Exceptions

James Brucker

Exceptions

Exceptions are unusual events detected by the computer or software.

- An exception is not necessarily an error.

Asynchronous exceptions can occur at any time, independent of program execution.

Example: hardware error, user terminates program

Synchronous exceptions occur in response to some *action by the program*.

Example: subscript out-of-bounds, read error

What Causes Exceptions?

Language Violations (semantic errors)

- illegal array subscript, referencing null pointer.
- illegal value of parameters.

Software Errors related to Environment:

- *try to read/write a file without permission*
- *access a URL that can't be reached (doesn't exist)*

User-defined (programmer-defined) conditions

- your app can "throw" exceptions to signal a problem
- example: "pop" from an empty Stack causes StackUnderFlowException

Hardware Errors - memory error, network error.

- usually fatal and handled by OS.

Examples

```
double [] score = new double[4];
score[4] = 0;
```

ArrayIndexOutOfBoundsException

wrong filename

```
FileInputStream in =
      new FileInputStream("data.tXt");
in.read( );
```

FileNotFoundException

Examples

```
double x = Double.parseDouble("one");
```

What? ______________________________

```
public boolean equals(Object obj) {
   Coin c = (Coin)obj;              //1
   return c.value == this.value;    //2
}
```

1? ClassCastException

2? NullPointerException

Bad URL

```
/** open an internet URL for read */
public InputStream openUrl(String urlstr)
{
   URL url = new URL(urlstr)           //1
   return url.openStream( );           //2
}
```

```
1? openUrl("not a url")
throws MalformedURLException
2? openUrl("http://intel.com/noway")
throws IOException
```

NullPointer: the most common error

```
/** Common error: constructor declares a local
 *  variable instead of initializing an attribute.
 */
public class Purse {
      private Coin[] coins;
      /** constructor for a new Purse */
      public Purse(int capacity) {
            Coin[] coins = new Coin[capacity];
      }
      public int getBalance( ) {
            int sum = 0;
            for(int k=0; k<= coins.length; k++)
              sum += coins[k].getValue();
         return sum;
   }
```

coins is null

How to Handle Exceptions

```
/** open a file and read some data */
public void readFile( String filename ) {
   // this could throw FileNotFoundException
   try {
          FileInputStream in = new
FileInputStream(filename);
      } catch( FileNotFoundException fne )
{
              System.err.println("File not found
"+filename);
              return;
      }
```

- This is called a "try - catch" block.

You can Catch > 1 Exception

```
   String s = scanner.next( ); // read a string
   try {
        int n = Integer.parseInt( s );
        double x = 1/n;
      } catch( NumberFormatException nfe )
{
          System.err.println("Not a number!");
          return;
      } catch( DivisionByZeroException
dze ){
          System.err.println("Fire the
programmer");
      }
```

Scope Problem

- Scope of "x" is the `try { .... }` block.
- Because it is declared inside the block.

```
   try {
     int n = Integer.parseInt( s );

     double x = 1/n;

   } catch( NumberFormatException nfe ) {

         System.err.println("Not a number!");
         return;
   } catch( DivisionByZeroException dze ) {
         System.err.println("Fire
programmer");
   }
   System.out.println("x = "+x);
```

Error: x not defined.

Fixing the Scope Problem

- Define x before the try - catch block.

```
    double x = 0;
    try {
      int n = Integer.parseInt( s );
      x = 1/n;
    } catch( NumberFormatException e ) {
          System.err.println("Not a number!");
          return;
    } catch( DivisionByZeroException e ) {
          System.err.println("Fire the 
programmer");
    }
    System.out.println("x = "+x);
```

IOException, FileNotFoundException

How would you handle these exceptions?

```
/** open a file and read some data */
public char readFile( String filename ) {

   // could throw FileNotFoundException
      FileInputStream in =
                 new
FileInputStream( filename );


      // could throw IOException (read
error)
      int c = in.read( );
```

