

EXCEPTION HANDLING

CIS*2430 (Fall 2010)

Exceptions

- Normal cases: things go smoothly and nothing unusual happens.
- Exceptional cases: events that cannot be controlled such as file doesn't exist or server goes down.
 - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Exception Handling

- Throwing an exception
 - Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
- Handling/catching the exception:
 - In another place in the program, the programmer must provide code that deals with the exceptional case

try-throw-catch Mechanism

- The **try** block contains the code for the basic algorithm
 - So called a **try** block because it "tries" to execute the case where all goes as planned
- The **try** block contain code that throws an exception if something unusual happens

```
try
{
    ...
    throw new
        ExceptionClassName(PossiblySomeArguments) ;
    ...
}
```

try-throw-catch Mechanism

- When an exception is thrown, the execution of the **try** block is stopped and the control is transferred to a corresponding **catch** block
- The argument to the **throw** operator is always an object of some exception class
- A **throw** statement is similar to a method call, but instead of calling a method, it calls a **catch** block

try-throw-catch Mechanism

- A **catch** block looks like a method definition that has a parameter of type *Exception* class

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

- Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block

Exception Classes

- There are more exception classes that extend the **Exception** class
 - Some exception classes are pre-defined in the standard Java libraries
 - New exception classes can be defined like any other classes
- All pre-defined exception classes have the following properties:
 - There is a constructor that takes a single **String** argument
 - There is an accessor method **getMessage** that can access the string created with the above constructor

Predefined Exception Classes

- Numerous predefined exception classes are included in the standard Java packages:

- For example:

`IOException`

`NoSuchMethodException`

`FileNotFoundException`

- Many exception classes must be imported in order to use them:

```
import java.io.IOException;
```


Predefined Exception Classes

- 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

Using the getMessage Method

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

Defining Exception Classes

- Different exception classes can identify different exceptional situations
- Every exception class to be defined must be a derived class of some already defined exception class
- Constructors are the most important members to define in an exception class

User-Defined Exceptions

Display 9.3 A Programmer-Defined Exception Class

```
1 public class DivisionByZeroException extends Exception
2 {
3     public DivisionByZeroException()           You can do more in an exception
4     {                                           constructor, but this form is common.
5         super("Division by Zero!");
6     }

7     public DivisionByZeroException(String message)
8     {
9         super(message);                       super is an invocation of the constructor for
10    }                                           the base class Exception.
11 }
```

Tip: Message of Any Type

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 }
```

Preserve getMessage

- For all predefined exception classes, **getMessage** returns the string that is passed to its constructor as an argument
- This behavior must be preserved in all programmer-defined exception class
 - A constructor must be included using a string argument for a call to **super**
 - A no-argument constructor must also be included using a default string as its argument for a call to **super**

Multiple catch Blocks

- A **try** block can potentially throw multiple kinds of exceptions, and thus must be handled by multiple **catch** blocks:
 - 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 exceptions can be thrown on different executions of the **try** block

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
 - Catch the more specific exceptions first.

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
 - The program that uses such method should enclose the method invocation in a **try** block, and catch the exception in a **catch** block that follows
- Declaring exception(s):

