Programming and Software Development
COMP90041

Lecture 8

Interfaces & Exception Handling

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 26-04-2021

- Introduction to UML & Packages & javadoc
- Inheritance
- Access
- Polymorphism
- Abstract Classes

Review: Week 7

- Interfaces
- Handling Exceptions

Outline

- Interfaces
- Handling Exceptions

Outline

Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that implements the interface*
- The syntax for defining an interface is similar to that of defining a class
  - Except the word **`interface`** is used in place of **`class`**
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains **method headings** and **constant definitions** only
  - It contains no instance variables nor any complete method definitions

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected, or package access
- When a class implements an interface, it must make all the methods in the interface public
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

Interfaces

Display 13.1 The Ordered Interface

```
1  public interface Ordered
2  {
3      public boolean precedes(Object other);

4      /**
5       For objects of the class o1 and o2,
6       o1.follows(o2) == o2.preceded(o1).
7      */
8      public boolean follows(Object other);
9  }
```

Do not forget the semicolons at the end of the method headings.

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

The Ordered Interface

- To *implement an interface*, a concrete class must do two things:
  1. It must include the phrase
     implements ***`Interface_Name`***
     at the start of the class definition
     - If more than one interface is implemented, each is listed, separated by commas
  2. The class must implement **all** the method headings listed in the definition(s) of the interface(s)
- Note the use of **`Object`** as the parameter type in the following examples

Interfaces

Display 13.2 Implementation of an Interface

```
1  public class OrderedHourlyEmployee
2           extends HourlyEmployee implements Ordered
3  {
4      public boolean precedes(Object other)
5      {
6          if (other == null)
7              return false;
8          else if (!(other instanceof OrderedHourlyEmployee))
9              return false;
10         else
11         {
12             OrderedHourlyEmployee otherOrderedHourlyEmployee =
13                                 (OrderedHourlyEmployee)other;
14             return (getPay() < otherOrderedHourlyEmployee.getPay());
15         }
16     }
```

Although `getClass` *works better than* `instanceof` *for defining* `equals`*,* `instanceof` *works better in this case. However, either will do for the points being made here.*

Implementation of an Interface

```
17      public boolean follows(Object other)
18      {
19          if (other == null)
20              return false;
21          else if (!(other instanceof OrderedHourlyEmployee))
22              return false;
23          else
24          {
25              OrderedHourlyEmployee otherOrderedHourlyEmployee =
26                                  (OrderedHourlyEmployee)other;
27              return (otherOrderedHourlyEmployee.precedes(this));
28          }
29      }
30  }
```

Implementation of an Interface

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

Abstract Classes Implementing Interfaces

THE UNIVERSITY OF MELBOURNE

Display 13.3 An Abstract Class Implementing an Interface

```
1  public abstract class MyAbstractClass implements Ordered
2  {
3      int number;
4      char grade;
5
6      public boolean precedes(Object other)
7      {
8          if (other == null)
9              return false;
10         else if (!(other instanceof HourlyEmployee))
11             return false;
12         else
13         {
14             MyAbstractClass otherOfMyAbstractClass =
15                                         (MyAbstractClass)other;
16             return (this.number < otherOfMyAbstractClass.number);
17         }
18     }

19     public abstract boolean follows(Object other);

20 }
```

Abstract Classes Implementing Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase
    extends ***BaseInterfaceName***
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Derived Interfaces

