CHAPTER 10

Exceptions

Contents

- What s an Exception?
- The Catch or Specify Requirement
- Catching and Handling Exceptions
- Specifying the Exceptions Thrown by a Method
- How to Throw Exceptions
- Unchecked Exceptions – The Controversy
- Advantages of Exceptions

Error Handling in C

- To check success of opening a file in C

```
#include <stdio.h>
main() {
File *fp;
char name[20];
If ( (fp = fopen("input.dat", "w") == NULL) { /* open fail */
   fprintf(stderr, "cannot open the file %s¥n","input.dat");
   exit(0);
 }
/* open success, so do next step */
fscanf(fp,"%s", name);
…..
```

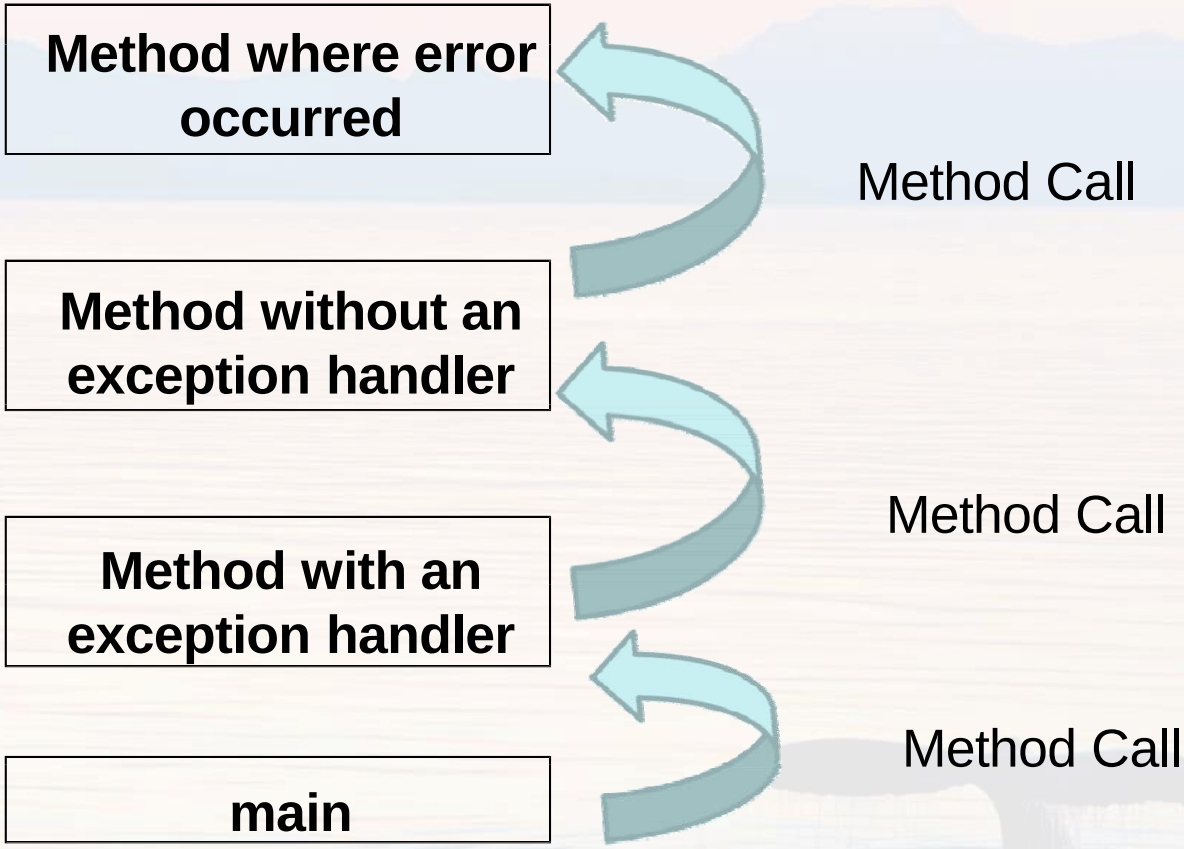
Error Handling Code

It is difficult to separate error-handling code from regular code!
How can we provide some systematic way for the error handling in Java?

