Object Oriented Programming Lecture

Exception Handling

School of Computing and Information Technology
Faculty of Engineering and Computing
University of Technology, Jamaica

Email: dwwhite@utech.edu.jm, taedwards@utech.edu.jm

Object Oriented Programming

Expected Outcome

At the end of this lecture the student should be able to:

- understand and be able to implement defensive programming using exception handling
- compare and contrast traditional error handling with exception handling
- know what type of errors exception handling is able to handle

Object Oriented Programming

Topics to be covered in this lecture:

- When things go wrong
- Traditional error handling
- Exception Handling
- Exception Handling Terminology
- Models of Exception Handling
- Checked vs. Unchecked Exceptions
- Unhandled Exceptions
- Quirks and Problems with Exception Handling

When things go wrong...

Many things can go wrong in a computer program:

- Errors can occur at compile time
- Errors can occur at run time
- Asynchronous errors can occur
- Synchronous errors can occur

When things go wrong...

Question:

- Which of these errors can be handled by the program while it is running?
- How to handle errors appropriately?

Compile Time Errors

- These are errors that surface when a program is being compiled
- These prevent the program from being compiled successfully
- For example, syntax errors in the code
- The following code will not compile in C++ or Java

Compile Time Errors

```
int count;
int sum=0;

for(count = 0, count < 10, ++count)
{
        sum = sum + count
}
```

The commas in the for statement should be replaced by semi-colons

This line is missing a semi-colon needed to terminate the statement

Run Time Errors

These occur while the program is running

Run time errors can be:

- Synchronous
- Asynchronous

Asynchronous Errors

These occur outside of your currently executing program

Therefore occurring independent of your program, Examples:

- Mouse and keyboard errors
- Network message errors
- Disk Input/output errors

Synchronous Errors

Occur while the statements of the program are being executed, some of the errors listed may occur:

- Divide by zero
- Invalid array index (array out of bounds)
- Invalid number format (entering a string when a number was expected)
- Overflow and underflow
- Null pointer dereferencing

Traditional Error Handling

Error handling involves dealing with errors that occur while the program is executing (synchronous errors)

Traditional error handling methods include:

- Interspersing error-handling code with code for normal execution
- Some approaches returned a special value (e.g. -1, 0, true or null) from a method to indicate than an error occurred
- Sometimes a global variable was used e.g. errno in the C programming language

