Exception Handling

Object-oriented programming
Outline

- What is exception handling
- Throwing and catching exceptions
- Rethrowing Exceptions
- Declaring new exception types
- Exceptions and polymorphism

Readings:
- *Java how to program*, chapter 13
Errors & Exceptions

- It's hard to be sure that a piece of code is error-free
  - Programming/designing errors
  - Data errors, abnormal system state
- Exception – an indication of a problem that occurs during a program’s execution
  - `ArrayIndexOutOfBoundsException` – an attempt is made to access an element past the end of an array
  - `NullPointerException` – a null reference is used where an object is expected
- Example?
import java.util.*;
public class TestException {
    public static void main (String args[]) {
        Scanner scanner = new Scanner(System.in);

        System.out.print( "Numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Denominator: " );
        int denominator = scanner.nextInt();

        int result = numerator/denominator; // what if denominator=0?

        System.out.printf("\nResult: %d / %d = %d\n",
            numerator, denominator, result );
    }
}
Exception - Concepts

- Exception: an object containing information about an error, which will be passed on to the code that handles it
- Thrown exception – an exception that has occurred
  - `ArithmeticException` – can arise from a number of different problems in arithmetic
  - `InputMismatchException` – occurs when Scanner method `nextInt` receives a string that does not represent an int value
- Throw point – the initial point at which the exception occurs, top row of call chain
- How do exceptions get thrown?
Example: Throw an Exception

class Fraction {
    private int numerator, denominator;

    public Fraction (int n, int d) throws ArithmeticException
    {
        if (d==0)
            throw new ArithmeticException();
        numerator = n; denominator = d;
    }
}

public class TestException2 {
    public static void main(String [] args) {
        Fraction f = new Fraction (2,0);
    }
}

Declaring what type of exceptions the method might throw

Throw point. An ArithmeticException object is created and thrown.

Exception in thread "main" java.lang.ArithmeticException
    at Fraction.<init>(TestException2.java:4)
    at TestException2.main(TestException2.java:11)
Java Exception Hierarchy

- All exceptions inherit either directly or indirectly from class `Exception`
- `Exception` classes form an inheritance hierarchy that can be extended
- Class `Throwable`, superclass of `Exception`
  - Only `Throwable` objects can be used with the exception handling mechanism
  - Has two subclasses: `Exception` and `Error`
    - Class `Exception` and its subclasses represent exception situations that can occur in a Java program and that can be caught by the application
    - Class `Error` and its subclasses represent abnormal situations that could happen in the JVM – it is usually not possible for a program to recover from Errors
Traditional error handling

```java
System.out.print( "Numerator: " );
int numerator = scanner.nextInt();
System.out.print( "Denominator: " );
int denominator = scanner.nextInt();

if (denominator == 0) {
    // error handling
} else {
    int result = numerator/denominator;
    System.out.printf("\nResult: %d / %d = %d\n", numerator, denominator, result );
}
```

- Error handling logic is mixed with program logic
- Difficult to read, modify, maintain, debug

What about the possible input errors?
Exception handling

- Exception handling
  - resolves exceptions that may occur so that the program can continue or terminate gracefully
  - enables programmers to create programs that are more robust and fault-tolerant

- How to handle an exception?
  - try and catch blocks
How to handle an exception?

- A method can
  1. catch and handle
  2. pass it on to the method’s caller
  3. catch, handle, then pass it on (rethrow)
public static void main (String args[]) {
    Scanner scanner = new Scanner(System.in);
    try {
        System.out.print( "Numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Denominator: " );
        int denominator = scanner.nextInt();
        int result = numerator/denominator;
        System.out.printf("\nResult: %d / %d = %d\n", numerator, denominator, result);
    } //end try
    catch ( InputMismatchException inputMismatchException )
    {
        System.err.println("Exception: " + inputMismatchException);
        scanner.nextLine(); // discard input
        System.out.println( "You must enter integers.\n" );
    } // end catch
    catch ( ArithmeticException arithmeticException )
    {
        System.err.println("Exception: " + arithmeticException);
        System.out.println("Zero is an invalid denominator" );
    } // end catch
}
Try and catch