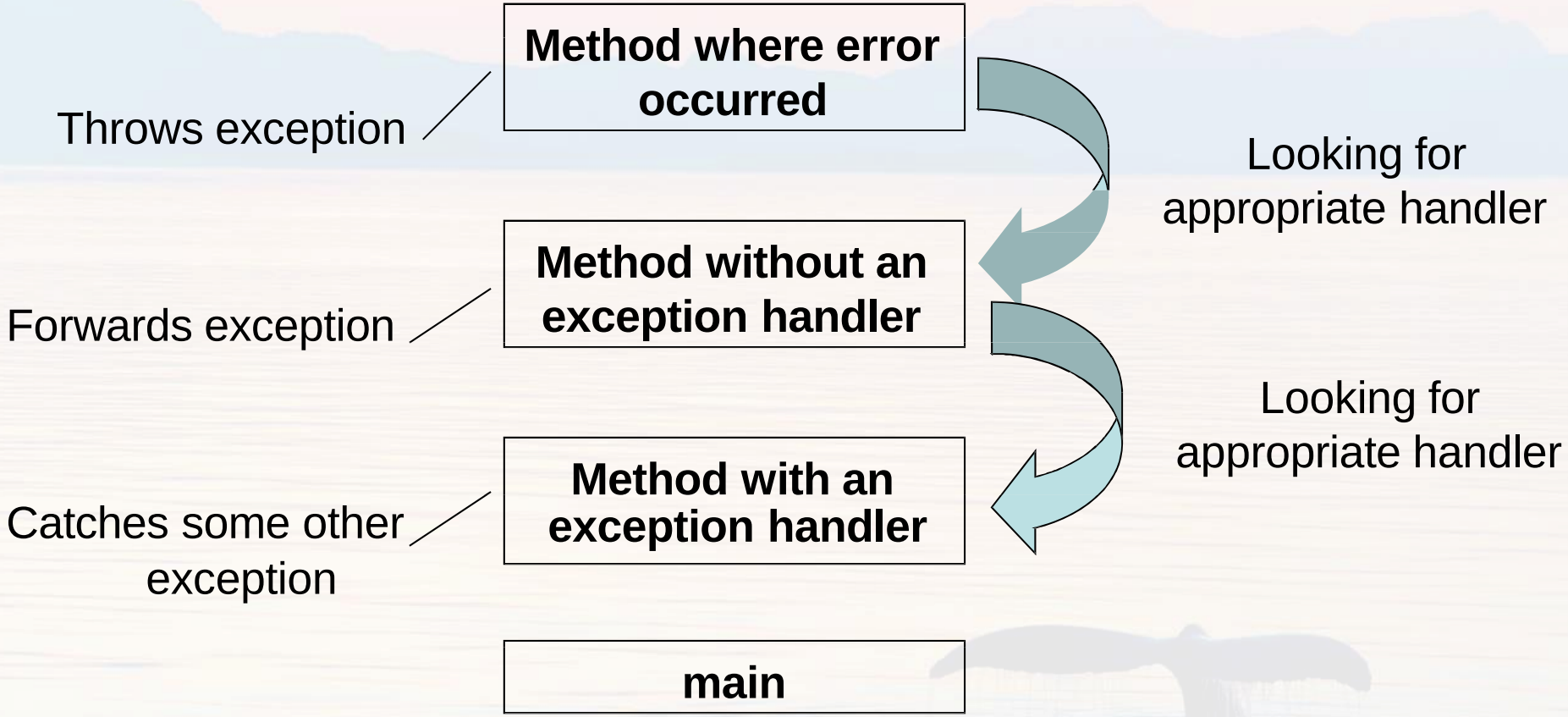
What Is an Exception?

- An **exception** (“exceptional event") is an event, which occurs during the execution of a program that disrupts the normal flow of the program’s instructions.
- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The **object (“exception object”)** contains information about the error.
- Creating an exception object and handing it to the runtime system is called “**throwing an exception**”

What Is an Exception?



