A few uses of Inheritance in Java

The Object class (review)

- Java's **`Object`** class captures things that are common to all objects in Java. For example
  - **`Object`**'s constructor communicates with the operating system to allocate memory at runtime
    - **`public Object()`** is called for ***every*** new object
- **`Object`** is the root of all other classes
  - All classes extend **`Object`**
  - Before your constructor executes, **`super()`** is called which calls Object's constructor, even with this code

```
class A {}
new A();
```

`EmptyClass` *inherits the methods of Object*

```
// This class inherits Object's 11 methods
public class EmptyClass extends Object  {
  super(); // These two are always present implicitly
}

// Send messages when methods are implemented in Object
EmptyClass one = new EmptyClass();
System.out.println(one.toString());
System.out.println(one.hashCode());
System.out.println(one.getClass());
System.out.println(one.equals(one));
```

Inheritance helps with the Swing framework

- Inheritance allows one class obtains behavior (methods) and attributes (instance variables) from an existing class *get something for nothing*

```
public class ImAJFrame2 extends JFrame {
}
```

Inherit methods and fields


```
import javax.swing.JFrame;

public class Try2JFrames {

  public static void main(String[] args)   {
    JFrame window1 = new JFrame();
    window1.setTitle("I'm a JFrame");
    window1.setSize(200,100);
    window2.setLocation(10, 0);
    window1.setVisible(true);

    ImAJFrame2 window2 = new ImAJFrame2();
    window2.setTitle("I'm a JFrame too");
    window2.setSize(200,100);
    window2.setLocation(210, 0);
    window2.setVisible(true);
   }
}
```

Has-A or Is-A

- "HAS-A" relationships represent containment within an object; realized by instance variables

```
public class MyList implements ListModel {
   private List<Songs> things;
}
```

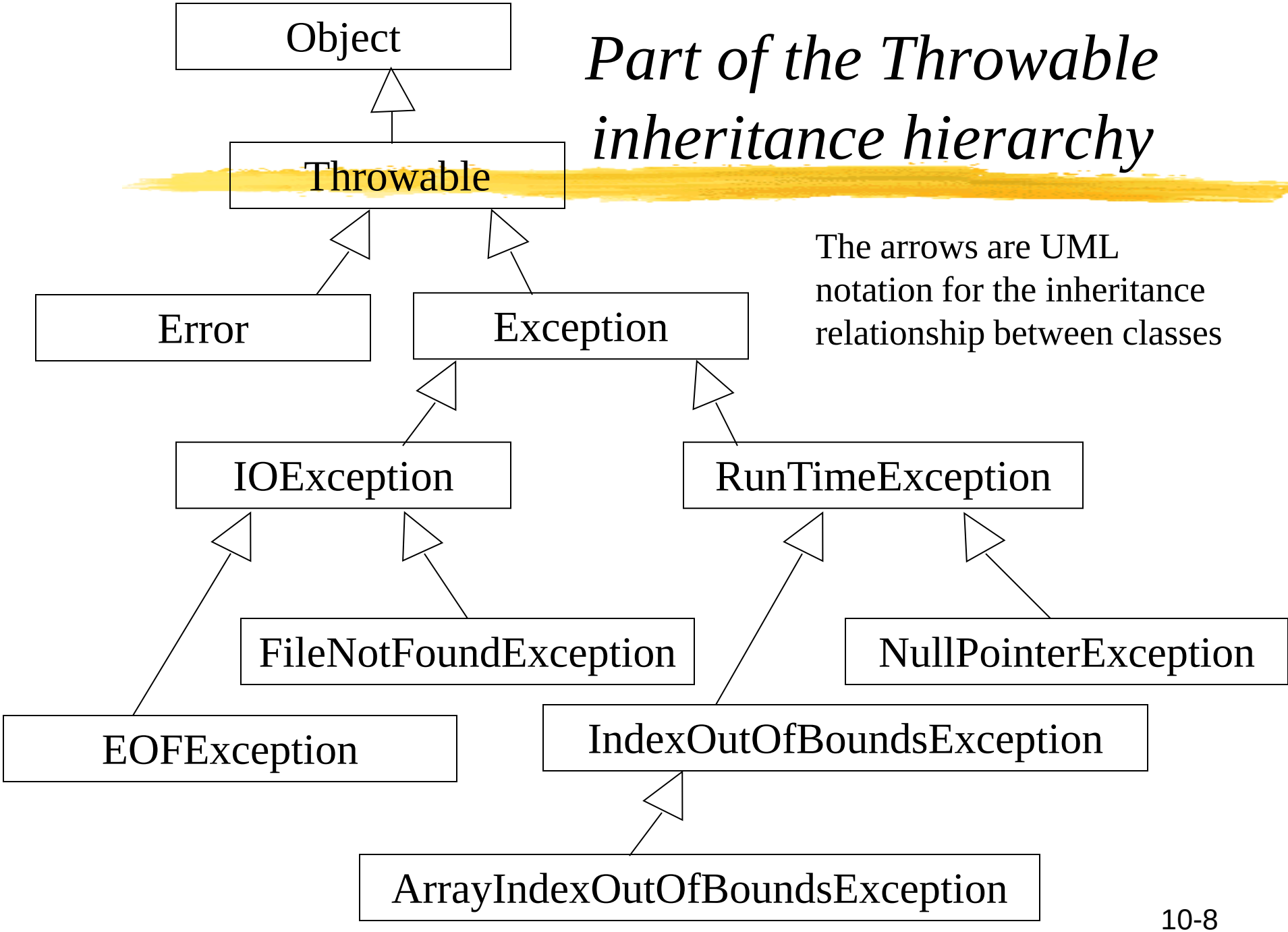
  - MyList object "has-a" List in it, and therefore can use it
- "IS-A" relationships represent supersets of abilities; realized by inheritance
  - **`ImAJFrame2`** IS-A **`JFrame`**

