Exceptions

James Brucker

Exceptions

Exceptions are unusual events detected by the hardware or software.

- 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: array index out-of-bounds, read error

What Causes Exceptions?

Language Violations

- illegal array subscript, referencing a null pointer.
- integer divide by zero

Environment:

- read a file without "read" permission

User-defined (programmer-defined) conditions

- app can "throw" exceptions to signal a problem
- ex: Iterator next() may throw NoSuchElementException

Hardware Errors - out of memory error, network error.

- usually fatal

Example of Exceptions

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

`NullPointerException`

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

`ArrayIndexOutOfBoundsException`

Example of Exceptions

```
List<String> list =
               Arrays.asList(score);
list.get( list.size() );
```

`IndexOutOfBoundsException`

Not "Array IndexOut..." as on previous slide

wrong filename

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

`FileNotFoundException`

Example of Error condition

```
String [] s = new String[1_000_000_000];
```

```
java.lang.OutOfMemoryError - not enough heap
space for array
```

null reference or not a coin?

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

What exceptions may be thrown?

1? ________________________________

2? ________________________________

Not a number

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

What exception? ______________________

NullPointerException - the #1 programming error

```
/** What statement throws NullPointerException? */
public class Purse {
    private Coin[] coins;
    private int count = 0;
   /** 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;
    }
```

Can this throw NullPointerException?

```
public class Purse {
    private Coin[] coins;
    private int count = 0;
   public Purse(int capacity) {
      coins = new Coin[capacity]; // fixed!
   }
   public int getBalance( ) {
      int sum = 0;
      for(int k=0; k < coins.length; k++)
           sum += coins[k].getValue();
       return sum;
    }
```

Bad URL

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

```
openUrl("not a url")
 1 throws MalformedURLException
openUrl("http://foo.com/doesnotexist")
 2 throws IOException
```

How to Handle Exceptions

1. "catch" the exception and do something.

2. declare that the method "throws exception"

This means that *someone else* will have to handle the exception.

Catching an Exception

This is called a "try - catch" block.

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

You can Catch > 1 Exception

```
scanner = new Scanner(System.in);
try {
   int n = scanner.nextInt();
   double x = 1/n;
} catch( InputMismatchException ex1 ) {
  System.err.println("Input is not an int");
} catch( DivisionByZeroException ex2 ){
  System.err.println("Fire the programmer");
}
```

Multi-catch

```
scanner = new Scanner(System.in);
try {
   int n = scanner.nextInt();
   double x = 1/n;
} catch( InputMismatchException |
         NoSuchElementException |
         DivisionByZeroException ex ){
  System.err.println("Fire the programmer");
}
```

Scope Problem

- `try { ... }` block defines a scope.

```
try {
    int n = scanner.nextInt( );
    double x = 1/n;
} catch( InputMismatchException ex1 ) {
    System.err.println("Not an int");
} catch( DivisionByZeroException ex2 ) {
    System.err.println("Fire the programmer");
}
System.out.println("x = " + x);
```

Error: x not defined here.

Fixing the Scope Problem

- Define x before the try - catch block.

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

"Throw" the Exception

A method or constructor that does not handle exception itself must declare that it "`throws Exception`"

```
/** Read data from an InputStream */
public void readData(InputStream in)
                throws IOException {

  // read the data from InputStream
  // don't have to "catch" IOException
  // but it is OK if you want to.
}
```

Throwing Many Exceptions

A method or constructor can throw many exceptions. Here is a constructor of FileInputStream:

```
/** Create an InputStream for reading from a file.
 * @throws FileNotFoundException
 *    if arg is not a regular file or not readable
 * @throws SecurityException
 *    if security manager denies access to file
 */
public FileInputStream(String filename)
 throws FileNotFoundException, SecurityException {

   // create inputstream to read from filename
}
```

How do you know what exceptions may be thrown?

The Java API tells you.

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

Document Exceptions You Throw!

Write `@throws` tag to document the exceptions your method throws.

Describe the <u>conditions</u> when exception is thrown.

```
public interface Iterator<E> {
  /**
   * Return the next element from the iterator.
   * @throws NoSuchElementException
   *       if the iterator has no more elements
   */
  public E next( );
}
```

Useful: IllegalArgumentException

```
public class Money implements Valuable {
  /**
   * Instantiate Money with a value and currency.
   * @param value of the money, may not be neg.
   * @throws IllegalArgumentException
   *     if value is negative
   */
  public Money(double value, String currency) {
     if (value < 0.0)
       throw new IllegalArgumentException(
          "Value may not be negative");
     this.value = value;
}
```

What if we don't catch the Exception?

- the method returns and the calling method gets a chance to catch exception.
- if caller does not catch it, it returns immediately, and its caller gets a chance.
- If no code catches the exception, the JVM handles it:
  - prints name of exception and where it occurred
  - prints a stack trace (e.printStackTrace())
  - terminates the program.

