Lecture 23: Interfaces; Event Programming

CS 170, Section 000
24 November 2009
Lecture Plan

- Homework 7 questions?

  - Multiple inheritance: interfaces (parts of ch. 11)
    - Example: Event programming (parts of ch. 15)

- Exceptions (chapter 18)
Recall: Superclasses and Subclasses

---

**GeometricObject**
- color: String
- filled: boolean
- dateCreated: java.util.Date

+GeometricObject()
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

**Circle**
- radius: double

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double

**Rectangle**
- width: double
- height: double

+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

---

GeometricObject1.java
Circle4.java
Rectangle1.java
Abstract Classes and Abstract Methods

GeometricObject

Abstract class

- color: String
- filled: boolean
- dateCreated: java.util.Date

#GeometricObject()
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ getDateCreated(): java.util.Date
+ toString(): String
+ getArea(): double
+ getPerimeter(): double

Circle

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

- radius: double

Rectangle

+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

- width: double
- height: double

The # sign indicates protected modifier.

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Overridden methods are generally omitted in the UML diagram for subclasses.
abstract class Fruit {
    // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}

class Orange extends Fruit {
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}
What if we want to inherit from multiple classes?

Example: Pluot (hybrid of plum and apricot)

class Plum extends Fruit{
    public void HowToEat(){ System.out.println(“Plum: Wash”);}
}
class Apricot extends Fruit{
    public void HowToEat(){ System.out.println(“Apricot: Pit it”);}
}
class Pluot extends Plum, extends Apricot{
    //class code
}
public static void main (String [] args){
    Pluot p = new Pluot();
    p.HowToEat(); //which HowToEat method will be called?
}
Interfaces

- What is an interface?
- Why is an interface useful?
- How do you define an interface?
- How do you use an interface?
What is an interface?
Why is an interface useful?

• An interface is a **class-like** construct that contains only constants and abstract methods.

• Interface is to specify **behavior** for objects
  
  – For example, we can specify that the objects are comparable, edible, cloneable using appropriate interfaces such as Comparable, Edible, and Cloneable.

• A class that implements an interface **must implement all the abstract methods**
  
  – For example, we can define Orange and Chicken classes that implement Edible interface.
Interface is a Special Class

• Like an abstract class, you cannot create an instance from an interface using the new operator

• You can create an instance from a class that implements an interface

• You can use an interface as a data type for a variable, as the result of casting, and so on.
Define an Interface

```java
public interface InterfaceName {
    constant declarations;
    method signatures;
}
```

```java
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}

abstract class Fruit implements Edible{
    // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}

class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
Omitting Modifiers in Interfaces

– All data fields are public final static (constants) in an interface

– All methods are public abstract in an interface

```java
public interface T1 {
    public static final int K = 1;
    public abstract void p();
}
```

```
public interface T1 {
    int K = 1;
    void p();
}
```
The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable {
    public int compareTo(Object o);
}
```
String and Date Classes

• Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects

```java
public class String extends Object implements Comparable {
    // class body omitted
}
```

```java
public class Date extends Object implements Comparable {
    // class body omitted
}
```
Declaring Classes to Implement Comparable

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

- Card class implements Comparable!
The Cloneable Interfaces

- Marker Interface: An empty interface.

- A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the `Cloneable` interface is marked cloneable, and its objects can be cloned using the `clone()` method defined in the `Object` class.

```java
package java.lang;
public interface Cloneable {
}
```
Implementing Cloneable Interface

• Declare a custom class that implements the Cloneable interface
Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
Caution: conflicting interfaces

– A class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type)
– This type of errors will be detected by the compiler.
Review questions

• Which of the following is a correct interface?

  A. interface A { void print() { }; }
  B. abstract interface A { print(); }
  C. abstract interface A { abstract void print() { }; }
  D. interface A { void print(); }
Using Interfaces for Event Programming

• Event programming
  – the flow of the program is determined by user actions (mouse clicks, key presses) or messages from other programs.

• Components
  – Event sources: user interface components or other sources that generate the events
  – Events: user actions or other events
  – Event listener: reactions on events

• Basic steps
  – Define event listener - implements an interface called ActionListener which contains a method actionPerformed() for processing the event
  – Register event listener with event sources
The Role of Event Listeners

• One way to visualize the role of a listener is to imagine that you have access to one of Fred and George Weasley’s “Extendable Ears” from the Harry Potter series.

