Exceptions

James Brucker

What are Exceptions?

Exceptions are unusual events detected by the hardware or software.

- not necessarily an error.

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

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

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

Example: hardware error, network error

What Causes Exceptions?

Language Violation

- illegal array subscript, using 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

Examples

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

`NullPointerException`

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

`ArrayIndexOutOfBoundsException`

Examples

```
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

Error Example

```
Double[] d = new Double[1_000_000_000];
```

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

What exceptions are thrown here?

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

What exceptions <u>may</u> be thrown?

1? ________________________________

2? ________________________________

Not a number

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

What exception? ______________________

The #1 programming error

Which statement throws NullPointerException?

```
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;
   }
```

Can this throw NullPointerException?

```
public class Purse {
    private Coin[] coins;

    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;
    }
```

How to Handle Exceptions?

1. "catch" the exception and do something.

2. declare that the method "throws exception"

   - This means that *the calling method* will need to handle the exception.

3. Ignore it.

   - Allowed for Error and RuntimeExceptions

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 {
   InputStream in = new FileInputStream(filename);
} catch( FileNotFoundException ex ) {
   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 (out of scope).

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);
```

"Propagate" an Exception

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

- Required only for Checked Exceptions

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

  // read the data from InputStream
  // don't have to "try - catch" IOException
}
```

Why not catch an exception?

Method does not know how to cope with the problem, so let the caller handle it.

Example: a method to open and read data from a specified file.

Caller should know if the file does not exist.

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

What if we don't catch the Exception?

- the current method returns *immediately*
- the exception is passed (propagated) to caller.
- caller can "catch" exception or the exception propagates again.
- 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
   }
```

Are we *required* to handle exceptions?

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

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

But we are required to try-catch or declare "throws ..." :

```
// 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 that *could* throw exception, but you didn't write try - catch)

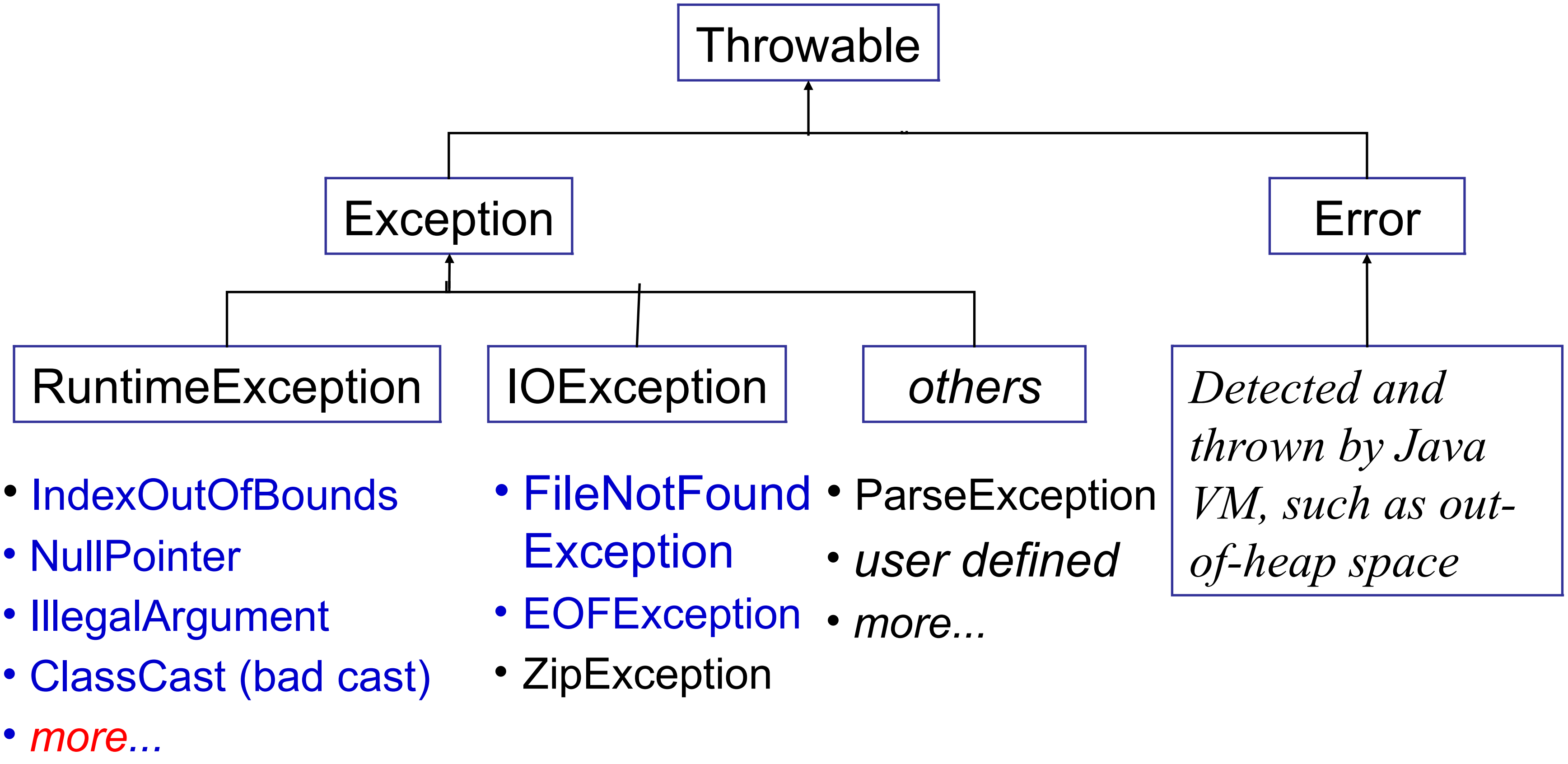
1.

2.

3.

Exceptions in Java

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



Two Exception Categories

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();
```

Exception Reading a File

```
public String readfile(String filename)
{
   InputStream in =
         new FileInputStream(filename);//1
   byte b = in.read();                  //2
```

```
1 may throw FileNotFoundException

2 may throw IOException
```

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. Always initialize variables!
- **`ArrayIndexOutOfBoundsException`** - avoid by correct programming -- correct bounds on loops, etc.
- **`ClassCastException`** - indicates faulty program logic
- **`IllegalArgumentException`** - don't pass invalid arguments. Validate input data before using it.

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 *might* this code throw?

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

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!

```java
/** open a file and read some data */
public void readFile( String filename )
    FileInputStream in = null;
    try {
        in = new FileInputStream( filena
        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 */ }
}
```

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.

Example: lazy equals method

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

   /** equals returns true if names are same */
   public boolean equals(Object obj) {
      Person other = (Person) 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.

Exception Handling in Python

1. Identify common exceptions

2. Use try - except

3. How to throw (raise) an exception in code