Problems with Traditional Error Handling

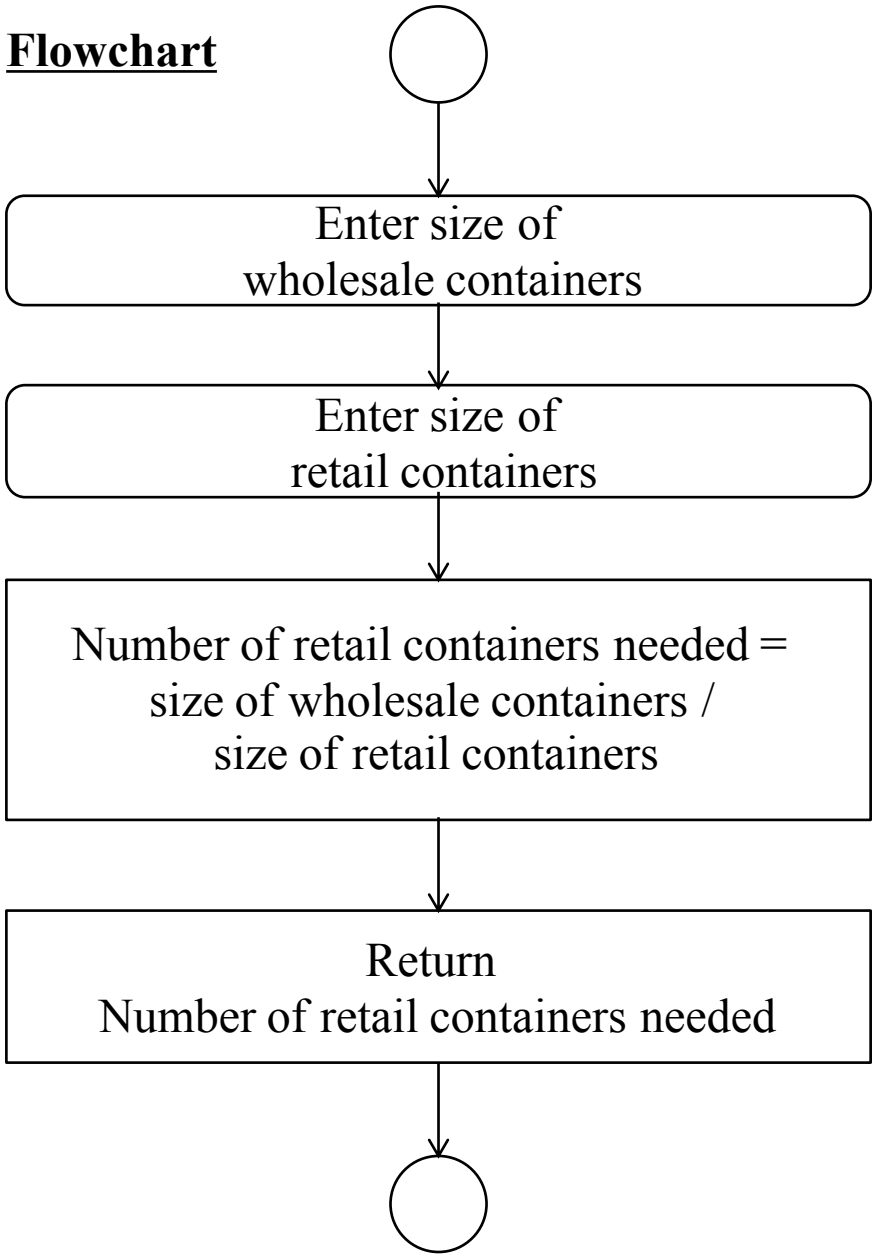
Some of the challenges faced with traditional error handling include:

- Normal code becomes cluttered with error handling code
- Programmer has to check the value returned by a method to see if an error occurred
- Error code returned has to be same data type as valid value returned
- Difficult to return error code in some cases e.g. a method that returns valid values **true** or **false** – there is no other value to use for errors

Traditional Error Handling

Consider the following algorithm...