```
public void aMethod() throws AnException;
```

```
public void anotherMethod() throws  
    AnException, AnotherException;
```

Catch or Declare Rule

- Catch method: place the code that throws an exception in a **try** block and catch the exception when it happens.
- Declare method: declare an exception at the header of a method through a **throws** clause
 - 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

Catch or Declare Rule

- In any one method, both techniques can be mixed
- However, these techniques must be used consistently for 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 that can throw an exception but does not catch it, then the first method must catch or declare it

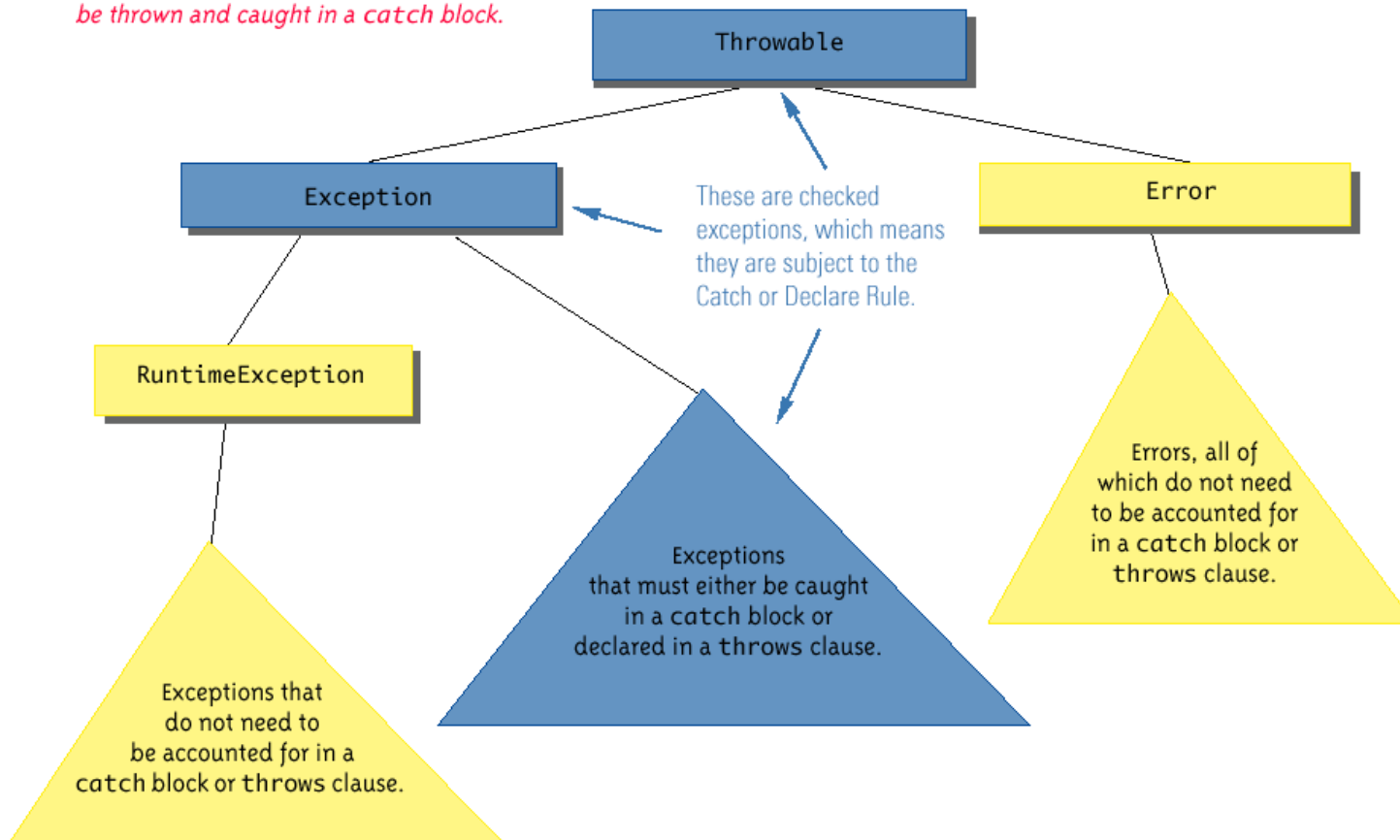
Checked vs Unchecked Exceptions

- 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
 - The class **Error** and all its descendant classes are called *error classes*
 - Error classes are *not* subject to the Catch or Declare Rule

Hierarchy of Throwable Objects

Display 9.10 Hierarchy of Throwable Objects

All descendents of the class Throwable can be thrown and caught in a catch block.



Throws in Derived Classes

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

What Happens if Never Caught

- 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

Event Driven Programming

- 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 also an object
 - Sending an event is called *firing an event*
 - 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; and they are sent to **catch** blocks, where the exceptions are handled

Nested try-catch Blocks

- 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

Re-throw an Exception

- A **catch** block can contain code that throws an exception
 - Sometimes it is useful to catch an exception and then, depending on the string produced by **getMessage** (or perhaps something else), throw the same or a different exception for handling further up the chain of exception handling blocks

The `finally` Block

- The `finally` block contains code to be executed whether or not an exception is thrown in a `try` block

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

The `finally` Block

- If the `try-catch-finally` blocks are inside a method definition, there are three possibilities when the code is run:
 - The `try` block runs to the end, no exception is thrown, and the `finally` block is executed
 - An exception is thrown in the `try` block, caught in one of the `catch` blocks, and the `finally` block is executed
 - 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

AssertionError Class

- 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 exception
 - In order to prevent the program from ending, it could be handled, but this is not required

InputMismatchException

- 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
- 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

Exception Controlled Loop (1/3)

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;
```

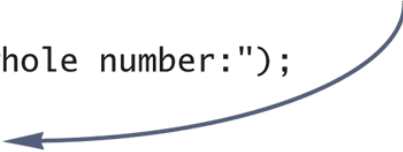
(continued)

Exception Controlled Loop (2/3)

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)

Exception Controlled Loop (3/3)

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

ArrayIndexOutOfBoundsException

- 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