Syntax of Try - Catch

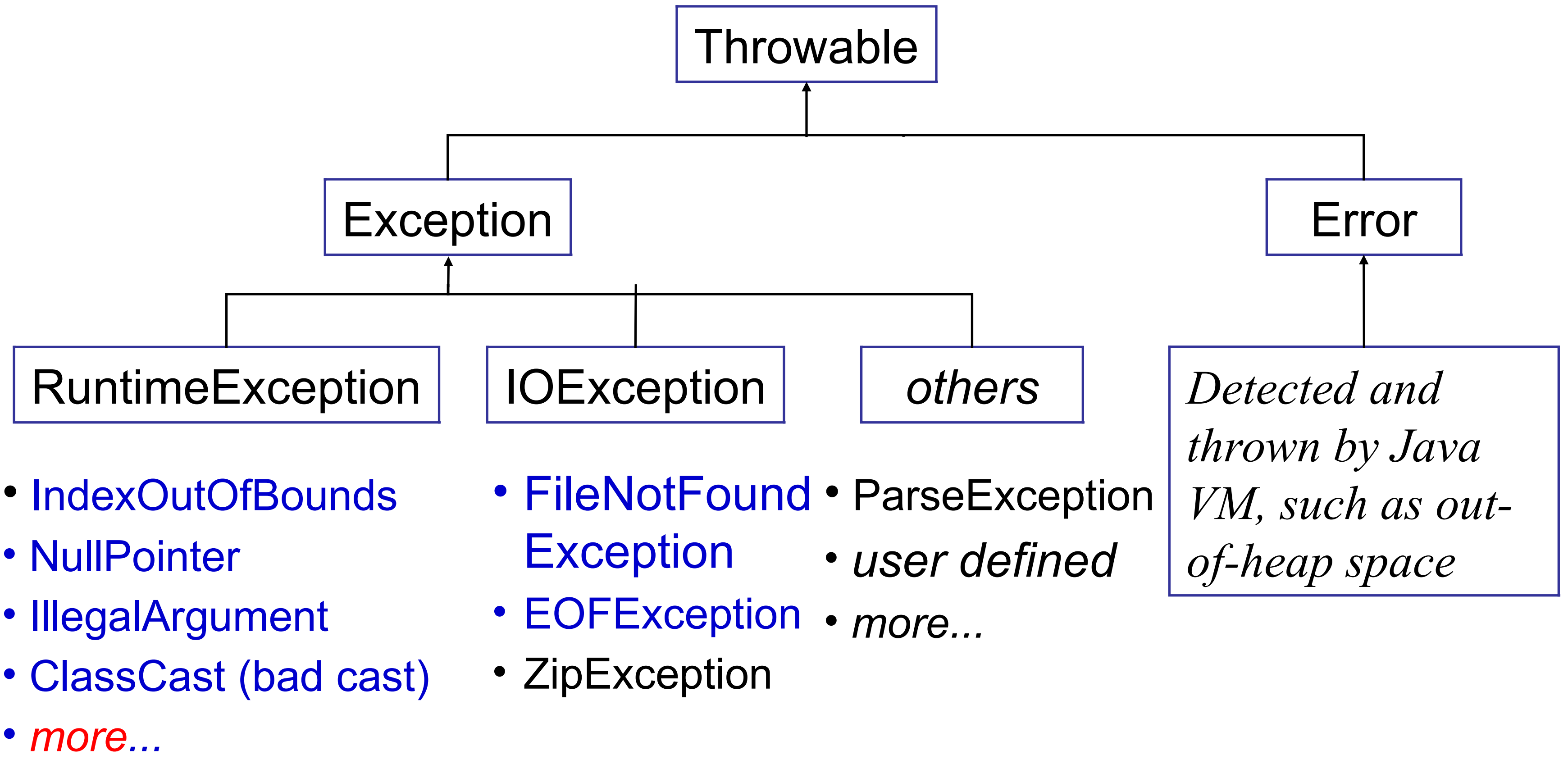
If an exception occurs, control branches to the first matching "catch" clause.

```
try {
      statements;
}
catch( ExceptionType1 e1 ) {
      doSomething;
}
catch( ExceptionType2 e2 ) {
      doSomethingElse;
}
```

Throwable type

Exceptions in Java

Exceptions are subclasses of `Throwable`.