Flowchart



Pseudocode

Enter size of wholesale containers

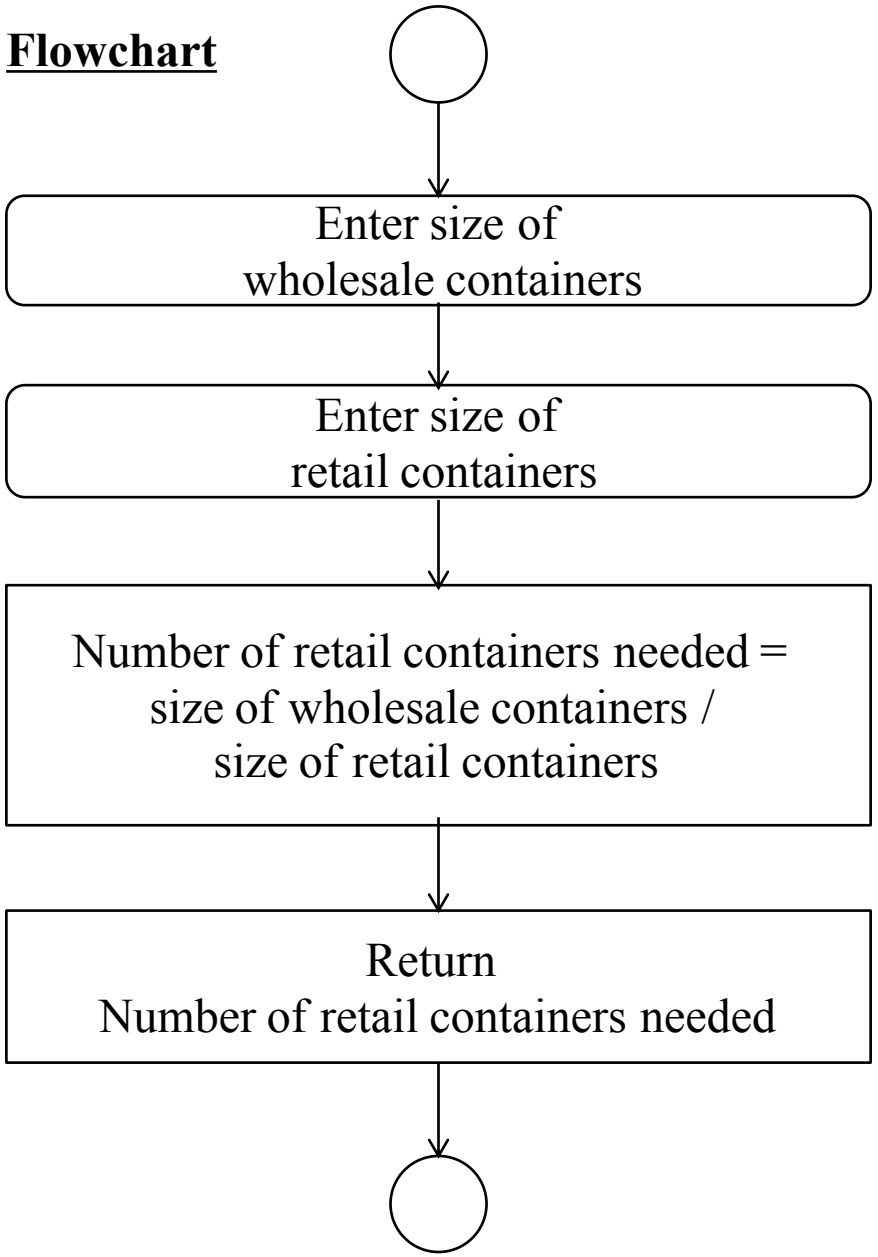
Enter size of retail containers

Let Number of retail containers needed =
size of wholesale containers /
size of retail containers

