Exception Handling

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Error Codes

A method wants to inform its caller about a problem.

- Original method call:

```
m(...);
```

- Method call with error code:

```
final int OKAY = 0;         // result was okay
final int IOERROR = 1;      // input/output error
final int ARITHERROR = 2;   // arithmetic error

int result = m(...);
if (result != OKAY)
{ ... } // error handling
else
{ ... } // normal operation
```

Function result describes whether operation was successful.

Program Structure

```
int result1 = m(...);
if (result1 != OKAY)
{ ... } // error handling
else
{
  int result2 = n(...);
  if (result2 != OKAY)
  { ... } // error handling
  else
  {
    int result3 = o(...);
    if (result3 != OKAY)
    { ... } // error handling
    else
    { ... } // normal operation
  }
}
```

Simplified Program Structure

```
int result1 = m(...);
if (result1 != OKAY)
{
  ...; return;
}
int result2 = n(...);
if (result2 != OKAY)
{
  ...; return;
}
int result3 = o(...);
if (result3 != OKAY)
{
  ...; return;
}
// normal operation
```

Functions with Error Codes

A global variable may hold the error code.

```
public static int error;
public static int m(...) { ... }

public static void main(...)
{
  int r = m(...);
  if (error != OKAY)
  { ... } // error handling
  else
  { ... } // normal operation
  ...
}
```

Setting global variables is not good style.

Functions with Error Codes

A value of the result domain may hold the error code.

```
Object r = m(...);
if (r == null)
{ ... } // error handling
else
{ ... } // normal operation, use r
```

What if the result domain does not have extra values?

Functions with Error Code

The result domain may be extended to include the error code.

```
public class Result
{
  public int value;
  public int error;
}

Result r = m(...);
if (r.error != OKAY)
{ ... } // error handling
else
{ int i = r.value; // normal value
  ...
}
```

The interface of the function has to be changed.

Exceptions

Error handling with error codes may make programs messy.

- Exceptions: mechanism to make error handling simpler
  1. Protected code block:
     - Code block in which errors are handled.

       `try { ... } ...`
  2. Exception handler:
     - Code block that handles an error.

       `... catch(...) { ... }`
  3. Exception:
     - An object that identifies the error.

       `throw new ExceptionClass(...)`

Exception raised in a protected code block is forwarded to a handler.

Throwing an Exception

An exception is an object of type `Exception`.

- Typically an exception identifies a specific error:

```
public class ExceptionClass extends Exception
{
  ...
}
```

  - *`ExceptionClass`* may hold error-specific data.

- An exception may be explicitly thrown:

```
throw new ExceptionClass(...)
```

  - Aborts normal exection.
  - Creates an exception object to identify the error.
  - Forwards exception object to appropriate handler.

Handling an Exception

Exception thrown in a protected code block is forwarded to a handler.

```
try
{
  // protected code block
  ...
}
catch (ExceptionClass e)
{
  // exception handler, takes e as parameter
  ...
}
```

After exception handling, execution continues after the construct.

Handling an Exception

Assume that an exception is thrown.

1. The exception is caught by a handler:
   - The exception is raised in a protected code block and
   - The parameter type of the handler matches the exception type.
   - The handler is chosen and executed.
2. There is not caught by a handler:
   - The exception is not raised in a protected code block or
   - The parameter type of the handler does not match the exception type.
   - The exception is forwarded to the caller of the method.

A method may throw an exception.

Methods that Throw Exceptions

A method must declare all the exceptions it may throw.

```
public type method(...) throws ExceptionClass, ...
{
   ...
}
```

All exceptions thrown in method and not caught by a handler.

Example

An exception for input errors.

```
public class InputException extends Exception
{
  public InputException(String msg)
  {
    super(msg);
  }
}
```

We may just use the object representation of the base class.

Example

A method that raises an exception.

```
public static int readInt() throws InputException
{
  int i = Input.readInt();
  if (Input.hasEnded()) return -1;
  if (!Input.isOkay()) throw new InputException("some error");
  return i;
}
```

Normal return value plus one or more exceptions.

Example

A code block that catches the exception.

```
try
{
  int i = readInt(); if (Input.hasEnded()) return;
  int j = readInt(); if (Input.hasEnded()) return;
  int k = readInt(); if (Input.hasEnded()) return;
  System.out.println(i + " " + j + " " + k);
}
catch(InputException e)
{
  System.out.println("Input error: " + e.getMessage());
}
System.out.println("done.");
```

Normal flow of operation is separated from error handling.

Example

Several control flows are possible.