Know the Exceptions

The Java API lists the exceptions each method throws.

class java.util.Scanner

public String **next**()

Finds and returns the next complete token from this scanner. A

...

...

Returns:

the next token

Throws:

NoSuchElementException - if no more tokens are available

IllegalStateException - if this scanner is closed

Know the Exceptions

What exceptions could this code throw?

```
Scanner input = new Scanner( System.in );
int n = input.nextInt( );
```

Catch Matches What?

A "catch" block matches any compatible exception type, including subclasses.

```
Date x = null;
try {
      // What exception is thrown?
      System.out.println( x.toString() );
}
catch( RuntimeException e ) {
      error("Oops");
}
```

Catches what exceptions?

First Match

If an exception occurs, control branches to the first matching "catch" clause.

```
try {
      value = scanner.nextDouble( );
}
catch( InputMismatchException e ) {
      error("Wrong input, stupid");
}
catch( NoSuchElementException e2 ) {
      error("Nothing to read.");
}
```

InputStream Example, Again

```
/** open a file and read some data */
public void readFile( String filename ) {
       FileInputStream in = null;
    // this could throw FileNotFoundException
    try {
              in = new FileInputStream( filename );
              c = in.read();
       }
    catch( FileNotFoundException e ) {
              System.err.println("File not found 
"+filename);
       }
    catch( IOException e ) {
              System.err.println("Error reading file");
       }
```

Exception Order Matters!

```
/** open a file and read some data */
public void readFile( String filename )
      FileInputStream in = null;
   try {
            in = new FileInputStream(
            c = in.read();
      }
   catch( IOException e ) {
            System.err.println("Error reading
      }
   catch( FileNotFoundException e ) {
            System.err.println("File not found
"+filename);
      }
```

FileNotFound Exception is a kind if IOException. First catch gets it.

This catch block is never reached!

Declaring Exceptions

- a method may declare that is may "throw" an exception
- in this case, the method doesn't need "try ... catch".
- the exception is propagated up the call chain

```
/** open a file and read some data.
    @param filename is file to read data from.
        @throws FileNotFoundException if file doesn't
exist
 */
public void readFile( String filename )
             throws FileNotFoundException {
   // this could throw FileNotFoundException
       FileInputStream in = new
FileInputStream( filename );
```

Two Exception Categories

Checked Exceptions

Java *requires* the code to either handle or explicitly declare ("throws") that it may generate this exception.

- "Checked" = you must check for the exception.

Examples:

IOException

MalformedURLException

ParseException

Checked Exceptions

The method must either:

1. use try - catch to handle the exception.

or

2. declare that is "throws" the exception:

```
/**
 * Read data from file.
 * @throws IOException if blah, blah, blah
 */
readFile(String fname) throws IOException {
    InputStream in = new FileInputStream(fname);
    ...
    in.close( );
  }
```

Unchecked Exceptions in Java

Unchecked Exceptions

code is **not** required to handle or declare this type of exception.

Unchecked Exceptions are:

- subclasses of **`RunTimeException`**

IllegalArgumentException

NullPointerException

ArrayIndexOutOfBoundsException

DivideByZeroException (integer divide by 0)

- subclasses of **`Error`**

Why Unchecked Exceptions?

1. Too cumbersome to declare ***every*** possible exception.
2. They can be avoided by correct programming, or
3. Errors that are beyond the control of the application.

```
/** getBalance if we declare all exceptions */
public double getBalance( ) throws
        NullPointerException,
        IndexOutOfBoundsException
{
        double sum = 0;
        for(Valuable v : valuables) sum += v.getValue();
```

Don't write this!