What Is an Exception?



Searching the call stack for the exception handler

The Catch or Specify Requirement

- Code that might throw certain exceptions must be enclosed by either of the following:
  - A try statement that catches the exception – catching and handling exception
  - A method that specifies that it can throw the exception. The method must provide a throws clause – Specifying the Exceptions Thrown by a Method
- Three Kinds of Exceptions
  - checked exception: can anticipate and recover the exception. Checked exceptions are subject to the Catch or Specify Requirement (CSR). All exceptions are checked exceptions, except for *Error*, *RuntimeException*, and their subclasses.
  - error (unchecked exception): cannot anticipate and recover, not subject to CSR, ex) system malfunction
  - runtime exception (unchecked exception): cannot anticipate and recover, not subject to CSR, ex) logic error or improper use of an API

Catching and Handling Exceptions

The try, catch, and finally block

```
try {
  // try block
}
catch (ExceptionType1 param1) {
  // Exception Block
}
catch (ExceptionType2 param2) {
  // Exception Block
}
 ……
catch (ExceptionTypeN paramN) {
  // Exception Bloc
}
finally {
  // finally Block
}
```

Statements that have some possibilities to generate exception(s).

Execute statements here when the corresponding exception occurred.

Do always

Catching and Handling Exceptions

```
//Note: This class won't compile by design!
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {

    private Vector vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(new Integer(i));
        }
    }
```

If can't be written the file "OutFile.txt"?

```
    public void writeList() {
        PrintWriter out = new PrintWriter(
            new FileWriter("OutFile.txt") );

        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " +
                        vector.elementAt(i) );
        }
        out.close();
    }
}
```

If the program tries to access the index of SIZE +1?

Catching and Handling Exceptions

```
public class ListOfNumbers {
   private Vector<Integer> victor;
   private static final int SIZE = 10;

   public ListOfNumbers () {
        victor = new Vector<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++)
            victor.addElement(new Integer(i));      }

    public void writeList() {
        PrintWriter out = null;
  try {
        System.out.println("Entering try statement");
      out = new PrintWriter(new
        FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
                out.println("Value at: " + i + " = " +
       vector.elementAt(i));
       }
```

There may be some exception..

```
catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Caught
ArrayIndexOutOfBoundsException: " +
                                e.getMessage());
}
catch (IOException e) {
            System.err.println("Caught IOException: "
+ e.getMessage());
 }
finally {
            if (out != null) {
                System.out.println("Closing
PrintWriter");
                out.close();
            } else {
                System.out.println("PrintWriter not
open");
            }
        }
    }
}
```

When the "ArrayIndexOutOfBounds Exception" has occurred…

try, catch, and finally

The finally clause is used to clean up internal state or to release non-object resources, such as open files stored in local variables.

```
public boolean searchFor (String file, String word) throws StreamException
{
  Stream input = null;

  try {
      input = new Stream(file);
      while (!input.eof())
        if (input.next().equals(word)) return true;
      return false;       // not found
   } finally {
        if (input != null) input.close();
   }
}
```

Specifying the Exceptions Thrown by a Method

- If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions.

```
public void writeList() {
    // This method won't compile by design!
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
            out.println("Value at: "+i+"="+vector.elementAt(i));
    }
    out.close();
}
```

- To specify that the writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method.

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException
{
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++){
        out.println("Value at: "+i+"="+vector.elementAt(i));
    }
    out.close();
}
```

The throws Clause

- The checked exceptions that a method throws are as important as the type of value it returns. Both must be declared.
- If you invoke a method that lists a checked exception in its throws clause, you have three choices:
  - Catch the exception and handle it.
  - Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own throws clause.
  - Declare the exception in your throws clause and let the exception pass through your method.
- Throws clauses and Method Overriding: An overriding or implementing method is not allowed to declare more checked exceptions in the throws clause than the inherited method does.

Review of Catch or Specify Requirement