Display 13.4 **Extending an Interface**

```
1  public interface ShowablyOrdered extends Ordered
2  {
3      /**
4        Outputs an object of the class that precedes the calling object.
5      */
6      public void showOneWhoPrecedes();
7  }
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

Extending an Interface

- When a class implements an interface, the compiler and run-time system check the syntax of the interface and its implementation
  - However, neither checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
  - It then becomes the responsibility of each programmer implementing the interface to follow the semantics
- If the method body does not satisfy the specified semantics, then software written for classes that implement the interface may not work correctly

Pitfall: Interface Semantics Are Not Enforced

- Chapter 6 discussed the Selection Sort algorithm, and examined a method for sorting a partially filled array of type **`double`** into increasing order
- This code could be modified to sort into decreasing order, or to sort integers or strings instead
  - Each of these methods would be essentially the same, but making each modification would be a nuisance
  - The only difference would be the types of values being sorted, and the definition of the ordering
- Using the **`Comparable`** interface could provide a single sorting method that covers all these cases

The Comparable Interface

The Comparable Interface

- The **`Comparable`** interface is in the **`java.lang`** package, and so is automatically available to any program
- It has only the following method heading that must be implemented:
  public int compareTo(Object other);
- It is the programmer's responsibility to follow the semantics of the **`Comparable`** interface when implementing it

- The method **`compareTo`** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter **`other`** is not of the same type as the class being defined, then a **`ClassCastException`** should be thrown

The Comparable Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

The Comparable Interface Semantics

- The following example reworks the **`SelectionSort`** class from Chapter 6
- The new version, **`GeneralizedSelectionSort`**, includes a method that can sort any partially filled array *whose base type implements the* ***`Comparable`*** *interface*
  - It contains appropriate **`indexOfSmallest`** and **`interchange`** methods as well
- Note: Both the **`Double`** and **`String`** classes implement the **`Comparable`** interface
  - Interfaces apply to classes only
  - A primitive type (e.g., **`double`**) cannot implement an interface

Using the Comparable Interface

Display 13.5 **Sorting Method for Array of Comparable** ***(Part 1 of 2)***

```
1  public class GeneralizedSelectionSort
2  {
3      /**
4       Precondition: numberUsed <= a.length;
5                     The first numberUsed indexed variables have values.
6       Action: Sorts a so that a[0, a[1], ... , a[numberUsed - 1] are in
7       increasing order by the compareTo method.
8      */
9      public static void sort(Comparable[] a, int numberUsed)
10     {
11         int index, indexOfNextSmallest;
12         for (index = 0; index < numberUsed - 1; index++)
13         {//Place the correct value in a[index]:
14             indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
15             interchange(index,indexOfNextSmallest, a);
16              //a[0], a[1],..., a[index] are correctly ordered and these are
17              //the smallest of the original array elements. The remaining
18              //positions contain the rest of the original array elements.
19         }
20     }
```

GeneralizedSelectionSort class: sort Method

Display 13.5 **Sorting Method for Array of Comparable** ***(Part 1 of 2)*** (continued)

```
21      /**
22       Returns the index of the smallest value among
23       a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
24      */
25      private static int indexOfSmallest(int startIndex,
26                                         Comparable[] a, int numberUsed)
27      {
28          Comparable min = a[startIndex];
29          int indexOfMin = startIndex;
30          int index;
31          for (index = startIndex + 1; index < numberUsed; index++)
32              if (a[index].compareTo(min) < 0)//if a[index] is less than min
33              {
34                  min = a[index];
35                  indexOfMin = index;
36                  //min is smallest of a[startIndex] through a[index]
37              }
38          return indexOfMin;
39      }
```

GeneralizedSelectionSort class: sort Method

Display 13.5 **Sorting Method for Array of Comparable** ***(Part 2 of 2)***

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, Comparable[] a)
    {
        Comparable temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }

}
```

GeneralizedSelectionSort class: sort Method

Display 13.6 **Sorting Arrays of Comparable** *(Part 1 of 2)*

```
1  /**
2   Demonstrates sorting arrays for classes that
3   implement the Comparable interface.
4  */
5  public class ComparableDemo
6  {
7      public static void main(String[] args)
8      {
9          Double[] d = new Double[10];
10         for (int i = 0; i < d.length; i++)
11             d[i] = new Double(d.length - i);

12         System.out.println("Before sorting:");
13         int i;
14         for (i = 0; i < d.length; i++)
15             System.out.print(d[i].doubleValue() + ", ");
16         System.out.println();

17         GeneralizedSelectionSort.sort(d, d.length);

18         System.out.println("After sorting:");
19         for (i = 0; i < d.length; i++)
20             System.out.print(d[i].doubleValue() + ", ");
21         System.out.println();
```

The classes `Double` and `String` do implement the `Comparable` interface.