- No `InputException` is thrown.
  - Protected code block is executed.
  - Execution continues after the try statement.

```
2 3 5
done.
```

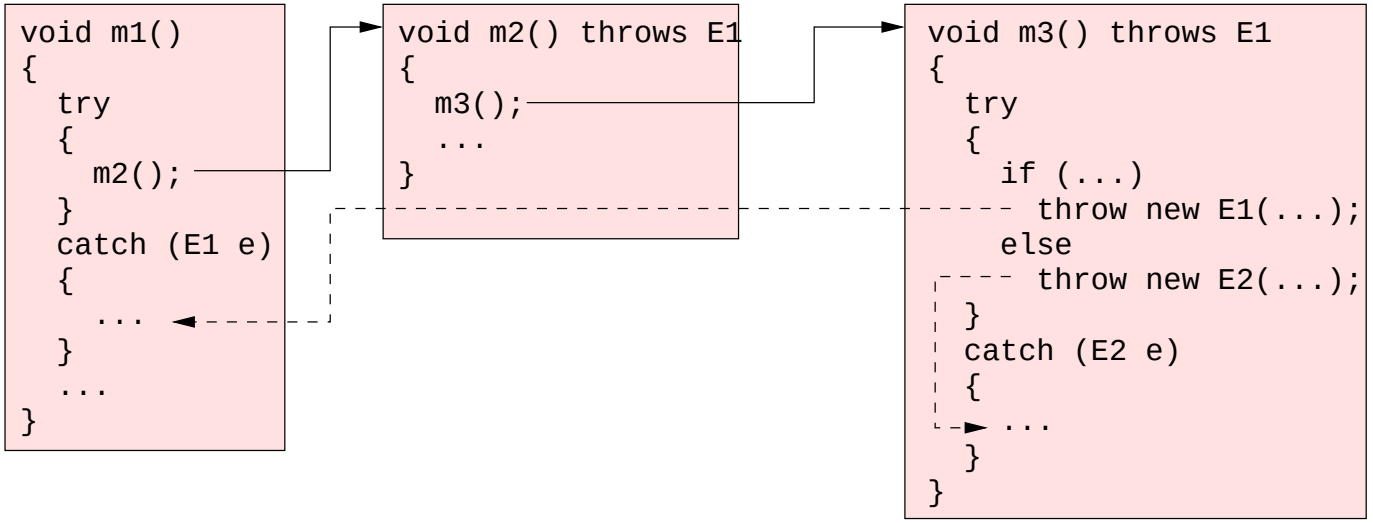
- An `InputException` is thrown in the second call of `readInt`.
  - Exception handler is executed.
  - Execution continues after the try statement.

```
Input error: some error
done.
```

The protected code block is left after an exception has been thrown.

Methods Raising Exceptions

What if an exception thrown is not caught?



Exceptions not caught are thrown "upwards".

Methods Raising Exceptions

A method *m* calls a method *n* that may throw an exception.

1. *m* catches the exception by a corresponding handler.
   - *m* calls *n* within a `try` statement.
2. *m* propagates the exception to its caller.
   - *m* declares the exception type in its method header.

All exceptions thrown by `main` are caught by the JVM.

Catching Multiple Exceptions

A handler for exception type E handles all subtypes of E.

```
try
{
  // may raise multiple kinds of exceptions
  ...
}
catch(Exception e)
{
  // catches all possible kinds of exceptions
  ...
}
```

Generic exception handling is possible.

Multiple Exception Handlers

A `try` statement may have multiple handlers.

```
try
{
  ... // protected code block
}
catch (ExceptionClass1 e1)
{
  ... // exception handler 1
}
catch (ExceptionClass2 e2)
{
  ... // exception handler 1
}
...
```

The handlers are tried in sequence when an exception is thrown.

Default Handler

More general handlers may follow more specific ones.

```
try
{
  // may raise multiple kinds of exceptions
  ...
}
catch(ExceptionClass1 e)
{
  // catches exception type 1
  ...
}
catch(Exception e)
{
  // catches all other kinds of exceptions
  ...
}
```

The `finally` Clause

```
try
{
  ... // protected code block
}
catch (Exception1 e1)
{
  ... // exception handler 1
}
...
finally
{
  ... // executed before try statement is exited
}
```

The `finally` clause is executed at the end of the block.

The `finally` Clause

The `finally` clause is executed in any case.

1. If the protected code block does not throw an exception,
2. If an exception is thrown and caught by one of the handlers,
3. If an exception is thrown and not caught by any of the handlers.

Useful for cleaning up.

Propagating Exceptions

```
public static void m1()
{ try
  { m2(); }
  catch (MyException e)
  { ... } // m1-specific handling of e
}

public static void m2() throws MyException
{ try
  { ... }
  catch (MyException e)
  {
     ... // m2-specific handling of e
     throw e;
  }
}
```

Exception Classes

Java has a hierarchy of exception classes.

```
Exception
  |
  +-- RuntimeException
  |     |
  |     +-- ArithmeticException
  |     +-- IndexOutOfBoundException
  |     +-- NullPointerException
  |     +-- ...
  +-- IOException
  +-- InterruptedException
  +-- ClassNotFoundException
  +-- ...
```

Example

```
try
{
  // let program sleep 100 ms
  Thread.sleep(100);
}
catch(InterruptedException e)
{
  // the sleep was interrupted
  System.out.println(‘‘Someone woke me up.’’);
}
```

Runtime Exceptions

Exceptions derived from `RuntimeException` are special.