Return Number of retail containers needed

Traditional Error Handling

Flowchart



Pseudocode

Enter size of wholesale containers

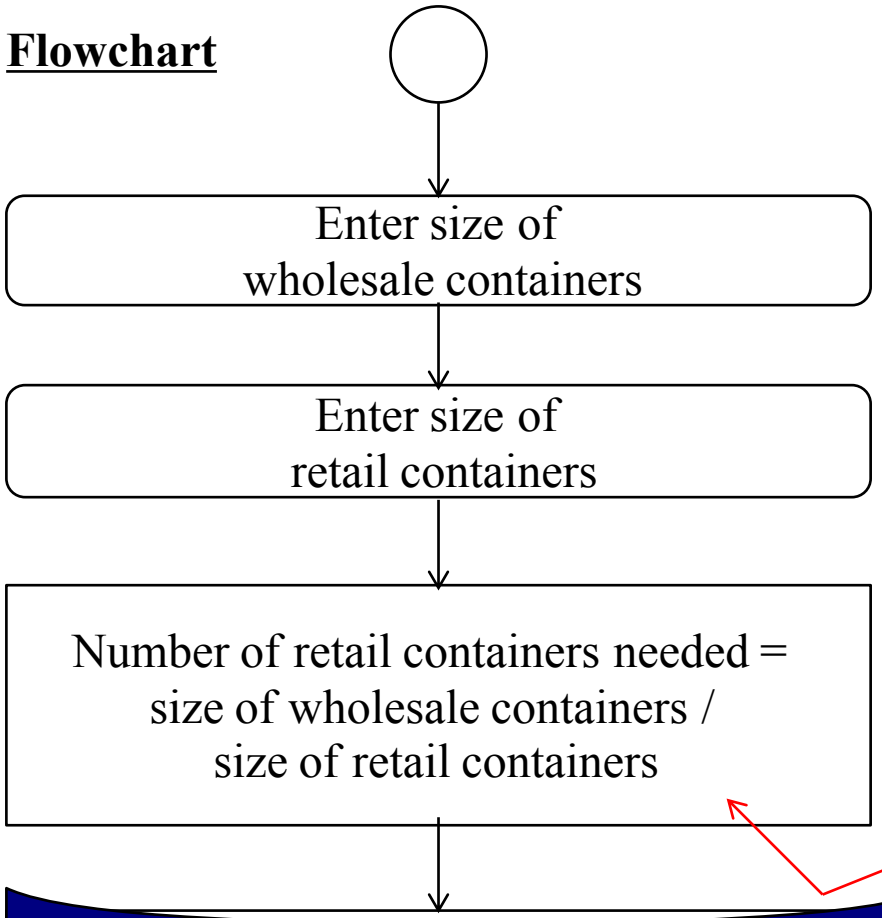
Enter size of retail containers

Let Number of retail containers needed =
size of wholesale containers /
size of retail containers

Return Number of retail containers needed

Traditional Error Handling

Flowchart



Pseudocode

Enter size of wholesale containers

Enter size of retail containers

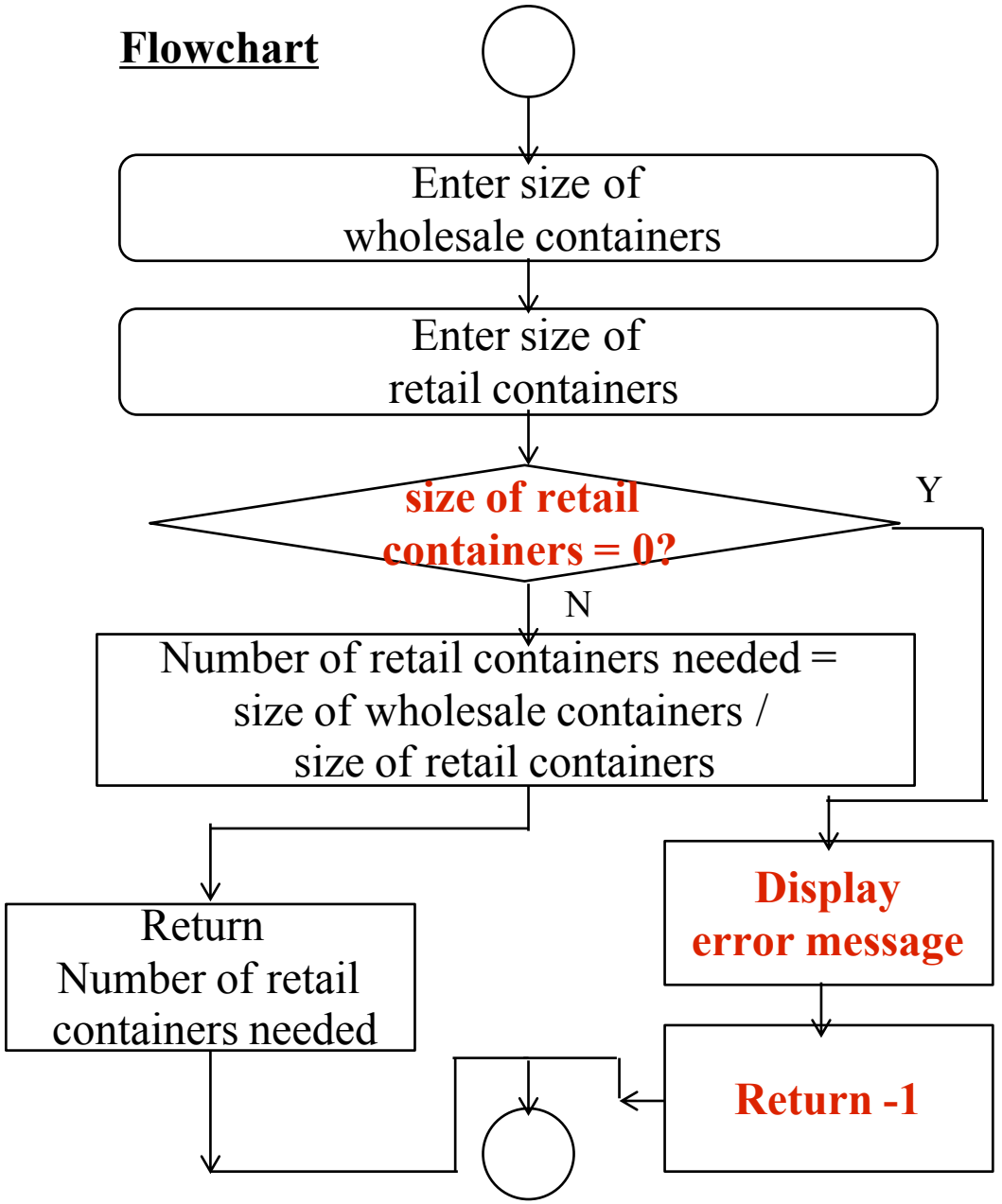
Let Number of retail containers needed =
size of wholesale containers /
size of retail containers