Sorting Arrays of Comparable

Display 13.6 **Sorting Arrays of Comparable** ***(Part 2 of 2)***

```
22          String[] a = new String[10];
23          a[0] = "dog";
24          a[1] = "cat";
25          a[2] = "cornish game hen";
26          int numberUsed = 3;

27          System.out.println("Before sorting:");
28          for (i = 0; i < numberUsed; i++)
29              System.out.print(a[i] + ", ");
30          System.out.println();
31
32          GeneralizedSelectionSort.sort(a, numberUsed);
```

Sorting Arrays of Comparable

Display 13.6 **Sorting Arrays of Comparable** ***(Part 2 of 2)*** (continued)

```
33            System.out.println("After sorting:");
34            for (i = 0; i < numberUsed; i++)
35                System.out.print(a[i] + ", ");
36            System.out.println();
37        }
38    }
```

SAMPLE DIALOGUE

```
Before Sorting
10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,
After sorting:
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
Before sorting;
dog, cat, cornish game hen,
After sorting:
cat, cornish game hen, dog,
```

Sorting Arrays of Comparable

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be **public**, **static**, and **final**
  - Because this is understood, Java allows these modifiers to be **omitted**
- Any class that implements the interface has access to these defined constants

Defined Constants in Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Pitfall: Inconsistent Interfaces

- When a class implements two interfaces:
  - One type of inconsistency will occur if the interfaces have **constants** with the same name, but with different values
  - Another type of inconsistency will occur if the interfaces contain **methods** with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is **illegal**

Pitfall: Inconsistent Interfaces

- Interfaces
- Handling Exceptions

Outline

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called ***throwing an exception***
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called ***handling the exception***

Introduction to Exception Handling

- The basic way of handling exceptions in Java consists of the ***try-throw-catch*** trio
- The **try** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **try** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
  CodeThatMayThrowAnException
}
```

try-throw-catch Mechanism

```
. . . // method code
try
{
  . . .
  throw new Exception(StringArgument);
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
} . . .
```

Eg. ExceptionDemo.java

The try-throw-catch Trio

```
throw new ExceptionClassName(PossiblySomeArguments);
```

- If exception is thrown, **`try`** block stops
  - Normally, the flow of control is transferred to another portion of code known as the **`catch`** block
- The value thrown is the argument to the **`throw`** operator, and is always an object of some exception class

throw

- A **`throw`** statement is similar to a method call:
  `throw new ExceptionClassName(SomeString);`
  - In the above example, the object of class ***`ExceptionClassName`*** is created using a string as its argument
  - This object, which is an argument to the **`throw`** operator, is the exception object thrown
- Instead of calling a method, a **`throw`** statement calls a **`catch`** block

throw

- When an exception is thrown, the **`catch`** block begins execution
  - The **`catch`** block has only **one** parameter
  - The exception object thrown is plugged in for the **`catch`** block parameter
- The execution of the **`catch`** block is called *catching the exception*, or *handling the exception;* the catch block is an exception handler
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **`catch`** block

catch

```
catch(Exception e)
{
  ExceptionHandlingCode
}
```

- A **`catch`** block looks like a method definition that has a parameter of type **`Exception`** class
  - It is not really a method definition

catch

`catch(Exception e) { ... }`

- ***`e` is called the catch block parameter***
- The **`catch`** block parameter does two things:
  - It specifies the type of thrown exception object that the **`catch`** block can catch (e.g., an **`Exception`** class object above)
  - It provides a name (for the thrown object that is caught) on which it can operate in the **`catch`** block
    - Note: The identifier **`e`** is often used by convention, but any non-keyword identifier can be used

catch

- When a **`try`** block is executed, two things can happen:
  1. No exception is thrown in the **`try`** block
     - The code in the **`try`** block is executed to the end of the block
     - The **`catch`** block is skipped
     - The execution continues with the code placed after the **`catch`** block

try-throw-catch Mechanism

2. An exception is thrown in the **`try`** block and caught in the **`catch`** block
   - The rest of the code in the **`try`** block is skipped
   - Control is transferred to a following **`catch`** block (in simple cases)
   - The thrown object is plugged in for the **`catch`** block parameter
   - The code in the **`catch`** block is executed
   - The code that follows that **`catch`** block is executed (if any)

try-throw-catch Mechanism