• Suppose that you wanted to use these magical listeners to detect events in the canvas shown at the bottom of the slide. All you need to do is send those ears into the room where, being magical, they can keep you informed on anything that goes on there, making it possible for you to respond.
Event Types

• Java events come in many different types. The event types used in this book include the following:
  – **Mouse events**, which occur when the user moves or clicks the mouse
  – **Keyboard events**, which occur when the user types on the keyboard
  – **Action events**, which occur in response to user-interface actions

• Each event type is associated with a set of methods that specify how listeners should respond. These methods are defined in a **listener interface** for each event type.

• As an example, one of the methods in the mouse listener interface is **mouseClicked**. As you would expect, Java calls that method when you click the mouse.

• Listener methods like **mouseClicked** take a parameter that contains more information about the event. In the case of **mouseClicked**, the argument is a **MouseEvent** indicating the location at which the click occurred.
Taste of Event-Driven Programming

- The example displays two buttons in the frame. A message is displayed on the console when a button is clicked.
- It’s all about the interfaces!

- HandleEvent.java
Events

• An *event* can be defined as a type of signal to the program that something has happened.

• The event is generated by external user actions such as mouse movements, mouse clicks, and keystrokes, or by the operating system or program activities, such as a timer.
The Delegation Model

(a) A generic source component with a generic listener

User Action

source: SourceClass
+addXListener(listener: XListener)

XListener
+handler(event: XEvent)

listener: ListenerClass

Register by invoking source.addXListener(listener);

(b) A JButton source component with an ActionListener

source: JButton
+addActionListener(listener: ActionListener)

ActionListener
+actionPerformed(event: ActionEvent)

listener: CustomListenerClass

Register by invoking source.addActionListener(listener);
Internal Function of a Source Component

(a) Internal function of a generic source object

source: SourceClass

+addXListener(XListener listener)

An event is triggered

event: XEvent

Invoke
listener1.handler(event)
listener2.handler(event)
... 
listenern.handler(event)

Keep it a list

(b) Internal function of a JButton object

source: JButton

+addActionListener(ActionListener listener)

An event is triggered

event: ActionEvent

Invoke
listener1.actionPerformed(event)
listener2.actionPerformed(event)
... 
listenern.actionPerformed(event)

Keep it a list
Chapter 18 Exception Handling
Motivation

When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully? This is the subject we will introduce in this chapter.
Exceptions

- exception = unusual situation
- here program cannot execute 'normally'
- with a more-or-less external cause
- not a program bug
- eg file not found
- connection with server lost
- out of memory
- broken Java machine
Exceptions

- Some methods are declared to 'throw' an exception
- Compiler will insist code 'catches' checked exception
- Code form -
  - try {
  - .. call of method that may throw ExceptionType
  - }
  - catch ( ExceptionType e)
  - {
  - .. code to deal with exception
  - }
Example try catch

FileReader fr = null;
try {
    fr = new FileReader("test.dat");
} catch (FileNotFoundException fe) {
    System.out.println("Can't find file");
}

FileReader is in package java.io
Note declare fr outside try and initialise
Else compiler complains fr may not be initialised
Exception-Handling Example

- Quotient
- QuotientWithIf
- QuotientWithException
Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.
Exception Types

- Object
- Throwable
- Exception
  - ClassNotFoundException
  - IOException
  - AWTException
  - RuntimeException
  - ArithmeticException
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException
  - LinkageError
  - VirtualMachineError
  - AWTError
  - Several more classes
  - Error
    - Several more classes
  - Error
    - Several more classes
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.
Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.
Runtime Exceptions

Exception

ClassNotFoundException

IOException

AWTException

Several more classes

RuntimeException

LinkageError

VirtualMachineError

AWTError

Several more classes

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

LinkageError

VirtualMachineError

Several more classes

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Several more classes

RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.
Checked Exceptions vs. Unchecked Exceptions

• `RuntimeException`, `Error` and their subclasses are known as *unchecked exceptions*.