1. Runtime exceptions are implicitly thrown by JVM statements:
   - `ArithmeticException`: triggered by a division by zero.
   - `IndexOutOfBoundException`: triggered by an array access with index outside the bounds.
   - `NullPointerException`: triggered by an attempt to dereference a `null` pointer.
2. Runtime exceptions need not be declared in method headers.
   - Since they may be thrown everywhere, this would be too cumbersome.
   - If not caught, they are propagated from `main` to the JVM.

These are the errors typically reported by the `java` interpreter.

Example

```
public class Main
{
  public public static void main(String[] args)
  {
    int i = 0;
    int j = 1/i;
  }
}
```

```
Exception in thread "main"
  java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:6)
```

Catching Runtime Exceptions

Also runtime exceptions may be caught.

```
try
{
  int i = 0;
  int j = 1/i;
}
catch(ArithmeticException e)
{
  System.out.println("Arithmetic Exception: " + e.getMessage());
}
```

Runtime exceptions may be also explicitly thrown with `throw`.

When to Use Exceptions

Use exceptions for situations that are not expected.

```
public static int readAndSum() throws InputException
{
  int s = 0;
  while (true)
  {
    int i = readInt();
    if (Input.hasEnded()) break;
    s += i;
  }
  return s;
}
```

Exceptional situation is separated from normal program flow.

When Not to Use Exceptions

Do not use exceptions for situations that have to be expected.

```
public class EndOfFileException extends Exception { }

public static public int readInt()
  throws EndOfFileException, InputException
{
  int i = Input.readInt();
  if (Input.hasEnded()) throw new EndOfFileException();
  if (!Input.isOkay()) throw new InputException(``some error'');
  return i;
}
```

When Not to Use Exceptions

```
public static int readAndSum() throws InputException
{
  int s = 0;
  try
  {
    while (true)
    {
      int i = readInt();
      s += i;
    }
  }
  catch(EndOfFileException e)
  { return s; }
}
```

Only makes sense, if the exception is eventually thrown.

Example: Reading from a File

```
String name = "in.txt";
try
{
  BufferedReader input = new BufferedReader(new FileReader(name));
  while (true)
  {
    String line = input.readLine();
    if (line == null) break;
    ...
  }
  input.close();
}
catch(FileNotFoundException e)
{
  System.out.println("File " + name + " does not exist.")
}
catch(IOException e)
{
  System.out.println("IO error: " + e.getMessage());
}
```

Example: Writing to a File

```
public public static boolean writeTable(int n, String name)
{
  try
  {
    PrintWriter output =
      new PrintWriter(new BufferedWriter(new FileWriter(name)));
    for (int i=1; i<=n; i++)
    {
      output.println(i);
    }
    output.close();
  }
  catch(IOException e)
  {
     return false;
  }
  return true;
}
```

The Class `kwm.Input`

```
package kwm;
import java.io.*;
import java.util.*;
public class Input
{
  // last read has returned end of stream
  private static boolean endFlag = false;

  // message describing the error
  private static String errorMessage = null;

  // buffered reader wrapped around standard input stream
  private static BufferedReader stdReader =
    new BufferedReader(new InputStreamReader(System.in));

  // current reader
  private static BufferedReader reader = stdReader;
  ...
}
```

Reading a Line

```
private static String readLine()
{
  try
  {
    String line = reader.readLine();
    if (line == null)
      endFlag = true;
    return line;
  }
  catch(Exception e)
  {
    errorMessage = e.getMessage();
  }
  return null;
}
```

Checking for Errors

```
public static boolean isOkay()
{
  return !endFlag && errorMessage == null;
}

public static boolean hasEnded()
{ return endFlag; }

public static String getError()
{ return errorMessage; }

public static void clearError()
{ errorMessage = null; }
```

Reading a Primitive Object

```
public static String readString()
{ return readLine(); }

public static int readInt()
{
  String line = readLine();
  if (line == null) return Integer.MIN_VALUE;
  try
  {
    return Integer.parseInt(line);
  }
  catch(Exception e)
  { errorMessage = e.getMessage(); }
  return Integer.MIN_VALUE;
}
```

Opening a File

```
public static void openInput(String name)
{
  try
  {
    reader = new BufferedReader(new FileReader(name));
  }
  catch(FileNotFoundException e)
  {
    errorMessage = e.getMessage();
  }
}
```

Closing a File

```
public static void closeInput()
{
  if (reader == stdReader)
  {
    errorMessage = "No file has been opened!";
    return;
  }
  try
  {
    reader.close();
  }
  catch (IOException e)
  { errorMessage = e.getMessage(); }
  reader = stdReader;
  endFlag = false;
}
```