```
. . . // method code
try
{
  . . .
  throw new Exception(StringArgument);
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
} . . .
```

Using the getMessage Method

- Every exception has a **`String`** instance variable that contains some message
  - This string typically identifies the reason for the exception
- In the previous example, **`StringArgument`** is an argument to the **`Exception`** constructor
- This is the string used for the value of the string instance variable of exception **`e`**
  - Therefore, the method call **`e.getMessage()`** returns this string

Using the getMessage Method

Exception Classes

- There are more exception classes than just the single class **`Exception`**
  - There are more exception classes in the standard Java libraries
  - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
  - There is a constructor that takes a single argument of type **`String`**
  - The class has an accessor method **`getMessage`** that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes must be derived from the class **`Exception`**

- The predefined exception class **`Exception`** is the root class for all exceptions
  - Every exception class is a descendent class of the class **`Exception`**
  - Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a derived class
  - The class **`Exception`** is in the **`java.lang`** package, and so requires no **`import`** statement

Exception Classes from Standard Packages

Exception Classes from Standard Packages

- Numerous predefined exception classes are included in the standard packages that come with Java
  - For example:
    IOException
    NoSuchMethodException
    FileNotFoundException
  - Many exception classes must be imported in order to use them
    import java.io.IOException;

- A **`throw`** statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the **`catch`** block
  - A different type of exception can be defined to identify each different exceptional situation

Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

Defining Exception Classes

Display 9.3 **A Programmer-Defined Exception Class**

```
1  public class DivisionByZeroException extends Exception
2  {
3      public DivisionByZeroException()
4      {
5          super("Division by Zero!");
6      }

7      public DivisionByZeroException(String message)
8      {
9          super(message);
10     }
11 }
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

DivisionByZeroException.java

A Programmer-Defined Exception Class

- An exception class constructor can be defined that takes an argument of another type
  - It would stores its value in an instance variable
  - It would need to define accessor methods for this instance variable

Tip: An Exception Class Can Carry a Message of Any Type: int Message

Display 9.5 **An Exception Class with an `int` Message**

```
1  public class BadNumberException extends Exception
2  {
3      private int badNumber;

4      public BadNumberException(int number)
5      {
6          super("BadNumberException");
7          badNumber = number;
8      }

9      public BadNumberException()
10     {
11         super("BadNumberException");
12     }

13     public BadNumberException(String message)
14     {
15         super(message);
16     }

17     public int getBadNumber()
18     {
19         return badNumber;
20     }
21 }
```

An Exception Class with an int Message

- The two most important things about an exception object are its **type** (i.e., exception class) and the **message** it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method `getMessage`, so that the catch block can use the message

Exception Object Chracteristics

- Must be a derived class of an already existing exception class
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method `getMessage` is inherited

Programmer-Defined Exception Class Guidelines

- For all predefined exception classes, **`getMessage`** returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class, two constructors must be included:
  - A constructor that takes a string argument and begins with a call to **`super`**, which takes the string argument
  - A no-argument constructor that includes a call to **`super`** with a default string as the argument

Preserve getMessage

- A `try` block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a `try` block, at most one exception can be thrown (since a throw statement ends the execution of the `try` block)
  - However, different types of exception values can be thrown on different executions of the `try` block

Multiple catch Blocks

- Each **`catch`** block can only catch values of the exception class type given in the **`catch`** block heading
- Different types of exceptions can be caught by placing more than one **`catch`** block after a **`try`** block
  - Any number of **`catch`** blocks can be included, but they must be placed in the correct order

Multiple catch Blocks

- When catching multiple exceptions, the order of the `catch` blocks is important
  - When an exception is thrown in a `try` block, the `catch` blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

Pitfall: Catch the More Specific Exception First

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

- Because a **`NegativeNumberException`** is a type of **`Exception`**, all **`NegativeNumberExceptions`** will be caught by the first **`catch`** block before ever reaching the second block
  - The catch block for **`NegativeNumberException`** will never be used!
- For the correct ordering, simply reverse the two blocks

ExceptionDemo.java

Pitfall: Catch the More Specific Exception First

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a **`try`** block, and catch the exception in a **`catch`** block that follows
- In this case, the method itself would not include **`try`** and **`catch`** blocks
  - However, it would have to include a ***`throws`*** *clause*

Throwing an Exception in a Method

- If a method can throw an exception but does not catch it, it must provide a warning in the heading
  - This warning is called a ***throws*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

    throws *AnException* //throws clause
  - The following states that an invocation of **aMethod** could throw **AnException**

    public void aMethod() throws AnException
- If a method throws an exception and does not catch it, then the method invocation ends immediately

Declaring Exceptions in a throw Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod()  throws
  AnException, AnotherException
```

Declaring Exceptions in a throw Clause

- Here is an example of a method from which the exception originates:

```
public void someMethod()
                throws SomeException
{
  . . .
  throw new
        SomeException(SomeArgument);
  . . .
}
```

Defining Exceptions in a Method

- When **someMethod** is used by an **otherMethod**, the **otherMethod** must then deal with the exception:

```
public void otherMethod()
{
  try
  {
    someMethod();
    ...
  }
  catch (SomeException e)
  {
    CodeToHandleException
  }
  ...
}
```

ExceptionDemo2.java

Handling Exceptions in another Method

- Two ways of handling exceptions thrown in a method:
  1. The code that can throw an exception is placed within a **`try`** block, and the possible exception is caught in a **`catch`** block within the same method
  2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **`throws`** clause

The Catch or Declar Rule: Two ways

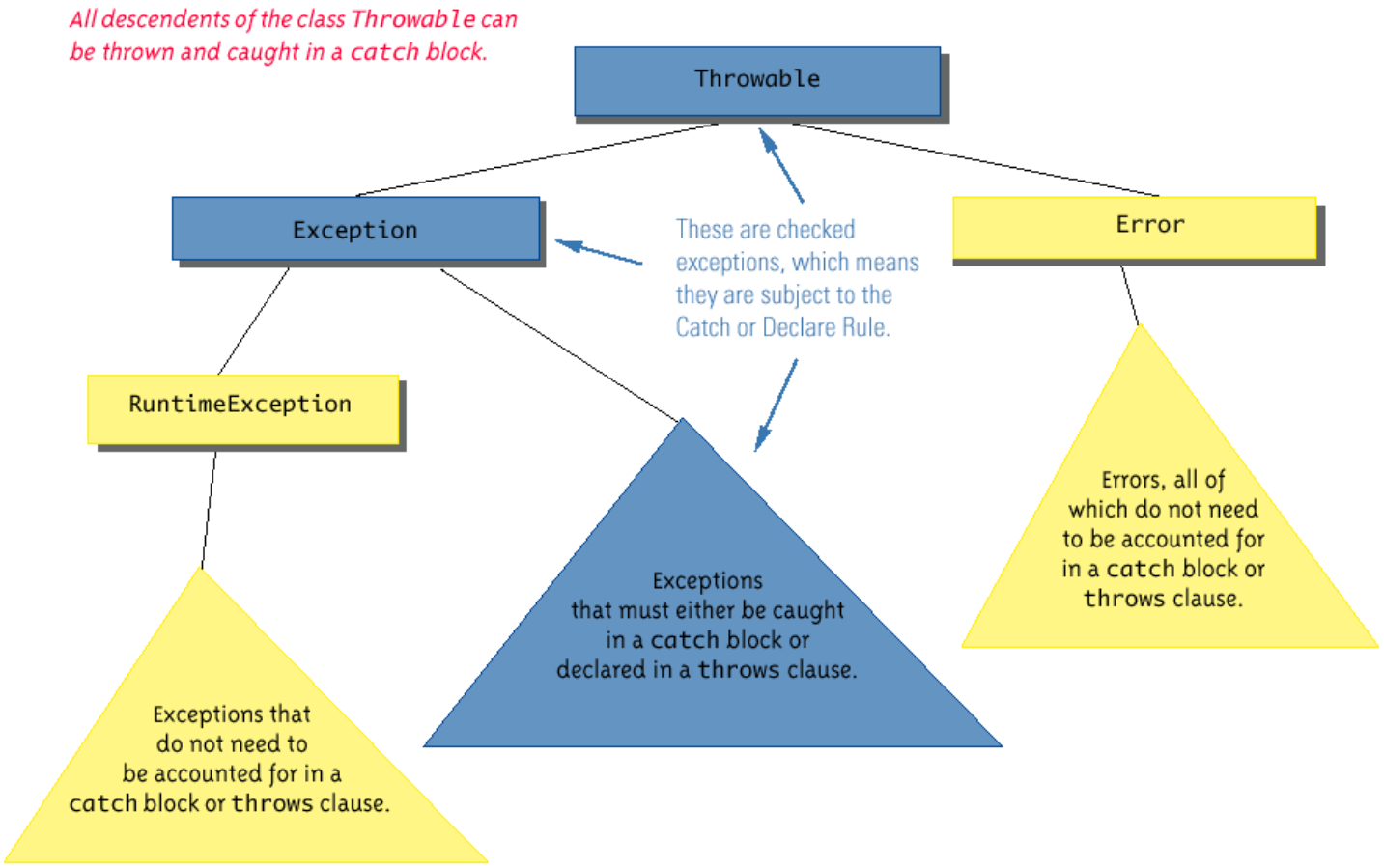
- The first technique handles an exception in a **`catch`** block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a **`catch`** block in some method that does not just declare the exception class in a **`throws`** clause

The Catch and Declare Rule: An Exception must be handled somewhere

- Two techniques can be **mixed**
  - Some exceptions may be caught, and others may be declared in a `throws` clause
- However, these techniques must be used consistently with a given exception
  - If an exception is not declared, then it must be handled within the method
  - If an exception is declared, then the responsibility for handling it is shifted to some other calling method
  - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

The Catch and Declare Rule: Mixed Usage

Display 9.10 **Hierarchy of Throwable Objects**



Hierarchy of Throwable Objects

Checked and Unchecked Exceptions

- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes **`Throwable`**, **`Exception`**, and all descendants of the class **`Exception`** are checked exceptions
- All other exceptions are *unchecked* exceptions -- **must be corrected**.
- The class **`Error`** and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

- When a method in a derived class is overridden, it should have the same exception classes listed in its **`throws`** clause that it had in the base class
  - Or it should have a subset of them
- A derived class may not add any exceptions to the **`throws`** clause
  - But it can delete some

The throws Clause in Derived Clasess

- If every method up to and including the main method simply includes a **`throws`** clause for an exception, that exception may be thrown but never caught
  - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable
  - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a **`catch`** block in some method

What happens if an exception is never caught?

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- How exceptions are handled depends on how a method is called.

When to use Exceptions?

- Exception handling is an example of a programming methodology known as *event-driven programming*
- When using event-driven programming, objects are defined so that they send events to other objects that handle the events
  - An event is an object also
  - Sending an event is called *firing an event*

Event Driven Programming

- In exception handling, the event objects are the exception objects
  - They are fired (thrown) by an object when the object invokes a method that throws the exception
  - An exception event is sent to a `catch` block, where it is handled

Event Driven Programming

- It is possible to place a **`try`** block and its following catch blocks inside a larger **`try`** block, or inside a larger **`catch`** block
  - If a set of **`try-catch`** blocks are placed inside a larger **`catch`** block, **different names** must be used for the **`catch`** block parameters in the inner and outer blocks, just like any other set of nested blocks
  - If a set of **`try-catch`** blocks are placed inside a larger **`try`** block, and an exception is thrown in the **inner `try`** block that is **not caught**, then the exception is thrown to the **outer `try`** block for processing, and may be caught in one of its **`catch`** blocks

Pitfall: Nested try-catch Blocks

The finally Block

- The **`finally`** block contains code to be executed whether or not an exception is thrown in a **`try`** block
  - If it is used, a **`finally`** block is placed after a **`try`** block and its following **`catch`** blocks