```
import java.io.*;
public class TestCSRException {
 public void noNeedException() {
   // No need the exception handling
   int i = 100;
   System.out.println("i = " + 100);   }
 public void useTryCatch() {
   String name;
   System.out.print("What is your name? ");
   // The code needs exception handling
   try {
     BufferedReader charStream = new
     BufferedReader (new
     InputStreamReader(System.in));
     name = charStream.readLine().trim();
   } catch(IOException e) {
   System.out.println("IOException: " + );     }
   System.out.println("Your name is " +
     name);    }
```

```
public void useThrowsMethod() throws
     IOException {
   String name;
   System.out.print("What is your name? ");
   BufferedReader charStream = new
     BufferedReader (new
     InputStreamReader(System.in));
   name = charStream.readLine().trim();
   System.out.println("Your name is  " +
     name);  }

 public static void main(String[] args) throws
     IOException {
   TestException obj = new TestException();

   obj.noNeedException();
   obj.useTryCatch();
   obj.useThrowsMethod();
 }
}
```

How to Throw Exceptions

- Before we can catch an exception, some code somewhere must throw one. Regardless of what throws the exception, it's always thrown with the throw statement.
- The throw Statement

  throw someThrowableObject;

```
public Object pop () {
   Object obj;
   if (size == 0) {
     throw new EmptyStackException();
   }
   obj = objectAt(size – 1);
   setObjectAt (size – 1, null);
   size--;
   return obj;
}
```

If the stack is empty, pop instantiates a new EmptyStackException object and throws it.

Throw Statement

- **Throw Statement**
  throw expression;

Throw a **user created** exception

```
public class
    NoSuchAttributeException
    extends Exception
{
  public final String attrName;

  public NoSuchAttributeException
    (String name) {
    super("No attribute named ¥"" +
    name + "¥" found");
    attrName = name;
  }
}
```

```
public void replaceValue(String name,
    Object newValue)
  throws NoSuchAttributeException
{
  Attr attr = find(name);
  if (attr == null)
    throw new
    NoSuchAttributeException(name);
  attr.setValue(newValue);
}
```

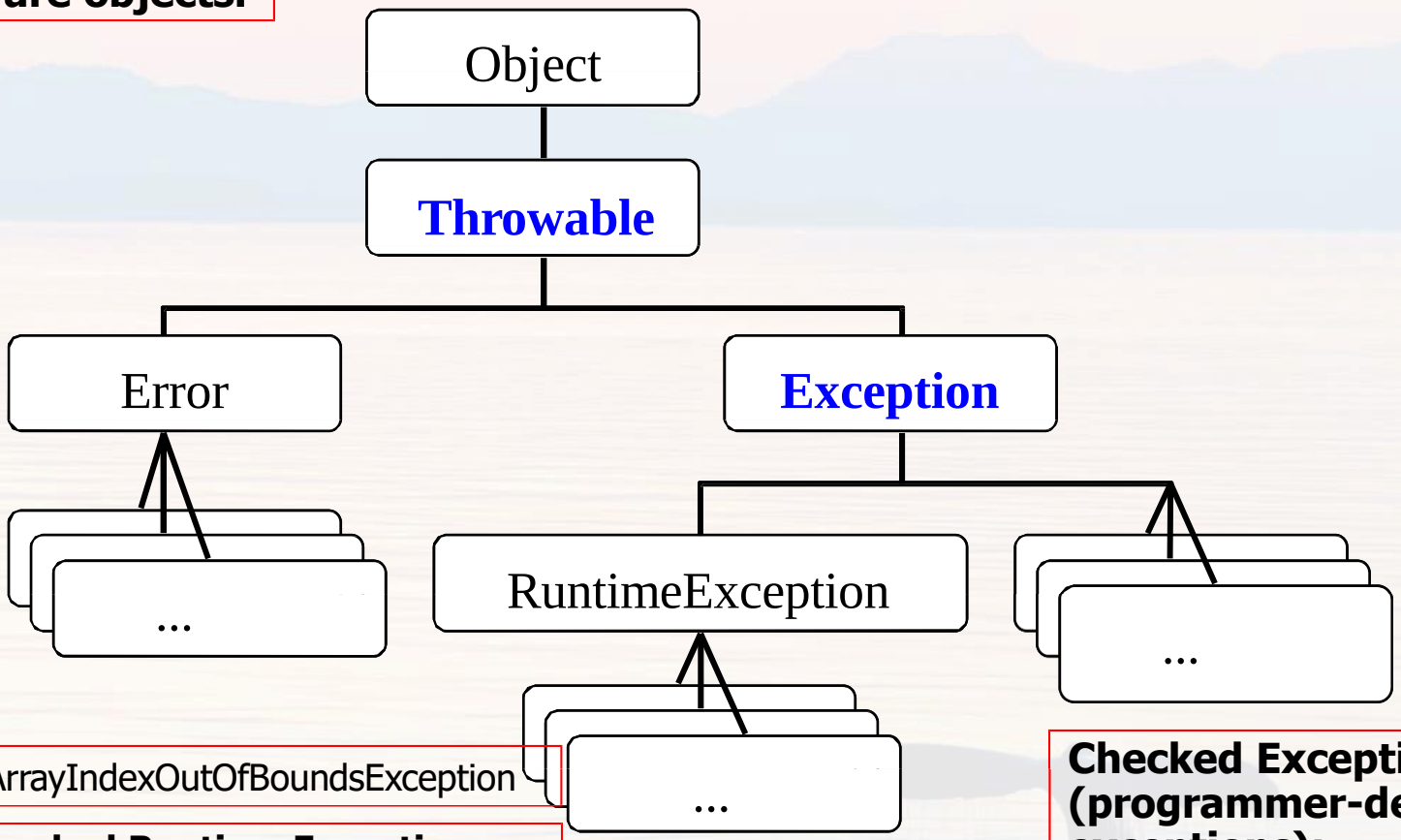
A Method with throws clause and Catch from it

- When the replaceValue method (shown in the slide #16) raise an “NoSuchAttributeException”, it is to be caught by the catch clause.
- And the exception will be handled by the code in the catch block.