Return Number of retail containers needed

If the "size of retail containers" value is zero
the program will crash, since integer division by zero
Is not generally allowed in most OOP languages

Traditional Error Handling

Flowchart



Pseudocode

Enter size of wholesale containers

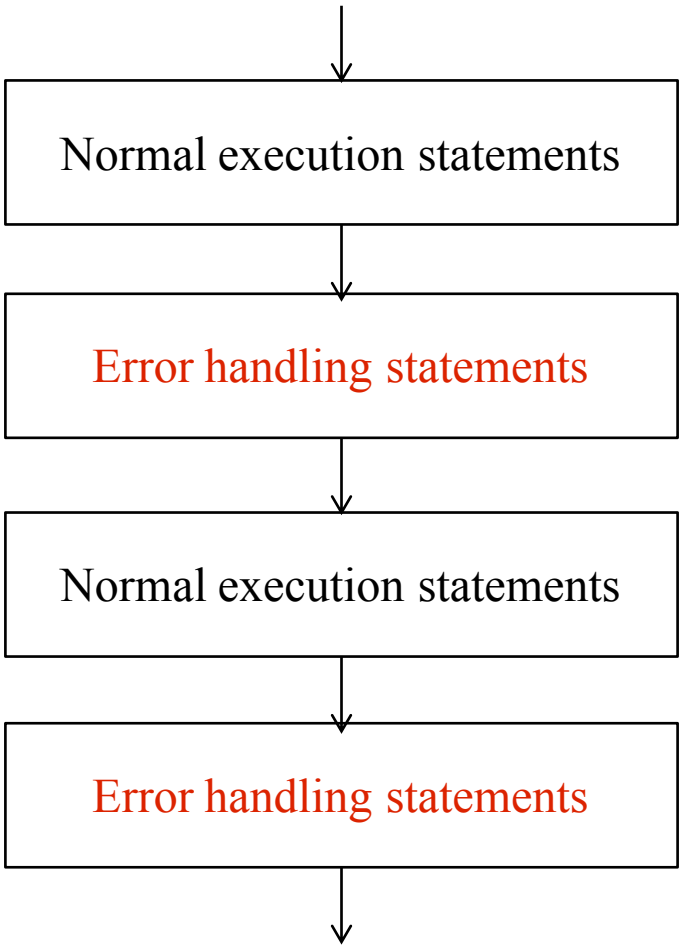
Enter size of retail containers

If size of retail containers is equal 0 then
Display error message
Return -1

Let Number of retail containers needed =
size of wholesale containers /
size of retail containers

Return Number of retail containers needed

Traditional Error Handling



.
.
.
Normal execution statements
Error handling statements

Normal execution statements
Error handling statements

Normal execution statements
Error handling statements
.
.
.