```
try
{ ... }
catch(ExceptionClass1 e)
{ ... }
  ...
catch(ExceptionClassN e)
{ ... }
finally
{
  CodeToBeExecutedInAllCases
}
```

- If the **`try-catch-finally`** blocks are inside a method definition, there are three possibilities when the code is run:
  1. The **`try`** block runs to the end, no exception is thrown, and the finally block is executed
  2. An exception is thrown in the **`try`** block, caught in one of the **`catch`** blocks, and the **`finally`** block is executed
  3. An exception is thrown in the **`try`** block, there is no matching **`catch`** block in the method, the **`finally`** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

The finally Block

- When a program contains an assertion check, and the assertion check fails, an object of the class **`AssertionError`** is thrown
  - This causes the program to end with an error message
- The class **`AssertionError`** is derived from the class **`Error`**, and therefore is an unchecked Throwable
  - In order to prevent the program from ending, it could be handled, but this is not required

The AssertionError Class

- The **`nextInt`** method of the **`Scanner`** class can be used to read **`int`** values from the keyboard
- However, if a user enters something other than a well-formed **`int`** value, an **`InputMismatchException`** will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If the exception is caught, the **`catch`** block can give code for some alternative action, such as asking the user to reenter the input

Exception Handling with the Scanner Class

- The **`InputMismatchException`** is in the standard Java package **`java.util`**
  - A program that refers to it must use an **`import`** statement, such as the following:
    import java.util.InputMismatchException;
- It is a descendent class of **`RuntimeException`**
  - Therefore, it is an unchecked exception and does not have to be caught in a **`catch`** block or declared in a **`throws`** clause
  - However, catching it in a **`catch`** block is allowed, and can sometimes be useful

The InputMismatchException

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
  try
  {
    CodeThatMayThrowAnException
    done = true;
  }
  catch (SomeExceptionClass e)
  {
    SomeMoreCode
  }
}
```