```
Object value = new Integer(8);
try {
    attributedObj.replaceValue(“Age”,
            value);
}
catch (NoSuchAttributeException e)
{
    Attr attr = new Attr(e.attrName, value);
    attributedObj.add(attr);
}
```

Throwable Class and Its Subclasses

Exceptions are objects.



(ex)ArrayIndexOutOfBoundsException

Unchecked Runtime Exceptions (system-defined exception):
Reflect errors in program's logic

- Error and RuntimeException

Checked Exceptions (programmer-defined exceptions):

The compiler checks

Exception Chaining

Exceptions caused by other exceptions

```
public double[] getDataSet(String setName) throws BadDataSetException
{
  String file = setName + ".dset";
  FileInputStream in = null;
  try {
     in = new FileInputStream(file);
     return readDataSet(in);
  } catch (IOException e) {
      throw new BadDataSetException();
    } finally {
        try {
          if ( in != null )   in.close();
        } catch (IOException e) {
            ;  // ignore: we either read the data OK
               // or we're throwing BadDataSetException
               }
    }
}
// … definition of readDataSet
```

For the notion of one exception being caused by another exception

```
 } catch (IOException e) {
      BadDataSetException bdse =
         new BadDataSetException();
      bdse.initCause(e);
      throw bdse;
     } finally {
       // …..
}
```

initCause is used to remember the exception that made the data bad. Later, the **getCause** method is used to retrieve the exception.

Exception Chaining

Another Way: to define new exception class

```
class BadDatasetException extends Exception {
  public BadDataSetException() {}

  public BadDataSetException(String details) {
     super(details);
  }

  public BadDataSetException(Throwable cause) {
     super(cause);
  }

  public BadDataSetException(String details,
Throwable cause) {
     super(details, cause);
  }
}
```

```
} catch (IOException e) {
    throw
        new BadDataSetException(e);
} finally {
    // …
}
```

Now you can write like this!

Exception Handling (Exercise 1)