Another example: Java's Exception Hierarchy

- Exceptions handle weird and awkward things
  - Some are standard exceptions that must be
    - caught with try and catch blocks,
    - or declared to be thrown in every method
      - The **read** message won't compile unless you do one or the other

```
public static void main(String[] args) {
   try {
     System.in.read();
   }
   catch(IOException e) {
     System.out.println("read went wrong");
   }
}
```

Part of the Throwable inheritance hierarchy



The arrows are UML notation for the inheritance relationship between classes

Base and derived classes

- **`Object`** is the super class of all classes
- The **`Throwable`** class is the superclass of all errors and exceptions in the Java language
- Error indicates serious problems that a reasonable application should *not* try to catch.
- **`Exception`** and its subclasses are a form of **`Throwable`** that indicates conditions that a reasonable application might want to catch

Java's Throwable hierarchy is wide and deep (many)

- See `http://download.oracle.com/javase/6/docs/api/java/lang/Throwable.html`
- **`RuntimeException`** is the superclass of exceptions that can be thrown during the normal operation of the Java Virtual Machine
- **`IOException`** classes are related to I/O
- **`IndexOutOfBounds`** exceptions indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range

Our own Exception classes

- A method can throw an existing exception

```
/**
  * @return element at the top of this stack
  */
  public E top() throws EmptyStackException {
   // The EmptyStackException is in java.util.*;
   if(this.isEmpty())
     throw new EmptyStackException();
   // If this stack is empty, return doesn't happen
   return myData.getFirst();
 }
```

- Declare what the method throws, then throw a new exception -- The superclass constructor does the work

Writing our own Exception classes

- Consider a `NoSongsInQueueException` method in class `PlayList` to inform users they sent a `playNextSong` message when the playlist has 0 songs

```
public void playNextSong() {
  if (songQueue.isEmpty())
    throw new NoSongsInQueueException();
  // ...
}
```

- You could start from scratch
  - find the line number, the file name, the methods, …
    - Or you could *extend* an Exception class