Propagation of Exceptions

Exception are propagated "up the call chain".

```
int a() throws Exception {
   int result = b( );
}
int b() throws Exception
{
   throw new Exception("Help!");
}
```

```
public static void main(String[] args) {
    try {
      answer = a( );
    }
    catch(Exception e) {
     // handle exception
    }
```

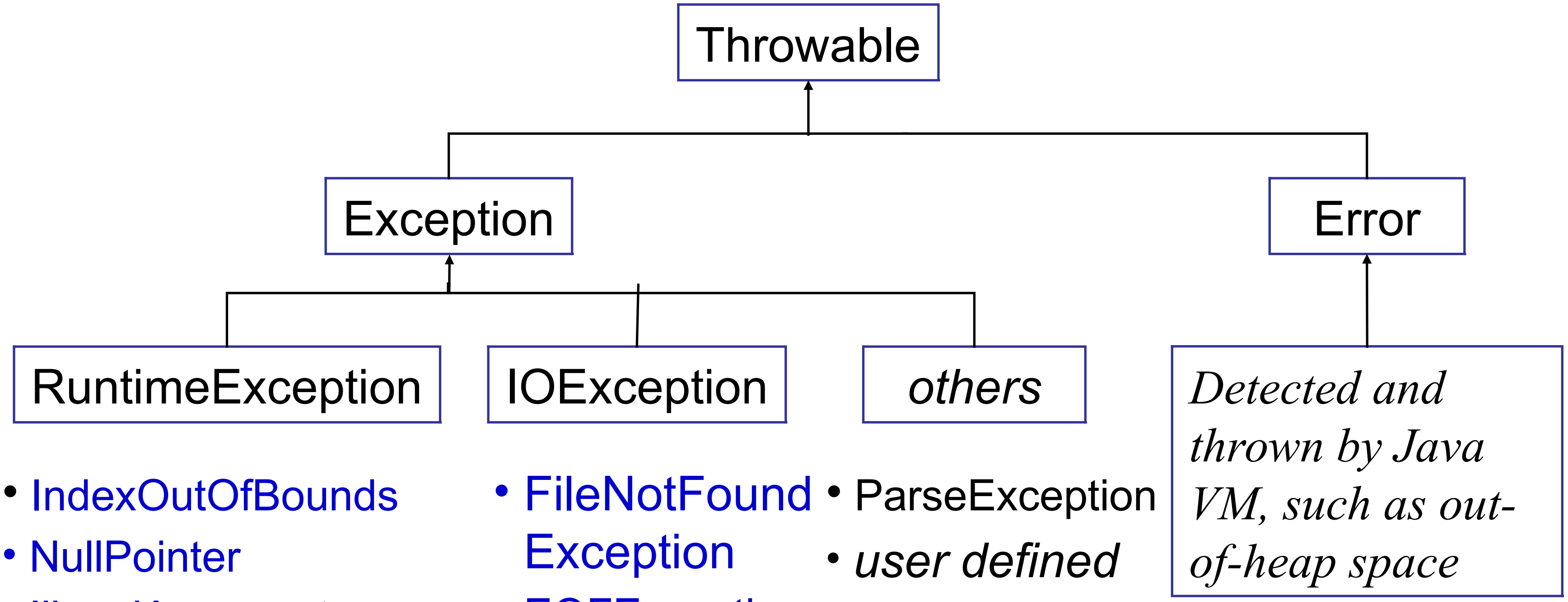
IOException, FileNotFoundException

How would you handle these exceptions?

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

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


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

   return (char)c;
```

Exceptions in Java

Exceptions are subclasses of **`Throwable`**.



- IndexOutOfBounds
- NullPointer
- IllegalArgument
- ClassCast (bad cast)
- *more...*

- FileNotFound Exception
- EOFException
- ZipException

- ParseException
- *user defined*
- *more...*

What exceptions must we handle?

Java does not require us to use try - catch here:

```
Scanner console = new Scanner( System.in );
// We don't have to catch NumberFormatException.
int n = console.nextInt( );
```

But we must use try-catch or "throws" here:

```
// Must handle FileNotFoundException
FileInputStream instream =
              new FileInputStream("mydata.txt");
```

Why?

Give 3 Examples

Name 3 exceptions that you are not required to handle using "try - catch".

(think of code you have written where Eclipse did not require you to write try - catch)

1.

2.

3.

Two Exception Categories

Java has 2 categories of exceptions:

Checked Exceptions

Java *requires* the code to either handle (try-catch) or declare ("throws") that it may cause this exception.

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

Examples:

`IOException`

`MalformedURLException`

`ParseException`

Unchecked Exceptions

Unchecked Exceptions

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

Unchecked Exceptions are:

- subclasses of **`RunTimeException`**

`IllegalArgumentException`

`NullPointerException`

`ArrayIndexOutOfBoundsException`

`DivideByZeroException` (integer divide by 0)

- all subclasses of **`Error`**

Why Unchecked Exceptions?

1. Too cumbersome to declare every possible occurrence

2. They can be avoided by correct programming, or

3. Something beyond the control of the application.

If you were required to declare all exceptions:

```
public double getBalance( ) throws
    NullPointerException, IndexOutOfBoundsException,
    OutOfMemoryError, ArithmeticException, ...
{
    double sum = 0;
    for(Valuable v : valuables) sum += v.getValue();
```

You can avoid RunTimeExceptions

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

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

You can avoid 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

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.

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 {
     ...
}
```

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( filename
     c = in.read();
  }
   catch( IOException e ) {
     System.err.println("Error reading file");
  }
   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 <u>never</u> reached!

try - catch - finally syntax

```
try {
   block-of-code;
}
catch (ExceptionType1 e1)
{
   exception-handler-code;
}
catch (ExceptionType2 e2)
{
   exception-handler-code;
}



{
   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
    if (in != null) try { 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);
    }
```

Rethrowing an Exception

A function can throw an exception it has caught:

```
try {
    sub();    // sub() 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);
      } catch ( NullPointerException e ) {
         return null;
      }
```

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;
}
```

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;
}
```

Multi-catch

In Java 8, you can catch multiple kinds of exception using one `catch( ... )` block.

```
try {
   statements;
}
catch( ExceptionType1 | ExceptionType2 e )
{
   doSomething;
}
```