Exceptions

Handling exceptional situations

Further work

Example: Stack implementation

Ugly way of handling push if full & pop if empty

These are situations in which we need protection

but, the Stack doesn't know the appropriate action (crash?, ignore?, display error?, where?)

Solution?

Return a value that is checked? (not good)

public **boolean** push (String s) ?

now my calling code is **cluttered**

if (push ("hello")) …

else processError( )

This would nearly double the **size** of my code
Single values don't carry appropriate meaning
Also, if I **forget** to handle it…what happens??
And, still **unclear what to do** in processError ( )
Still unclear how to handle the error properly

Maze Runner

Reads from a file

What if…

file not found?

file access is denied?

file contents are corrupt?

Exceptions Chap 11

Exceptional situations only

bad situations "throw" (aka "raise") an exception to the runtime system

The runtime system searches for someone to handle the exception: going up the call stack
(to whoever called this method)

That code can either
handle the situation as it decides, or
propagate (throw) the exception to who-ever called it (up the call stack)
Handle the exception with a **catch**

Example

```
int  x = 0 ;

int y = 15 / x ;      << ArithmeticException!
```

- Exception in thread "main" java.lang.ArithmeticException: / by zero at …

Exception message
Which exception & where
Stack trace - list all methods from call stack

Example

main ( ) calls foo( )

foo ( ) calls bar ( )

bar has a divide by zero…
Find the appropriate exception handler!

call stack:
bar <- *top*
foo
main

Handling Exceptions(1)

```
try {
    …"try block" : attempt code that might throw exceptions…
} catch (ExceptionType1 e){
    …"catch block": code to 'handle' this exception type…
} catch (ExceptionType2 e){
    …code to handle this type of exception…
} finally {
 …optional code executed after try block…
}
```


Three fundamental exception types

Checked Exceptions

invalid conditions - outside program control; file does not exist; network failure; invalid user input

Required to explicitly catch or explicitly declare as propagated

Unchecked Exceptions

Defects in program; "conditions that, generally speaking, reflect errors in your program's logic" - The Java Programming Language

Could be avoided by better programming

Subclass RuntimeException

Are NOT required to be caught - automatically thrown/propagated as needed

Errors *(also not required to Catch or Specify)*

System malfunctions, unanticipatable (?) and unrecoverable

Checked Exceptions

MUST explicitly handle exception

1) Handle by *propagating* the exception

no try-catch — let someone else handle it

technically, this is *handling* the exception because it is explicit

2) Handle using *try-catch* options

Catch that exception and do something (or nothing)

Catch that exception and throw a different exception

Checked Exceptions

Propagate it - must declare that!

```
public int myMethod ( ) throws SomeException
{   ..don't catch it..
```

Catch it - (no "throws" clause)

```
public int myMethod ( )
{    ..try-catch block for that exception..
```

Catch it by and throw a different exception

```
public int myMethod ( ) throws DiffException
{   ..partially handle it..
```


Unchecked Exceptions

May (optionally) handle exception

1) *Default* : Propagate the exception
no handler — let someone else handle it
and no explicit throw clause

2) Handle using try-catch options

Catch that exception and do something *(or nothing)*

(b) Catch that exception and throw a different exception
(which now might require a throws clause)

Unchecked Exceptions

May (optionally) handle exception

```
public int myMethod ( )
{     ..nothing or handle it..
```

no “throws” clause

optionally handle exception with try-catch

This is what we have been doing so far!

Full example

doc.oracle.com - Java tutorials

```
public class ListOfNumbers {
    private ArrayList<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        FileWriter fw = new FileWriter (“Outfile.txt”);
        PrintWriter out = new PrintWriter(fw);

        for (int i = 0; i < SIZE; i++) {
            // The get(int) throws IndexOutOfBoundsException
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
```

Will not compile!

try

catch

Full example

doc.oracle.com - Java tutorials

```
public void writeList() {
 try {
    FileWriter fw = new FileWriter (“Outfile.txt”);
    PrintWriter out = new PrintWriter(fw);

    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
 }
 catch (IOException e)
 {
    //do something other than this
     System.out.println (“IO Exception”+e.getMessage( ));
 }
 catch (IndexOutOfBoundsException e)
 {
     //do something for index out of bounds
     . . .
 }
```

Surround code that can throw exceptions

Full example

doc.oracle.com - Java tutorials

```
public void writeList() {
 try {
    FileWriter fw = new FileWriter (“Outfile.txt”);
    PrintWriter out = new PrintWriter(fw);

    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
 }
 catch (IOException e)
 {
    //do something other than this
     System.out.println (“IO Exception”+e.getMessage( ));
 }
 catch (IndexOutOfBoundsException e)
 {
     //do something for index out of bounds
     . . .
 }
 finally {
    if (out != null)
       out.close( );     //clean up NO MATTER HOW ‘try’ exits
 }
```


Full example

doc.oracle.com - Java tutorials

```
public class ListOfNumbers {
    private ArrayList<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }
```

If the method doesn’t know how to handle exceptions, throw them!

```
  public void writeList() throws IOException, IndexOutOfBoundsException {
        FileWriter fw = new FileWriter (“Outfile.txt”);
        PrintWriter out = new PrintWriter(fw);

        for (int i = 0; i < SIZE; i++) {
            // The get(int) throws IndexOutOfBoundsException
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
  }
```


Throw your own

if you detect situation that you can’t handle

```
throw someThrowableObject;
```

Like-

if trying to create a 148-sided die…

```
if (n > MAX_SIDES_ALLOWED)
   throw new InValidArgumentException();
```

- or as two lines, create exception and throw it -

```
MyException myEx = new MyException( );
throw myEx;
```

Exception Class Hierarchy

Many subclasses of Exception exist
- RunTimeException
- IndexOutOfBoundsException
- NullPointerException

Can create your own too

```
public class MyOwnException extends Exception
    public MyOwnException (String m)
    {
        super (m);
        . . .
    }
```


JUnit Testing and Exceptions

```
@Test(expected = Exception.class)
```

- JUnit now expects a Exception from this test — Success only if it throws it

```
@Test(expected = NullPointerException.class)
public void constructorNullPointerTest( ) {
    Example example = new Example(null);
}
```

- Newest Java has addition techniques too

Maze Runner

The Stack can now handle push-full; and pop-empty situations gracefully

Test cases can now test for exceptions

The GridRunner class can now handle file not found problems