Create a new Exception

```
// The work of exception handling will be extended to our
// new NoSongsInQueueException. All we have to do is imple-
// ment one or two constructors that calls the superclass's
// constructor (RuntimeException here) with super.
class NoSongsInQueueException extends RuntimeException {

  public NoSongsInQueueException() {
    // Send a message to RuntimeException() constructor
    super();
  }

  public NoSongsInQueueException(String errorMessage) {
    // Send a message to RuntimeException(String) constructor
    super("\n " + errorMessage);
  }
}
```

super calls the superclass constructor, which in this new exception class is **RunTimeException**

Using our new default Constructor

Download

```
class PlayList {
  Queue songQueue = new LinkedBlockingQueue();
  public PlayList() {
    songQueue = new LinkedBlockingQueue();
  }
  public void playNextSong() {
    if (songQueue.isEmpty())
      throw new NoSongsInQueueException();
    // ...
  }
}

PlayList pl = new PlayList();
pl.playNextSong();

Exception in thread "main" NoSongsInQueueException
at PlayList.playNextSong(NoSongsInQueueException.java:36)
at NoSongsInQueueException.main(NoSongsInQueueException.java:12)
```

Use constructor with string parameter

```
class PlayList {
  Queue songQueue = new LinkedBlockingQueue();
  public PlayList() {
    songQueue = new LinkedBlockingQueue();
  }
  public void playNextSong() {
    if (songQueue.isEmpty())
      throw new NoSongsInQueueException(
            "Hey, there ain't no songs in this PlayList");
  }
}

PlayList pl = new PlayList();
pl.playNextSong();
```

```
Exception in thread "main" NoSongsInQueueException:
Hey, there ain't no songs in this PlayList
at PlayList.playNextSong(NoSongsInQueueException.java:36)
at NoSongsInQueueException.main(NoSongsInQueueException.java:12)
```

java.io uses inheritance too

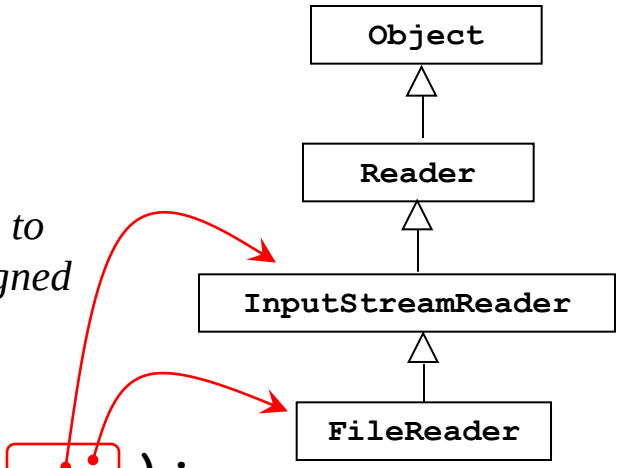
- The `BufferedReader` class is often used with `InputStreamReader`
  - `BufferedReader` has a `readLine` method
- `BufferedReader` is used for input from keyboard or a text file

```
InputStreamReader bytesToChar
             = new InputStreamReader(System.in);

BufferedReader objectWithReadline
             = new BufferedReader(bytesToChar);

System.out.print("Enter a number: ");
String line = objectWithReadline.readLine();
double number = Double.parseDouble(line);
```

Constructor takes a Reader parameter or any class that extends Reader

- Since the **`BufferedReader`** constructor a Reader parameter **`public BufferedReader(Reader in)`**
  - any class that extends **`Reader`** can be passed as an argument to the **`BufferedReader`** constructor
    - **`InputStreamReader`** such as Java's **`System.in`** object
      - For keyboard input
    - **`FileReader`**
      - for reading from a file

Part of Java's inheritance hierarchy. References to InputStreamReader and FileReader can be assigned to a Reader reference (one-way assignment)




```
BufferedReader hasReadline
        = new BufferedReader(    );
```

New Listener

- WindowListener has 7 methods to implement
- We only need **`WindowClosing`**
- When users close the window, have that method ask the user to save files, quit without save, or cancel
  - Need to change **`defaultCloseOperation`** to **`DO_NOTHING_ON_CLOSE`**

```
this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

Add a WindowListener to this by implementing all 7 methods