```
class DivideByZero {

 public static void main(String args[]) {
    a();
 }
 static void a() {
    b();
 }
 static void b() {
    c();
 }
 static void c() {
    d();
 }
 static void d() {
    int i = 1;
    int j = 0;
    System.out.println(i / j);
 }
}
```

Result :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZero.d(DivideByZero.java:22)
        at DivideByZero.c(DivideByZero.java:16)
        at DivideByZero.b(DivideByZero.java:12)
        at DivideByZero.a(DivideByZero.java:8)
        at DivideByZero.main(DivideByZero.java:4)
```

Exception Handling (Exercise 2)

```
class Divider {

  public static void main(String args[]) {

    try {


      System.out.println("Before Division");
      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i / j);
      System.out.println("After Division");
    }
    catch (ArithmeticException e) {
      System.out.println("ArithmeticException");
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("ArrayIndex" +
        "OutOfBoundsException");
    }
    catch (NumberFormatException e) {
      System.out.println("NumberFormatException");
    }
```

```
finally {
      System.out.println("Finally block");
}
}
}
```

No arguments, so no args[0], args[1]

Result: java Divider

Before Division

ArrayIndexOutOfBoundsException

Finally block

Result: java Divider 1 0

Before Division

ArithmeticException

Finally block

Exception Handling
(Exercise: Catch Block Searches)

```
class CatchSearch {

  public static void main(String args[])
  {
    try {
      System.out.println("Before a");
      a();
      System.out.println("After a");
    }
    catch (Exception e) {
      System.out.println("main: " + e);
    }
    finally {
      System.out.println("main: finally");
    }
  }

  public static void a() {
    try {
      System.out.println("Before b");
      b();
      System.out.println("After b");
    }
    catch (ArithmeticException e) {
      System.out.println("a: " + e);
    }
    finally {
      System.out.println("a: finally");
    }
  }
```

```
  public static void b() {
    try {
      System.out.println("Before c");
      c();
      System.out.println("After c");
    }
    catch
  (ArrayIndexOutOfBoundsException e)
  {
      System.out.println("b":   + e);
    }
    finally {
      System.out.println("b: finally");
    }
  }

  public static void c() {
    try {
      System.out.println("Before d");
      d();
      System.out.println("After d");
    }
    catch (NumberFormatException e) {
      System.out.println("c: " + e);
    }
    finally {
      System.out.println("c: finally");
    }
  }
```

```
  public static void d() {
    try {
      int array[] = new int[4];
      array[10] = 10;
    }
    catch (ClassCastException e) {
      System.out.println("d: " + e);
    }
    finally {
      System.out.println("d: finally");
    }
  }
}
```

After handle the exception, it is reached here.

Choosing a Super or Sub class

- We cannot put a superclass catch clause before a catch of one of its subclasses.

```
Class SuperException extends Exception {}
Class SubException extends SuperException {}
Class BadCatch {
  public void goodTry() {
   /* This is an INVALID catch ordering */
   try {
      throw new SubException();
   } catch (SuperException superRef) {
        // Catches both SuperException and SubException
    } catch (SubException subRef) {
      // This would never be reached
    }
}
```

Creating My Own Exception Types

- Why Create New Exception Type:
  - Adding useful data
  - Type of the exception is important part of the exception data.

<Example>

```
public class
    NoSuchAttributeException
    extends Exception
{
  public final String attrName;

  public NoSuchAttributeException
    (String name) {
    super("No attribute named ¥"" +
    name + "¥" found");
    attrName = name;
  }
}
```

Another Example Using Constructors

```
public class BadDataSetException
    extends Exception
{
    public BadDataSetException() {}

    public BadDataSetException(String s)
    { super(s);}

    public BadDataSetException(Throwable
    cause) { super(cause);}

    public BadDataSetException(String s,
    Throwable cause) { super(s, cause);}
 }
```

