Exception Handling

CIS*2430 (Fall 2021)

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

```
try {
   ...
   throw new ExceptionClass(Arguments);
   ...
} catch(Exception e) {
  // exception handling code

   ...
}
```

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.

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 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:
  - Examples: **Exception, IOException, NoSuchMethodException, FileNotFoundException**
  - **Exception** is the root class for all exceptions, defined in **java.lang** package.
  - Many exception classes should be imported:

    import java.io.IOException;

Using the `getMessage` Method

```
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

```
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

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`.

Tip: Message of Any Type

Display 9.5 An Exception Class with an `int` Message

```
public class BadNumberException extends Exception
{
    private int badNumber;

    public BadNumberException(int number)
    {
        super("BadNumberException");
        badNumber = number;
    }

    public BadNumberException()
    {
        super("BadNumberException");
    }

    public BadNumberException(String message)
    {
        super(message);
    }

    public int getBadNumber()
    {
        return badNumber;
    }
}
```

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 mutiple **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.
- 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.

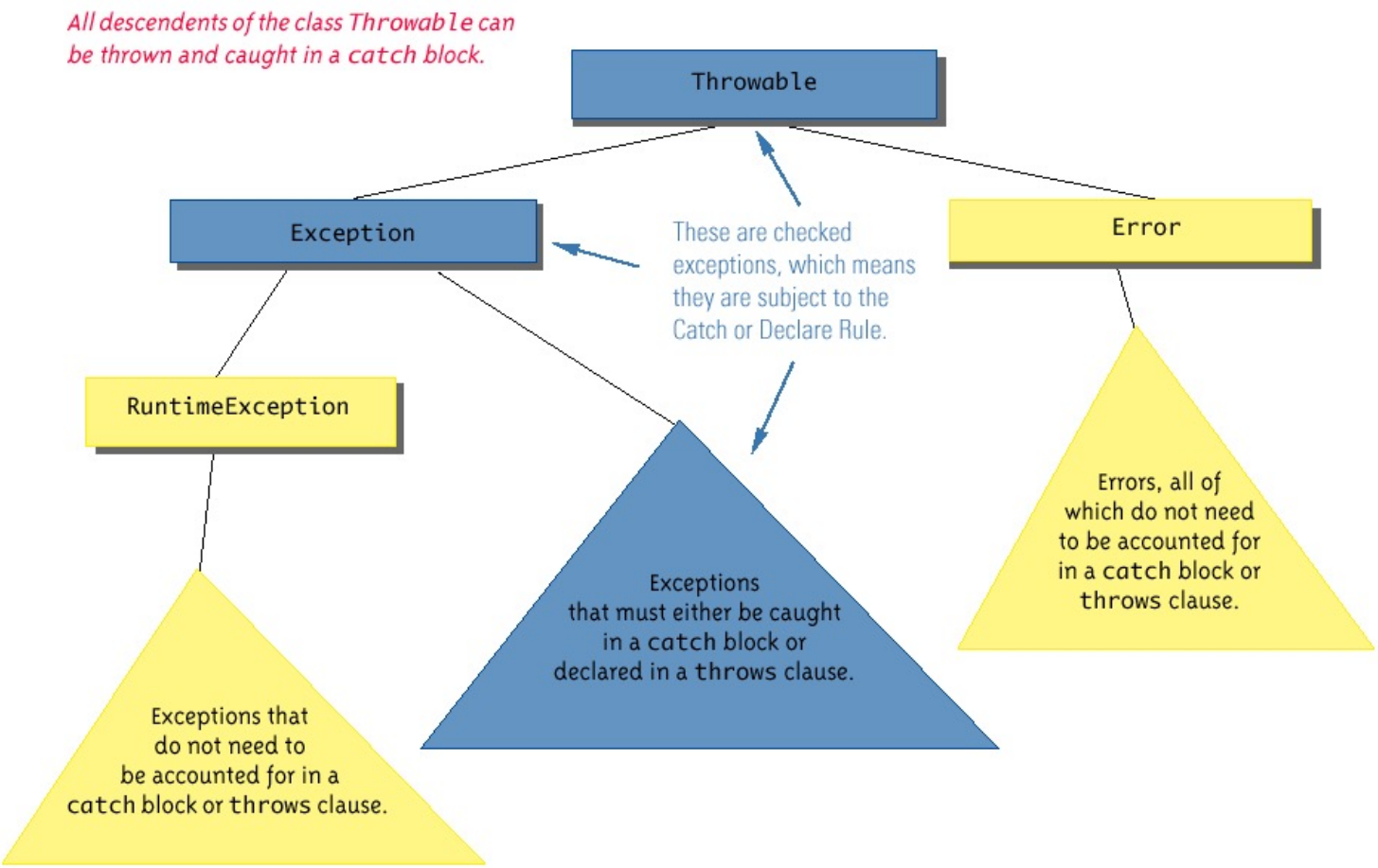
Checked vs Unchecked Exceptions

- 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 Exception Classes

Display 9.10 **Hierarchy of Throwable Objects**



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: nothing happens, but the user may be left in an unexplained situation, making the program no longer reliable
  - In non-GUI programs: it 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.

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(ExceptionClass2 e) {
   . . .
} catch(ExceptionClassN e) {
   . . .
} finally {
  // code to be executed in all cases
}
```

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.

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 that 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

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

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

(continued)

Exception Controlled Loop (2/3)

Display 9.11 An Exception Controlled Loop

```
        while (! done)
        {
            try
            {
                System.out.println("Enter a whole number:");
                number = keyboard.nextInt();
                done = true;
            }
            catch(InputMismatchException e)
            {
                keyboard.nextLine();
                System.out.println("Not a correctly written whole number.");
                System.out.println("Try again.");
            }
        }

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

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`

- **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 other classes of **RuntimeException**, it is an unchecked exception (not required for handling).
- 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.