Exception Handling

An exception is a signal generated when an abnormal event has occurred in a running program

- Used to manage situations which are outside the normal flow of program execution
- Separates error-handling code from normal processing code
- Exceptions alter the flow of execution in a program – from normal processing to executing an exception handler

Exception Handling

Exception handling allows errors to be trapped and an appropriate message displayed to the user informing that an error has occurred

- Uses exceptions to indicate or signal that an error has occurred
- Exceptions are separate from the return values of methods
- Exceptions can be logged so that the developer can use the log to debug the program at a later date

Exception Handling Terminology

When an error occurs, an exception is generated

- This is called **raising** or **throwing** the exception
- The block of code where exceptions can be thrown is called the **try block**
- The block of code to handle the exceptions generated is known as the **catch handler** (or **catch block**)
- The point in the code where the exception is generated is known as the **throw point**

Exception Handling Terminology

Term	Description
Exception	**An object, attribute, variable or value that is used to signal that a particular runtime error has occurred**
Throw an exception	**Generate an exception to indicate that an error has occurred during the running of a program**
Raise an exception	**Same as throw an exception**
Throw point	**The line of code from which an exception was generated**
Catch handler	**The portion of code that is written to handle exceptions thrown.**
Exception handler	**The code involved in the try and catch blocks**
Try Block	**A block of code which surrounds normal code and which may contain one or more throw points**
Catch Block	**A block of code written to handle a specific exception. There may be more than one catch blocks**

