Exceptions in Java

Object Oriented Programming
Winter 2015-16
• Motivation
• Syntax
• Finally Clause
• Exception throwing
• Checked vs. Unchecked.
• Exception and inheritance.
• Exceptions in Eiffel
Examples of Errors

• System failures (considered exceptions)
  – Resource exhaustion
    • Memory allocation failures
    • No more file handles
  – Permissions
  – Hardware failures
  – Network errors

• User errors (considered normal control flow structures)
  – Bad input
  – User interruption

• Program bugs (considered assertions)
  – Array bounds overflow
  – Division by 0
  – Illegal parameters
How Could a Library Author Deal with an Error? (Other than Exceptions)

• Terminate program
  – Do you really want your library to kill you?
• Print an error message and continue
  – Would you allow a library to clobber your screen?
  – To which window? In which language?
• Return a special “error” value
  – Constructors and destructors have no return value
  – What should array subscript operator return?
  – Every `int` is a legal `int`.
  – What’s an error value of `T` in:

    ```java
    public <T> T genericFunction() {...}
    ```

• Call a user supplied function
  – Cumbersome control structure
  – What should this function do?
What is Exception Handling?

- A mechanism for transferring control and information from a point in the execution of the program, to an exception handler.

- Participants:
  - **Code author**: Detects a problem, but doesn't know what to do with it.
  - **Code user**: Cannot detect the problem (otherwise it wouldn't happen), but may be able to handle it.

- The Game:
  - The author detects the error and tells the user.
  - The user takes appropriate action.
A throw causes the run time stack to be unwound until a handler for an exception of the correct type is found. This handler will then be executed.

• **Catcher:**
  – Sets “receiving device(s)” on the stack.
  – Executes code.

• **Thrower:**
  – Detects an exception
  – Throw an object up the stack.

• **Catcher:**
  – Checks which receiving device caught the thrown object.
  – Executes the associated error handler.

Thus, instead of calling an error function, the thrower invokes the exception handling code which resides at the catcher side.
B. Stroustrup enumerates the following options for an exception handler:

- Fix something and retry.
- Recalculate using another method and return a result.
- Pass the bug up the stack (re-throw the exception).
- Throw another exception.
- Fix up and carry on with next statement.
- Give up and terminate the program.
• Exception-Handling Mechanism – similar to C++
  – `throw e` signals the occurrence of an exception
  – The statement `try/catch` allows the calling method to “catch” the “thrown” exception object

```java
public class SyntaxExample{
    public static void main(String[] args){
        try{
            // Open a file:
            FileReader f = new FileReader(args[0]);
            System.out.println(args[0] + " opened");
            f.close();
        } catch (IOException x){
            System.out.println(x);
        }
    }
}
```

Object Oriented Programming, Winter 2015-2016
The **finally** Clause

- For code that must ALWAYS run
  - Regardless of which catch block executed
  - Even if no catch block executed
  - Even if a `return` or `break` occurs first
  - Executes before transferring control to caller
  - Exception (**finally** doesn’t run): `System.exit()`

- Placed after handlers (if they exist)
  - try-block must either have a handler or a finally-block

- Can be used to clean up the programming environment

```java
void n() {
    try {
        ...
        //open window
        p();
    }
    catch(SomeException e) { ... }
    finally { ... //close window ... }
}
void p() throws SomeException {
    throw new SomeException();
}
```
Throwing Exceptions

- Thrown objects must derive (ultimately) from `Throwable`
  - Provides methods like `printStackTrace()`
- Usually derive from `java.lang.Exception`
- Try can catch any exception using the following code:

```java
try{
    ...
} catch(Exception e) {
    //handle any type of exception
}
```

- Be careful: Java executes the first catch statement capable of handling the exception
  - Like C++

```java
try{
    ...
} catch(Exception e) {
    ...
} catch (ArrayIndexOutOfBoundsException ae) {
    ...
}
```

The second handler is never executed.
Checked vs. Unchecked Exceptions

- **Checked Exceptions**
  - Errors a program should handle (recoverable conditions)
  - Subclasses of `Exception` but not `RuntimeException`
  - Compiler requires “catch or declare”
    - Catch and handle the exception in the method **OR**
    - Declare the method can throw an exception:
      ```java
      void A() throws ExceptionType {
      ...
      }
      ```

- **Unchecked Exceptions**
  - Represent *defects* in the program, or programming errors.
  - Are subclasses of `RuntimeException`
  - Methods are *not* required to declare or handle these
Exceptions and Inheritance

• Methods overridden in subclasses must maintain the parent method’s contract
  – Allows substitution of an instance of the child for an instance of the parent
  – Cannot add exceptions to specification
  – Can omit `throws` of exceptions from parent declaration
  – Can throw subclasses of parent’s exceptions

```java
class Parent{
    public void f() throws IOException { ... }
    public void g() throws IOException { ... }
    public void h() throws IOException { ... }
}
class Child extends Parent{
    public void f() { ... } //OK
    public void g() throws SocketException{ ... } //OK
    public void h() throws IOException, OtherException { ... } //Error!
}
```
Multi-Catch Statement

- A catch clause combining two or more exception types
- The catch parameter is implicitly final

```java
public class ExampleMultiCatch{
    public static void main(String[] args) throws Exception {
        try{
            ...
        } catch (FileSystemException | FileNotFoundException e) {
            //e is final
            //e's type is IOException
        }
    }
}
```

- The type of e is the closest shared base
  - In this case IOException
public void m() throws Exception{
    try {
        System.out.println( "m’s throw" );
        throw new Exception();
    }
    catch( Exception e ){
        System.out.println( "m’s handler" );
        throw e;
    }
    finally {
        System.out.println( "m’s finally" );
    }
}

public void n() {
    try {
        System.out.println( "n calling m" );
        m();
    }
    catch( Exception e ){
        System.out.println( "n’s handler" );
    }
    finally {
        System.out.println( "n’s finally" );
    }
}

What is the output of a call to n()?