When *should* you catch an exception?

- catch an exception only if you can do something about it
- if the caller can handle the exception better, then "throw" it instead... let the caller handle it.
- declare exceptions as specific as possible

```
/* BAD.  Not specific. */
readFile(String filename) throws Exception {
            ...
}
/* Better. Specific exception. */
readFile(String filename)
            throws FileNotFoundException {
            ...
}
```

You can avoid RunTimeExceptions

"If it is a `RuntimeException`, it's ***your*** fault!"

-- *Core Java, Volume 1*, p. 560.

You can prevent RuntimeExceptions by careful programming.

- **`NullPointerException`** - avoid by testing for a null value before referencing a variable. Or use assertions.
- **`ArrayIndexOutOfBoundsException`** - avoid by correct programming (correct bounds on loops, etc).
- **`ClassCastException`** - indicates faulty program logic
- **`IllegalArgumentException`** - don't pass invalid arguments (duh!).

Avoiding RunTimeExceptions

"If it is a `RuntimeException`, it's ***your*** fault!"

-- *Core Java, Volume 1*, p. 560.

1. Document what your method *requires* and what it *returns.*

2. Know what other code (you use) requires and returns, too.

3. Review and *test* your code.

try - catch - finally syntax

```
try {
      block-of-code;
}
catch (ExceptionType1 e1)
{
      exception-handler-code;
}
catch (ExceptionType2 e2)
{
      exception-handler-code;
}
finally
{
      code to always execute after try-catch
}
```

try - catch - finally example

```
Stringbuffer buf = new StringBuffer();
InputStream in = null;
try {
      in = new FileInputStream( filename );
      while ( ( c = System.in.read() ) != 0 )
            buf.append(c);
}
catch (IOException e){
   System.out.println( e.getMessage() );
}
finally {  // always close the file
   try { if (in != null) in.close(); }
   catch(IOException e) { /* ignored */ }
}
```

Multiple Exceptions

- In C and Java a "try" block can catch multiple exceptions.
- Exception handlers are tried in the order they appear.

```
try {
        System.in.read(buf);
        parseLine(buf);
}
catch (IOException ioe)
    { System.out.println("I/O exception "+ioe); }
catch (Exception ex)
    { System.out.println("Unknown exception "+ex); }
catch (ParseException pe)
    { /* This catch is never reached! */
      System.out.println("Parse exception "+pe);
    }
```

Nested Exception Handlers

You may nest try - catch inside another try - catch.

```
try {
        try {
          out = new FileOutputStream("my file");
        } catch ( FileNotFoundException e ) {
           System.out.println("Error opening file");
           throw e;
        }
        out.write(buf);
}
catch (IOException ioe)
   { System.out.println("I/O exception "+ioe); }
catch (Exception ex)
   { System.out.println("Unknown exception "+ex); }
```

Propagation of Exceptions

Exception are propagated according
to the path of execution of a program.

```
int test1() {
      try {
            answer =
B( );
      }
   catch(Exception e)
      { // handle
exception
   }
}
```

```
int A() throws Exception
{
      throw new

Exception("Help!");
}

int B() throws Exception
{
      ...
      int result = A( );
}
```

Propagation of Exceptions (2)

An exception is propagated to the first dynamically scoped level that can "catch" the exception.

```
int A(Object obj) {
      Integer k = (Integer)obj;//
ClassCastException
      return k.IntValue();
}
/* B() only catches IOException */
int B(Object obj) {
      try {
            result = A(obj);
      } catch (IOException e) { /* do something
*/ }
}
/* C() catches any RuntimeException */
int C() {
      try {
            result = B("10");
```

What if we don't catch the Exception?