- Syntax:

```java
try {
    // throw an exception
}
catch (TypeOfException e) {
    // exception-handling statements
}
```

- Separate the code that describes what you want to do (program logic) from the code that is executed when things go wrong (error handling).
  - **try** block – program logic
    encloses code that might throw an exception and the code that should not execute if an exception occurs
  - **catch** block – error handling
    catches (i.e., receives) and handles an exception,
Catching Exceptions

- A `catch` block can catch:
  - Exception of the declared type
    ```java
catch (IOException ioe) {...} can catch exceptions of type IOException
```
  - Exception of a subclass of the declared type
    ```java
catch (IOException ioe) {...} can also catch exceptions of types FileNotFoundException, EOFException,...
```
- Uncaught exception – an exception that occurs for which there are no matching `catch` blocks
  - Cause the current program thread to terminate
How **try** and **catch** work?

![Diagram showing how try and catch work](image-url)
Finally block

- Optional in a `try` statement
- If present, `finally` block is placed after the last `catch` block
- `finally` block executes whether or not an exception is thrown in the corresponding `try` block or any of its corresponding `catch` blocks
- `finally` block will not execute if the application exits early from a `try` block via method `System.exit`
- `finally` block typically contains resource-release code, such as file closing
How finally works?
Throwing exceptions – Why?

- Often exceptions should not be handled right in the method where it happens
  - Not enough information to handle
  - Not having the right authority

```java
class AccountList {
    void loadFromFile (String fileName) {
        Scanner input = new Scanner (new File(fileName));
        //What if the file doesn’t exist?
        //use default name?
        //request another?
        //silently doing nothing?
        ...
    }
}
class Fraction {
    public Fraction(int num, int denom) {
        if (denom==0) {
            //Can’t even return anything
            //a default value for denom?
            ...
        }
    }
}
```

⇒ Pass it on, let others deal with it
Throwing exceptions – How?

- **throws** clause – specifies the exceptions a method may throw
- Exceptions can be thrown
  - by throw statements in method’s body, or
  - from methods called in method’s body
- Thrown exceptions can be of types listed in **throws** clause or their subclasses

```java
void loadFromFile (String fileName) throws FileNotFoundException {
    Scanner input = new Scanner (new File(fileName));
    //This line not reached if the file is not found
    //read the file.
    ...
}

public class Fraction {
    public Fraction(int num, int denom) throws Exception {
        if (denom==0) throw new Exception();
        //normal processing
    }
    ...
}
```
Rethrowing Exceptions

- Exceptions can be rethrown when a catch block decides that
  - it cannot process the exception, or
  - it can process it only partially.

```java
try {... }catch (Exception e) {
    System.out.println(e.getMessage());
    throw e;
}
```
Java Exception Hierarchy

Two categories of exceptions: checked and unchecked

- Checked exceptions (*to be checked by compiler*)
  - Exceptions that inherit from class `Exception` but NOT from `RuntimeException`
  - Compiler enforces a catch-or-declare requirement:
    - checks each method call and method declaration to determine whether the method throws checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a `throws` clause. If not caught or declared, compiler error occurs.

```java
try {
    Scanner input = new Scanner (new File("test.txt"));
    // read the file
} catch (java.io.FileNotFoundException e) { }
```
Java Exception Hierarchy

Two categories of exceptions: checked and unchecked

- Unchecked exceptions
  - Inherit from class RuntimeException or class Error
  - Compiler does not check code to see if exception is caught or declared
  - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
  - Can typically be prevented by proper coding

```java
... 
int numerator = ...;
int denominator = ...;
...
int result = numerator/denominator;
...
```
Declaring new exception types