```
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JFrame;

public class NewListener extends JFrame {

  public static void main(String[] args) {
    JFrame frame = new NewListener();
    frame.setVisible(true);
  }

  public NewListener() {
    setTitle("Let someone list to me");
    setSize(200, 150);
    setLocation(100, 100);
    this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    this.addWindowListener(new RespondToWindowEvents());
  }
```

Or extend WindowAdapter

- To help, you can have the **`WindowListener`** extend **`WindowAdapter`** to save writing all 7 methods
- This gives you all 7 as method stubs that do nothing
- Then override **`WindowClosing`**
- To terminate program

```
System.exit(0);
```

ConfirmMessageDialog

```
private class RespondToWindowEvents extends WindowAdapter {
   public void windowClosing(WindowEvent evt) {
     int userInput
           = JOptionPane.showConfirmDialog(null, "Save data?");

     assert (userInput == JOptionPane.NO_OPTION
         || userInput == JOptionPane.YES_OPTION
         || userInput == JOptionPane.CANCEL_OPTION);

     // Do whatever is appropriate for your application

     // You will want to terminate the program after saves
     System.exit(0);
   }
 }
```

Benefits of Inheritance

- According to Sun's Java website, inheritance offers the following benefits:
  - Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
  - Programmers can implement superclasses called ***abstract classes*** that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

Purpose of Abstract Classes

- Why have Abstract classes?
  - Define generic behaviors
  - Can implement the common behaviors
- Summary of how to guarantee that derived classed implement certain methods
  - Make the method abstract, do not implement it
  - Use the key word abstract in the method heading and end with ; rather than { }

Example of Simple Abstract Class

```
public abstract class AnimalKingdom {
   private String phylum;

   public AnimalKingdom(String p) {
      phylum = p;
   }

   public String getPhylum(){
     return phylum;
   }

   public abstract void eat();
}
```

Particularities of Abstract Classes

- *Cannot* be instantiated
- A class can be declared abstract even though it has no abstract methods
- You can create *variables* of an abstract class
  - it must reference a concrete (nonabstract) subclass

```
Animal giraffe = new Giraffe("Chordata");
```

...More particularities

- A subclass cannot access the private fields of its superclass (might want to use protected access modifier to do so, or private with getters and setters)
- If a subclass does not implement the abstract methods of its parent, it too must be abstract
- Protected methods and fields are known throughout the package, and to all subclasses even if in another package

Summary of Access Modifiers

Modifier	Visibility
private	None
None(default)	Classes in the package
protected	Classes in package and subclasses inside or outside the package
public	All classes

Another consideration

- You can not reduce visibility
  - you can override a private method with a public one
  - you can not override a public method with a private one

Uses of inheritance *continued*

- You can print any Object with toString
  - Inheritance is one feature that distinguishes the object-oriented style of programming
  - At a minimum, because every class extends **`Object`**, every class is guaranteed to understand the **`toString`** message. If a class does not override **`toString`**, the **`toString`** method in the **`Object`** class executes
- Inheritance gives us polymorphic messages
  - Inheritance is one way to implement polymorphism (Java interfaces are the other way). Polymorphism allows the same message to be sent to different types of objects for behavior that is appropriate to the type

Design Principle

- Favor object composition over class inheritance

Inheritance is a cool way to change behavior. But we know that it's brittle because the subclass can easily make assumptions about the context in which a method it overrides is getting called. …

Composition has a nicer property. The coupling is reduced by just having some smaller things you plug into something bigger, and the bigger object just calls the smaller object back. …

Or read all this page

http://www.artima.com/lejava/articles/designprinciples4.html#resources

Example of bad use of Inheritance

- Stack<E> extends Vector<E>

```
Stack<Integer> s = new Stack<Integer>();
s.push(5);
s.push(1);
s.push(4);
s.push(2);
s.push(3);
System.out.println(s);
Collections.shuffle(s);
System.out.println(s);
s.remove(2);
System.out.println(s);
s.insertElementAt(-999, 2);
System.out.println(s);
```

Output (is this LIFO?)

```
[5, 1, 4, 2, 3]
[4, 2, 5, 3, 1]
[4, 2, 3, 1]
[4, 2, -999, 3, 1]
```