Another Example: Custom Exceptions

Subclass of Exception

```
import java.util.*;

class ExceptionSubclass {

  public static void main(String args[]) {
    a();
  }

  static void a() {
    try {
      b();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  static void b() throws ExceptionA {
    try {
      c();
    }
    catch (ExceptionB e) {
      e.printStackTrace();
    }
  }

  static void c() throws ExceptionA, ExceptionB {
    Random random = new Random();
    int i = random.nextInt();
    if (i % 2 == 0) {
      throw new ExceptionA("We have a problem");
    }
    else {
      throw new ExceptionB("We have a big problem");
    }
  }
}

class ExceptionA extends Exception {
  public ExceptionA(String message) {
    super(message);
  }
}


class ExceptionB extends Exception {
  public ExceptionB(String message) {
    super(message);
  }
}
```

In the "b" method, the ExceptionB is to be handled, but pass the ExceptionA, which will be caught in the method a().

Execution:

```
ExceptionA: We have a problem
        at ExceptionSubclass.c(ExceptionSubclass.java:31)
        at ExceptionSubclass.b(ExceptionSubclass.java:20)
        at ExceptionSubclass.a(ExceptionSubclass.java:11)
        at ExceptionSubclass.main(ExceptionSubclass.java:6)
```

The throw Statement

```
class ThrowDemo {

 public static void main(String args[])
 {
    try {
      System.out.println("Before a");
      a();
      System.out.println("After a");
    }
    catch (ArithmeticException e) {
      System.out.println("main: " + e);
    }
    finally {
      System.out.println("main: finally");
    }
  }

  public static void a() {
    try {
      System.out.println("Before b");
      b();
      System.out.println("After b");
    }
    catch (ArithmeticException e) {
      System.out.println("a: " + e);
    }
    finally {
      System.out.println("a: finally");
    }
```

```
  public static void b() {
    try {
      System.out.println("Before c");
      c();
      System.out.println("After c");
    }
    catch (ArithmeticException e) {
      System.out.println("b: " + e);
    }
    finally {
      System.out.println("b: finally");
    }
  }

  public static void c() {
    try {
      System.out.println("Before d");
      d();
      System.out.println("After d");
    }
    catch (ArithmeticException e) {
      System.out.println("c: " + e);
      throw e;
    }
    finally {
      System.out.println("c: finally");
    }
  }
```

```
  public static void d() {
    try {
      int i = 1;
      int j = 0;
      System.out.println("Before division");
      System.out.println(i / j);
      System.out.println("After division");
    }
    catch (ArithmeticException e) {
      System.out.println("d: " + e);
      throw e;
    }
    finally {
      System.out.println("d: finally");
    }
  }
}
```

Unchecked Exceptions – The Controversy

- Programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from *RuntimeException* because they can avoid the Catch or Specify Requirement.
- Exceptions are as much a part of that method's programming interface as its parameters and return value.
- Runtime exceptions represent problems that are the result of a programming problem (such as dividing by zero, pointer exceptions, etc.). They can occur anywhere in a program, and in a typical program they can be very numerous.
- Guideline
  - If a client can reasonably be expected to recover from an exception, make it a checked exception.
  - If a client cannot do anything to recover from the exception, make it an unchecked exception.

Advantages of Exceptions

- Seperating Error-Handling Code from “Regular” Code

```
readFile {
  try {
     open the file;
     determine its size;
     allocate that much memory;
     read the file into memory;
     close the file;
  } catch (fileOpenFailed) {
     doSomething;
  } catch (sizeDeterminationFailed) {
     doSomething;
  } catch (memoryAllocationFailed) {
     doSomething;
  } catch (readFailed) {
     doSomething;
  } catch (fileCloseFailed) {
     doSomething;
  }
}
```

- Grouping and Differentiating Error Types
  - Specific Handler

```
catch (FileNotFoundException e)
{
  ...
}
```