- Tthe JVM will catch it.
- The default exception handler:
  - prints name of exception and where it occurred
  - prints stack trace (e.printStackTrace())
  - terminates the program.

```
try {
            dosomething( );
} catch (Exception e ) {
            e.printStackTrace(); // complete "trace" of where exception occurs
}
```

Rethrowing an Exception

- A function can throw an exception it has caught:

```
try {
    sub(); /* sub() that throws exception */
} catch ( RuntimeException e ) {
    System.out.println(
            "Fire the programmer!" );
    // throw it again!
    throw e;
}
```

Exception Handling is Slow

1. Runtime environment must locate first handler.
2. Unwind call chain and stack
   - locate return address of each stack frame and jump to it.
   - invoke "prolog" code for each function
   - branch to the exception handler

Recommendation:
avoid exceptions for *normal* flow of execution.

Exception Handling is Slow

Example: Java code to find a string match in a tree

```
class Node {
        String value;       // value of this node
        Node left = null;   // left child of this
node
        Node right = null; // right child of this
node

        /** find a mode with matching string value
*/
        Node find(String s) {
                int compare = value.compareTo(s);
                if (compare == 0) return this;
                try {
                        if (compare > 0) return
left.find(s);
                        if (compare < 0) return
right.find(s);
```

Avoided Exception Handling

- More efficient to rewrite code to avoid exceptions:

```
class Node {
       String value;
       Node left, right; // branches of this node

       /** find a mode with matching string value
*/
       Node find(String s) {
              int compare = value.compareTo(s);
              if (compare == 0) return this;
              if (compare > 0 && left != null)
                            return left.find(s);
              else if (compare < 0 && right != null)
                            return right.find(s);
              else return null;
       }
```

Multiple catch blocks

```
try {  /* What is wrong with this code? */
          y = func(x);
} catch ( exception ) { cerr << "caught exception";
} catch ( bad_alloc ) { cerr << "caught bad_alloc";
} catch ( ... ) { cerr << "what's this?";
} catch ( logic_error ) { cerr << "Your Error!!";
}
```

```
try {  /* What is wrong with this code? */
     System.in.read(buf); /* throws IOException */
}
catch ( Exception e ) { /* A */
     System.err.println("Exception "+e);
}
catch ( IOException e ) { /* B */
     System.err.println("IO exception "+e);
}
```

Example: lazy equals method

```
public class LazyPerson {
      private String firstName;
      private String lastName;

      /** equals returns true if names are same */
      public boolean equals(Object obj) {
            LazyPerson other = (LazyPerson) obj;
            return
firstname.equals( other.firstName )
                  &&
lastName.equals( other.lastName );
      }
```

What exceptions may be thrown by equals?

Example

```
/**
 * Sum all elements of an array
 */
public int sumArray( int [] arr ) {
      int sum = 0;
      for(int k=0; k<=arr.length; k++)
            sum += arr[k];
      return sum;
}
```

What exceptions may be thrown?

1.

2.

How To Write Code that NEVER crashes?

```
/**
 * Run the Coin Purse Dialog.
 * Don't crash (except for hardware error).
 */
public static void main(String [] args) {
   while(true) try {
        Purse purse = new Purse( 20 ); // capacity 20
        ConsoleDialog dialog =
                              new
ConsoleDialog(purse);
        dialog.run( );
   } catch(Exception e) {
        System.out.println("System will restart...");
     log.logError( e.toString() );
   }
}
```

Exceptions Questions

- Do exception handlers use lexical or dynamic scope?
- What is the purpose of "finally" ?
- Efficiency: see homework problem.