- New exception classes can be created to serve specific purposes.
- Must extend (subclass) an existing exception class.
- Typically contains only two constructors:
  - One takes no arguments, passes a default exception messages to the superclass constructor.
  - One that receives a customized exception message as a string and passes it to the superclass constructor.
- Could have empty class definition if the class name contains enough information for programmers.
Examples

code

```java
public class SimpleException extends Exception {
}

public class ZeroDenominatorException extends RuntimeException {
    public ZeroDenominatorException () {
        this("Denominator is zero.");
    }
    public ZeroDenominatorException (String name) {
        super(name + "Division by zero!!");
    }
}
```
Examples

```java
public class ZeroDenominatorException extends Exception {}

public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction(int numerator, int denominator)
            throws ZeroDenominatorException {
        if (denominator == 0)
            throw new ZeroDenominatorException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    ...}
```
public static void main (String args[]) {
    Scanner scanner = new Scanner(System.in);
    boolean continueLoop = true; // if more input is needed
    do {
        try {
            // read two numbers and create a Fraction
            System.out.print( "Numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Denominator: " );
            int denominator = scanner.nextInt();
            Fraction f = new Fraction(numerator, denominator);
            continueLoop = false;
        } catch (InputMismatchException iException) {
            System.err.printf( "Exception: %s
", iException);
            scanner.nextLine(); // discard input
            System.out.println( "You must enter integers. Please try again."
            );
        } catch (ZeroDenominatorException zException) {
            System.out.println("Zero is an invalid denominator" );
        }
    } while (continueLoop);
} // end do...while
} // end main

If we have reached this point, input was valid and denominator was non-zero, so looping can stop.

If line continueLoop = false was never successfully reached, loop continues and user can try again.
Exceptions and method overriden

- When you override a method, you can throw only
  - exceptions that have been specified in the superclass’s version of the method, or
  - exceptions of their subclasses.
- Or, you throw no exceptions

- Why?
- polymorphism principle: code that works with the base class will automatically work with any object derived from the base class
  (*objects of subclass can be treated as objects of superclass*)
class A {
    public void methodA() throws ExceptionA {... }
}

class B extends A {
    public void methodA() throws ExceptionB {... }
}

void blah(A a) {
    try {
        a.methodA();
    } catch( ExceptionA) {... }
}

A aa = new A(); blah(aa);
B b=new B(); blah(b);
class A {
    public void methodA() throws RuntimeException {
    }
}

class B extends A {
    public void methodA() throws ArithmeticException {
    }  // subclass of RuntimeException
}

class C extends A {
    public void methodA() throws Exception {
    }  // not a subclass of RuntimeException
}

class D extends A {
    public void methodA() {  // throws no exceptions
    }
}
class A {
    public void methodA() throws RuntimeException {
    }
}
class B extends A {
    public void methodA() throws ArithmeticException {
        // subclass of RuntimeException
    }
}
class D extends A {
    public void methodA() {
        // throws no exceptions
    }
}
A a = new B();
...
try {
    a.methodA();
}
catch (RuntimeException e) {
    ...
Tracing Exceptions

- Can use `printStackTrace()` to trace back to the point where an exception was issued.
  - Used in debugging

- Stack trace
  - Name of the exception in a descriptive message that indicates the problem
  - Complete method-call stack
Which one gets caught here?
1: public class TestStackTrace {
2:   void methodA() throws Exception {
3:       methodB();
4:       throw new Exception();
5:   }
6:   void methodB() throws Exception {
7:       methodC();
8:       throw new Exception();
9:   }
10:  void methodC() throws Exception {
11:       throw new Exception();
12:  }
13:  public static void main(String[] args) {
14:     TestStackTrace t = new TestStackTrace();
15:     try {
16:       t.methodA();
17:     }
18:     catch(Exception e) {
19:       e.printStackTrace();
20:     }
21:  }
22: }

java.lang.Exception
   at TestStackTrace.methodC(TestStackTrace.java:11)
   at TestStackTrace.methodB(TestStackTrace.java:7)
   at TestStackTrace.methodA(TestStackTrace.java:3)
   at TestStackTrace.main(TestStackTrace.java:16)