<K,V>, HashTable<K,V>
  ✦ TreeMap<K,V>

• Some methods in the Map interface:
  ✦ isEmpty, containsKey(e), put(k,v), get(k), remove(k)
  ✦ values(): Collection<V>, keySet(): Set<K>
Linked Lists in the Java Library

- An linked list supports **efficient** insertion and removal at any location:

![Diagram of linked list]

- `java.util.LinkedList<T>` is an class that implements `List<T>`
  - `void add(T e)` appends to the end of the list
  - `T get(int i)` and `void set(int i, T e)` are supported, but not efficient. Each call traverses the list.
  - Use an iterator to access elements in the middle.

Collections and Iterators: example

```java
public class ListIteratorTester {
    public static void main(String[] args) {
        LinkedList<String> countries = new LinkedList<String>();
        countries.add("Belgium");
        countries.add("Italy");
        countries.add("Thailand");
        Iterator<String> iterator = countries.iterator();
        while (iterator.hasNext()) {
            String country = iterator.next();
            System.out.println(country);
        }
        System.out.println();
        // Or use a for each loop
        for(String country : countries) {
            System.out.println(country);
        }
        System.out.println();
        // An Iterator can also remove elements:
        iterator = countries.iterator(); //reset to first element
        iterator.next();
        iterator.remove(); //removes second element
    }
}
```

Iterators in Java

- An iterator is an object that cycles through all the elements in a collection. It points to an element of the collection.
- `java.util.Iterator<T>` is an interface with the following methods:
  - `public T next()` returns the next element in the collection (and advances)
  - `public boolean hasNext()` returns true if next() is not done.
  - `public void remove()` (Optional) removes the last element returned by next.
- You can get Iterators from Collections (and Maps):
  - `ArrayList<Double> x = new ArrayList<Double>;
    Iterator<Double> it = x.iterator();`
  - `HashMap<String,Double> hm = new HashMap<String,Double>;
    Iterator<Double> it = hm.values().iterator();`

Diagram of Collections and Maps in Java
Exceptions:
Error Handling in Java

- Run time errors
  ✦ It is difficult to recover gracefully from run-time errors that occur in the middle of a program.
  ✦ At the point where the problem occurs, there often isn’t enough information in that context (the method) to resolve the problem.
  ✦ In Java, that method hands off the problem out to a higher context (a calling method) where someone is qualified to make the proper decision

- If the error can be resolved in the immediate context where it occurs, it is NOT called an exception.

Exception semantics - 1

- When an error occurs inside a method, the method creates an exception object.
  ✦ could be in a library method or a user-defined method
- Reporting an exception to the runtime system is called throwing an exception.
- When a method throws an exception,
  ✦ the current path of execution is interrupted, and
  ✦ the runtime system attempts to find an appropriate place to continue executing the program.

Exception semantics - 2

- The runtime system searches the call stack for an appropriate exception handler
  ✦ the call stack: the list of methods that have been called and are waiting for the current method to return.
  ✦ A calls B that calls C that calls D: The call stack contains A, B, C and D with D on the top.
- The runtime system is looking for a previous method call that is embedded in a block that has an exception handler associated with it.
  ✦ It starts at the top of the call stack and goes down (in reverse order in which the methods were called)

Exception semantics - 3

- The runtime system is searching for an appropriate exception handler
  ✦ An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
- The first exception handler encountered that matches the exception is said to catch the exception.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system terminates the program.
  ✦ And usually the exception is output to the screen
Exception syntax: how to throw an exception

- To throw an exception, use the keyword throw.
- To create an exception, use the appropriate constructor.

```java
if (t==null)
    throw new NullPointerException();
```

- Exception classes can be found in the API website: see `java.lang.Exception`

Exception syntax: how to catch an exception

- To catch an exception, use the try-catch block.
- Surround the code that might generate an exception in the try block.
- Make an exception handler (a catch clause) for every type of exception you want to catch.

```java
try {
    // Code that calls methods that might throw exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
} catch(Type3 id3) {
    // Handle exceptions of Type3
}
// etc...
```

- Each catch clause is like a little method that takes one argument of a particular type.
- The parameters (id1, id2, and so on) can be used inside the handler, just like a method argument.
- If the handler catches an exception, its catch block is executed, and the flow of control proceeds to the next statement after (outside) the try/catch.
  - only the first matching catch clause is executed.

Exception simple example

```java
import java.io.*;
public class ExceptionTester{
    public static void main(String args[]){
        try{
            int a[]= new int[2];
            System.out.println("Access element three :" + a[3]);
            System.out.println("After element access");
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown ": + e);
        }
        System.out.println("Out of the block");
    }
}
```

- What part of the code throws the exception?
- Output:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```
The exception specification: being civil

- In Java, you are (strongly!) encouraged to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method.
  - Then the caller can know exactly what catch clauses to write to catch all potential exceptions.
- The exception specification states which exceptions are thrown by a method.
  - void f() throws TooBig, TooSmall, DivZero { //...
  - Also use the @throws tag in the javadoc comment to describe these in more detail (when/why each one is thrown).
- Catch or specify requirement: If the method throws exceptions, it must handle them or specify them in the signature.
  - Otherwise it’s a compiler error.

Catch or Specify: example

```java
public class ListOfNumbers {
    private ArrayList<Integer> ints;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        ints = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            ints.add(i);
        }
    }

    public void writeList() throws IOException {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt");
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + ints.get(i));
        }
        out.close();
    }
}
```

This compiles with no errors.

Catch or Specify: solution 1

```java
public class ListOfNumbers {
    private ArrayList<Integer> ints;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        ints = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            ints.add(i);
        }
    }

    public void writeList() throws IOException {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt");
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + ints.get(i));
        }
        out.close();
    }
}
```

ListOfNumbers.java:16: error: unreported exception IOException; must be caught or declared to be thrown
  PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt");)

ListOfNumbers.java:16: error: unreported exception IOException; must be caught or declared to be thrown
  PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt");)

This compiles with no errors.

Catch or Specify: solution 2

```java
public class ListOfNumbers {
    private ArrayList<Integer> ints;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        ints = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            ints.add(i);
        }
    }

    public void writeList() throws IOException {
        PrintWriter out = null;
        try {
            out = new PrintWriter(new FileWriter("OutFile.txt");
            for (int i = 0; i < SIZE; i++) {
                out.println("Value at: " + i + " = " + ints.get(i));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (out != null)
            out.close();
    }
}
```

This compiles with no errors.
Runtime Exceptions: an exception to the rule

- RuntimeExceptions are a special (sub)class of Exceptions.
  ✦ They are thrown automatically by Java in certain contexts
  ✦ This is part of the standard run-time checking that Java performs for you
- These exceptions are “unchecked exceptions”, they do not need to conform to the “Catch or specify rule.
  ✦ Methods are not required to indicate if they might throw one
  ✦ Methods are not required to try to catch them
- What if they are not caught?
  ✦ If a RuntimeException gets all the way out to main() without being caught, printStackTrace() is called for that exception as the program exits

You can create your own exceptions

- If one of the Java Exceptions is not appropriate for your program, you can create your own Exception classes
  ✦ The class must inherit from an existing exception class, preferably one that is close in meaning to your new exception.

```java
class SimpleException extends Exception {}
class SimpleExceptionDemo {
    public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException();
    }
}
class DemoDriver {
    public static void main(String[] args) {
        SimpleExceptionDemo sed = new SimpleExceptionDemo();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.err.println("Caught it!");
        }
    }
}
```