• All other exceptions are known as *checked exceptions*:

  → `compiler` forces the programmer to check and deal with the exceptions.
Unchecked Exceptions

- unchecked exceptions reflect programming logic errors that are not recoverable.
  - `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it;
  - `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.
- Unchecked exceptions can occur anywhere in the program.
- Java does not mandate you to write code to catch unchecked exceptions.
Checked or Unchecked Exceptions

- Checked exceptions:
  - ClassNotFoundException
  - IOException
  - AWTException
  - ArithmeticException
  - NullPointerException
  - RuntimeException

- Unchecked exceptions:
  - Several more classes
  - LinkageError
  - VirtualMachineError
  - Error
  - AWTError
  - Several more classes

Unchecked exception.
Declaring, Throwing, and Catching Exceptions

```java
method1() {
    try {
        invoke method2;
    }
    catch (Exception ex) {
        Process exception;
    }
}
method2() throws Exception {
    if (an error occurs) {
        throw new Exception();
    }
}
```

declar exception

throw exception

catch exception
Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```java
public void myMethod()
    throws IOException

public void myMethod()
    throws IOException, OtherException
```
Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```
/** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
    if (newRadius >= 0)
        radius = newRadius;
    else
        throw new IllegalArgumentException(
            "Radius cannot be negative");
}
Catching Exceptions

```java
try {
    statements; // Statements that may throw exceptions
} catch (Exception1 exVar1) {
    handler for exception1;
} catch (Exception2 exVar2) {
    handler for exception2;
} ...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```
Catching Exceptions

```java
main method {
  ...  
  try {
    ...  
    invoke method1;
    statement1;
  } catch (Exception1 ex1) {
    Process ex1;
  } statement2;
}

method1 {
  ...  
  try {
    ...  
    invoke method2;
    statement3;
  } catch (Exception2 ex2) {
    Process ex2;
  } statement4;
}

method2 {
  ...  
  try {
    ...  
    invoke method3;
    statement5;
  } catch (Exception3 ex3) {
    Process ex3;
  } statement6;
}

An exception is thrown in method3

Call Stack

main method

Call Stack

main method

method1

method2

method3

47
Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method $p_1$ invokes method $p_2$ and $p_2$ may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```java
void p1() {
    try {
        p2();
    } catch (IOException ex) {
        ...
    }
}

void p1() throws IOException {
    p2();
}
```

(a)  (b)
Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the `setRadius` method in the `Circle` class defined in Chapter 6. The new `setRadius` method throws an exception if radius is negative.

  `TestCircleWithException`          `CircleWithException`
The finally Clause

```java
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
```
Trace a Program Execution

```java
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
Next statement;
```

Suppose no exceptions in the statements
Trace a Program Execution

```java
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
Next statement;
```

The final block is always executed
Trace a Program Execution

```java
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
Next statement in the method is executed
```

Next statement;
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;

Suppose an exception of type Exception1 is thrown in statement2
Trace a Program Execution

```java
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}
Next statement;
```

The exception is handled.
Trace a Program Execution

```java
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}
Next statement;
```

The final block is always executed.
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
try {
    statement1;
    statement2;
    statement3;
} catch(Exception1 ex) {
    handling ex;
} catch(Exception2 ex) {
    handling ex;
    throw ex;
} finally {
    finalStatements;
}

Next statement;

statement2 throws an exception of type Exception2.
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}

Next statement;
try {
    statement1;
    statement2;
    statement3;
} catch (Exception1 ex) {
    handling ex;
} catch (Exception2 ex) {
    handling ex;
    throw ex;
} finally {
    finalStatements;
}

Next statement;
try {
    statement1;
    statement2;
    statement3;
} catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}
Next statement;

Rethrow the exception and control is transferred to the caller
Cautions When Using Exceptions

• Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

• Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.
When to Throw Exceptions

- An exception occurs in a method.
- If you want the exception to be processed by its caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw it.
When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code is inefficient:

```java
try {
    System.out.println(refVar.toString());
} catch (NullPointerException ex) {
    System.out.println("refVar is null");
}
```
When to Use Exceptions

is better to be replaced by

```java
if (refVar != null)
    System.out.println(refVar.toString());
else
    System.out.println("refVar is null");
```