C++ Exception Handling

Exceptions in C++

- An exception can be any type!
- Exceptions can be programmer defined or exceptions from the C++ standard library.

```
struct Error { } e;
try {
       if ( n < 0 ) throw n;
       else if ( n == 0 ) throw "zero";
       else if ( n == 1 ) throw e;
}
catch (int e1)
   { cout << "integer exception raised" << endl; }
catch (string e2)
   { cout << "string exception " << endl; }
catch (Error e3)
   { cout << "struct Error" << endl; }
```

Standard Exceptions in C++

- C++ defines exception classes in <exception>.
- Hierarchy of classes:
  - exception (top level class)
    - runtime_error
    - logic_error
    - others
- Exceptions can be thrown by C++ language features:

  bad_alloc (thrown by "new")

  bad_cast (thrown by "dynamic_cast")

  bad_exception (generic exception)

Exceptions in C++

Class Hierarchy	include file
exception	<exception>
bad_alloc	<new>
bad_cast	<typeinfo>
bad_exception	<exception>
bad_typeid	<typeinfo>
failure <ios>	
logic_error (has subclasses)	<stdexcept>
runtime_error (has subclasses)	<stdexcept>

- bad_exception is a generic type for unchecked exceptions.

Exception Handler in C++

- Example: catch failure of "new".

```
#include <iostream>
using namespace std;
using std::bad_alloc;
char *makeArray(int nsize) {
       char *p;
       try {
              p = new char[nsize];
       } catch ( bad_alloc e ) {
              cout << "Couldn't allocate array: ";
              cout << e.what( ) << endl;
              p = null;
       }
```

C++ Rethrowing an Exception

In C++ *anything* can be "thrown".

```
try {
     sub(); // sub() can throw exception
} catch ( bad_alloc e ) {
     cerr << "Allocation error " << e.what();
     throw;
}
```

Declaring exceptions

- To declare that your function throws an exception:

```
#include <iostream>
using namespace std;
using std::bad_alloc;
char *makeArray(int nsize) throw(bad_alloc) {
       char *p;
       try {
              p = new char[nsize];
       } catch ( bad_alloc e ) {
              cout << "Couldn't allocate array: ";
              cout << e.what( ) << endl;
              throw; // re-throw bad_alloc exception
       }
```

Declaring no exceptions

- To declare that your function throws no exceptions:

```
#include <iostream>
using namespace std;
using std::bad_alloc;
char *makeArray(int nsize) throw() {
       char *p;
       try {
              p = new char[nsize];
       } catch ( bad_alloc e ) {
              cout << "Couldn't allocate array: ";
              cout << e.what( ) << endl;
              return NULL;
       }
```

Exception Handler in C++

- A function can have multiple "catch" blocks.

```
int main( ) {
       // ... other code goes here ...
       try {
              sub(); /* sub() that throws exceptions
*/
       } catch ( bad_alloc e ) {
              cerr << "Allocation error " <<
e.what();
       }
       } catch ( exception e ) {
              cerr << "Exception " << e.what();
       }
       } catch ( ... ) {
              // "..." matches anything:  this catch
              // block catches all other exceptions
              cerr << "Unknown exception " << endl;
```

C++ Default Exception Handler

- If an exception is not caught, C++ provides a default exception handler:
  - If the function didn't use "throw(something)" in its header, then a method named **`terminate()`** is called.
  - If a function declares exceptions in its header, but throws some *other* exception, then the function **`unexpected()`** is called. **`unexpected()`** also calls **`terminate()`**.

C++ Default Exception Handler

- **`unexpected()`** in implemented as a pointer. You can change it to your own exception handler using: **`set_unexpected(`** *your_function* **`)`**
- Similarly, use **`set_terminate()`** to replace **`terminate()`** with some other function.
- Prototypes for set_unexpected() and set_terminate() are defined in the header file <**`exception`**>.

C++ Default Exception Handler

```
#include <exception>
void my_terminator() {
  cerr << "You're terminated!" << endl;
  exit(1);
}
void my_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(1);
}
int main() throw() {
  set_unexpected(my_unexpected); // ignore return value
  set_terminate(my_terminator);
  for(int i = 1; i <=3; i++)
  try { f(i); }
  catch(some_exception e) {
     cout << "main: caught " << e.what() << endl;
       throw;
}
```