  - More General Handler

```
catch (IOException e) {
   ...
}
```

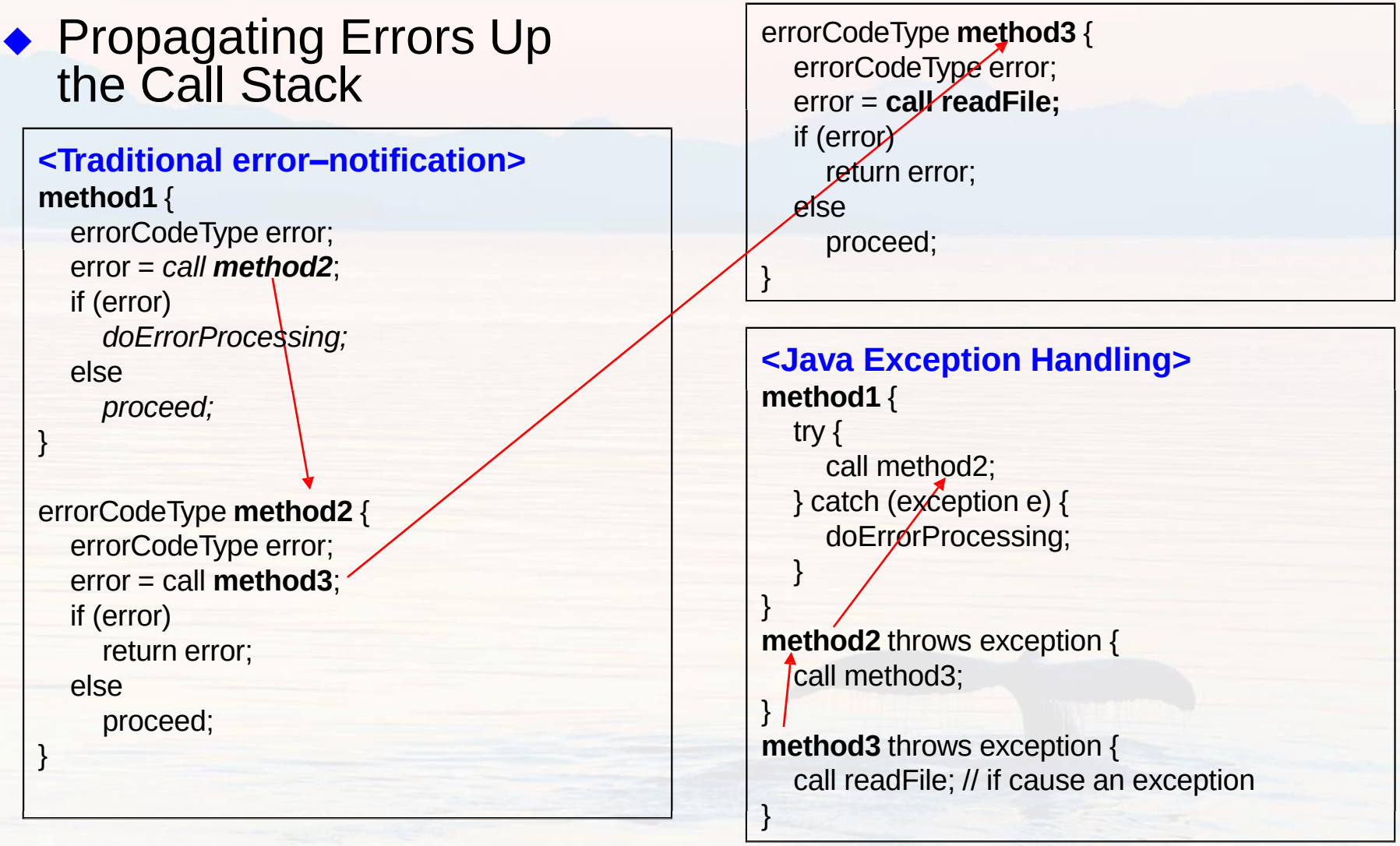
  - Most General Handler

```
catch (Exception e) {
   ...
}
```

  - One can create groups of exceptions and handle exceptions in a general fashion, or use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

Advantages of Exceptions

- Propagating Errors Up the Call Stack



Exception in C, C++, and Java

http://blog.csdn.net/ljlove2008/article/details/3076337

http://www.cnblogs.com/Z-D-/p/7170977.html