Tip: Exception Controlled Loops

Display 9.11 **An Exception Controlled Loop**

```
1  import java.util.Scanner;
2  import java.util.InputMismatchException;

3  public class InputMismatchExceptionDemo
4  {
5      public static void main(String[] args)
6      {
7          Scanner keyboard = new Scanner(System.in);
8          int number = 0; //to keep compiler happy
9          boolean done = false;
```

(continued)

An Exception Controlled Loop (Part 1 of 3)

Display 9.11 An Exception Controlled Loop

```
10          while (! done)
11          {
12              try
13              {
14                  System.out.println("Enter a whole number:");
15                  number = keyboard.nextInt();
16                  done = true;
17              }
18              catch(InputMismatchException e)
19              {
20                  keyboard.nextLine();
21                  System.out.println("Not a correctly written whole number.");
22                  System.out.println("Try again.");
23              }
24          }

25          System.out.println("You entered " + number);
26      }
27  }
```

If `nextInt` *throws an exception, the* `try` *block ends and so the* `boolean` *variable* `done` *is not set to* `true`.

(continued)

An Exception Controlled Loop (Part 2 of 3)

Display 9.11 **An Exception Controlled Loop**

SAMPLE DIALOGUE

```
Enter a whole number:
forty two
Not a correctly written whole number.
Try again.
Enter a whole number:
fortytwo
Not a correctly written whole number.
Try again.
Enter a whole number:
42
You entered 42
```

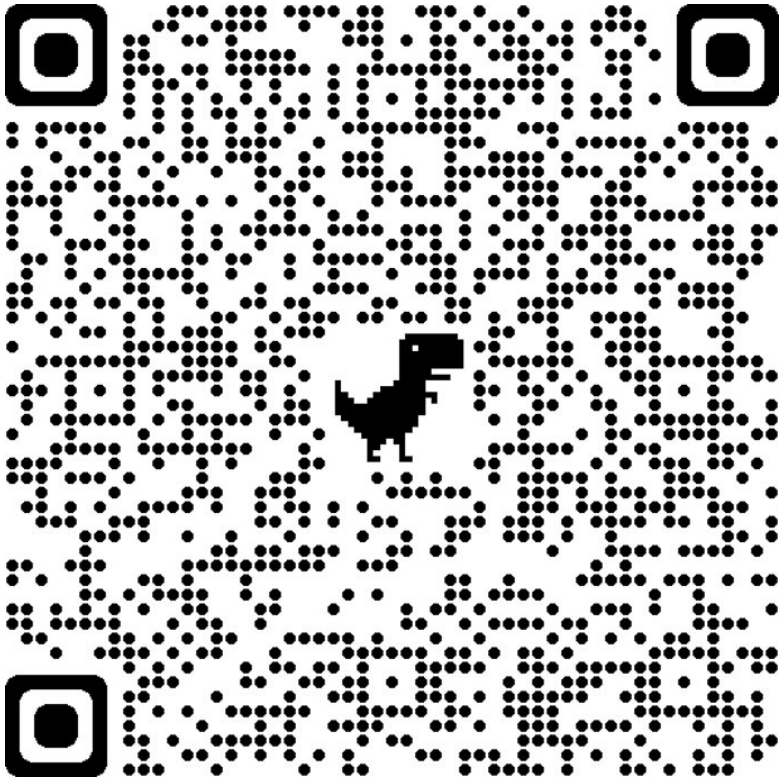
An Exception Controlled Loop (Part 3 of 3)

- An **`ArrayIndexOutOfBoundsException`** is thrown whenever a program attempts to use an array index that is out of bounds
  - This normally causes the program to end
- Like all other descendents of the class **`RuntimeException`**, it is an unchecked exception
  - There is no requirement to handle it
- When this exception is thrown, it is an indication that the program contains an error
  - Instead of attempting to handle the exception, the program should simply be fixed

ArrayIndexOutOfBoundsException

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections



http://go.unimelb.edu.au/5o8i.

Class Reflections