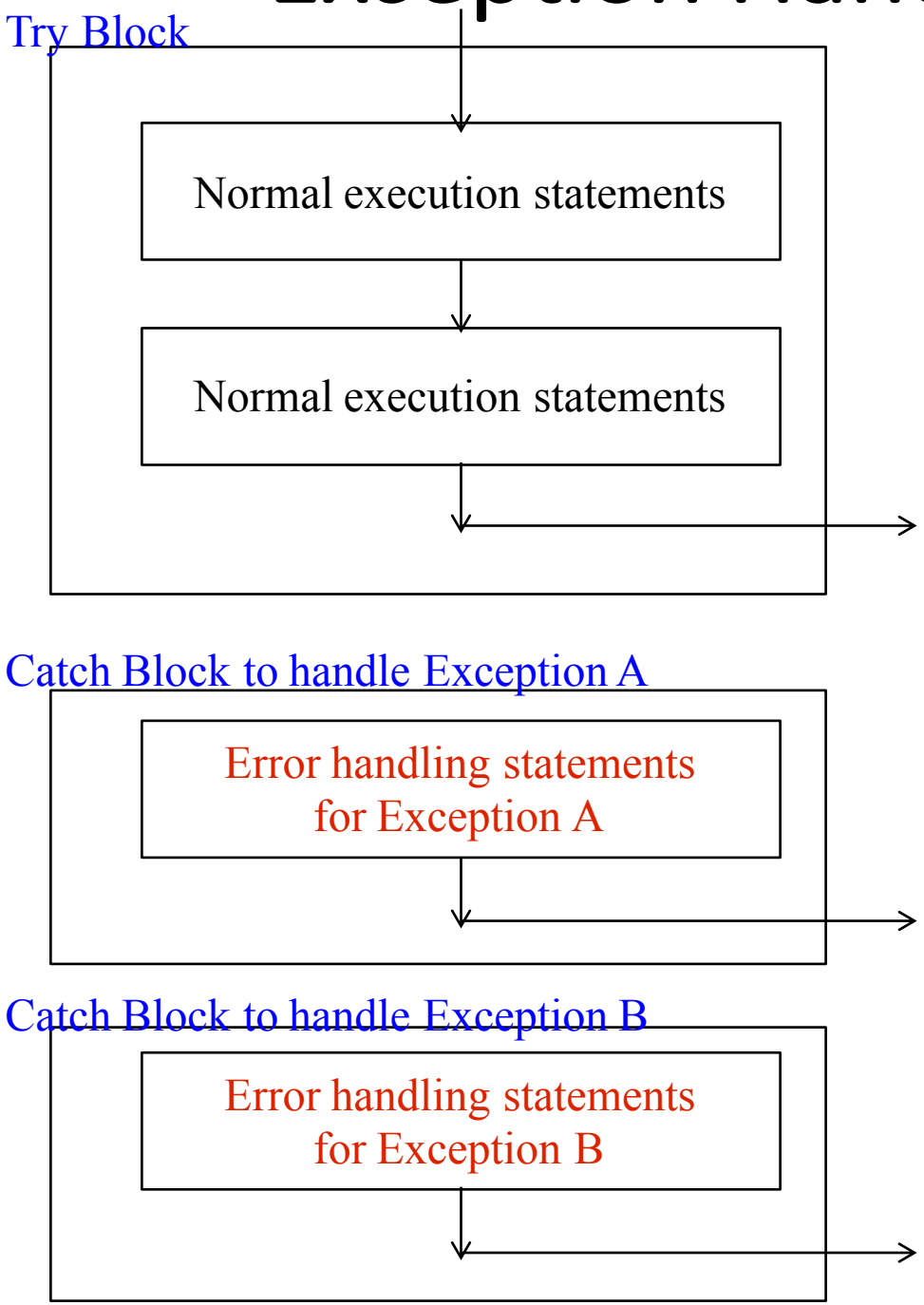
Exception Handling - Benefits

- ✓ **Separates error handing code from regular code,** handling the exception in a catch handler
- ✓ If possible, **allows program to run normally after exception has occurred**
- ✓ If this is not possible, **allows a graceful shutdown of the program** instead of an abrupt termination
- ✓ Sufficiently **handle unexpected occurrences**

Exception Handling - Benefits

- ✓ Improve **fault tolerance**
- ✓ Improve **robustness**
- ✓ Improve **reliability**
- ✓ provides **defensive programming capabilities**

Exception Handling Framework




```
Try block
        Normal execution statements
        Normal execution statements
        Normal execution statements
        Normal execution statements

Catch Block for Exception A
        Error handling statements

Catch Block for Exception B
        Error handling statements

Catch Block for Exception C
        Error handling statements
```

Exception Handling - Models

Resumption model

- In the resumption model of exception handling, statement execution resumes with the line following the one that raised the exception
- Used in languages such as BASIC

Exception Handling - Models

Termination model

- In the termination model of exception handling, the block of code containing the line that raised the exception is immediately terminated and control is passed to the exception handler in the method if one is present, otherwise the method immediately terminates
- Used in languages such as C++ and Java

Checked vs. Unchecked Exceptions

Unchecked Exceptions

- These are exceptions that a program is not required to handle – the program will compile successfully even if no exception handler is present

Checked Exceptions

- These are exceptions that a program must handle. The program will not compile if no exception handler is present or if the exception is not declared (so it can be propagated forward).

Checked vs. Unchecked Exceptions

Unchecked Exceptions

- All exceptions in C++ are unchecked, but only exceptions that inherit from **RuntimeException** in Java are unchecked.

Checked Exceptions

- All exceptions that inherit from class **Exception** in Java are checked exceptions (*apart from those that inherit from class **RuntimeException***). All checked exceptions in Java must be handled or declared.

Unhandled Exceptions

If an unchecked exception is raised or thrown but there is no catch handler for that exception...

- The exception is propagated upward until an appropriate catch handler is found or the program terminates
- This process is called **stack unwinding**
- However, if an appropriate catch handler is found then the exception is considered handled and stack unwinding stops

Unhandled Exceptions

For example:

- If main() called A() and A() called B() and an unhandled exception is thrown in B(),
- B() will terminate and return to A(),
- If the exception is not handled in A(), A() will terminate and return to main()
- If the exception is still not handled in main(), main() too will terminate and the entire program will terminate abnormally

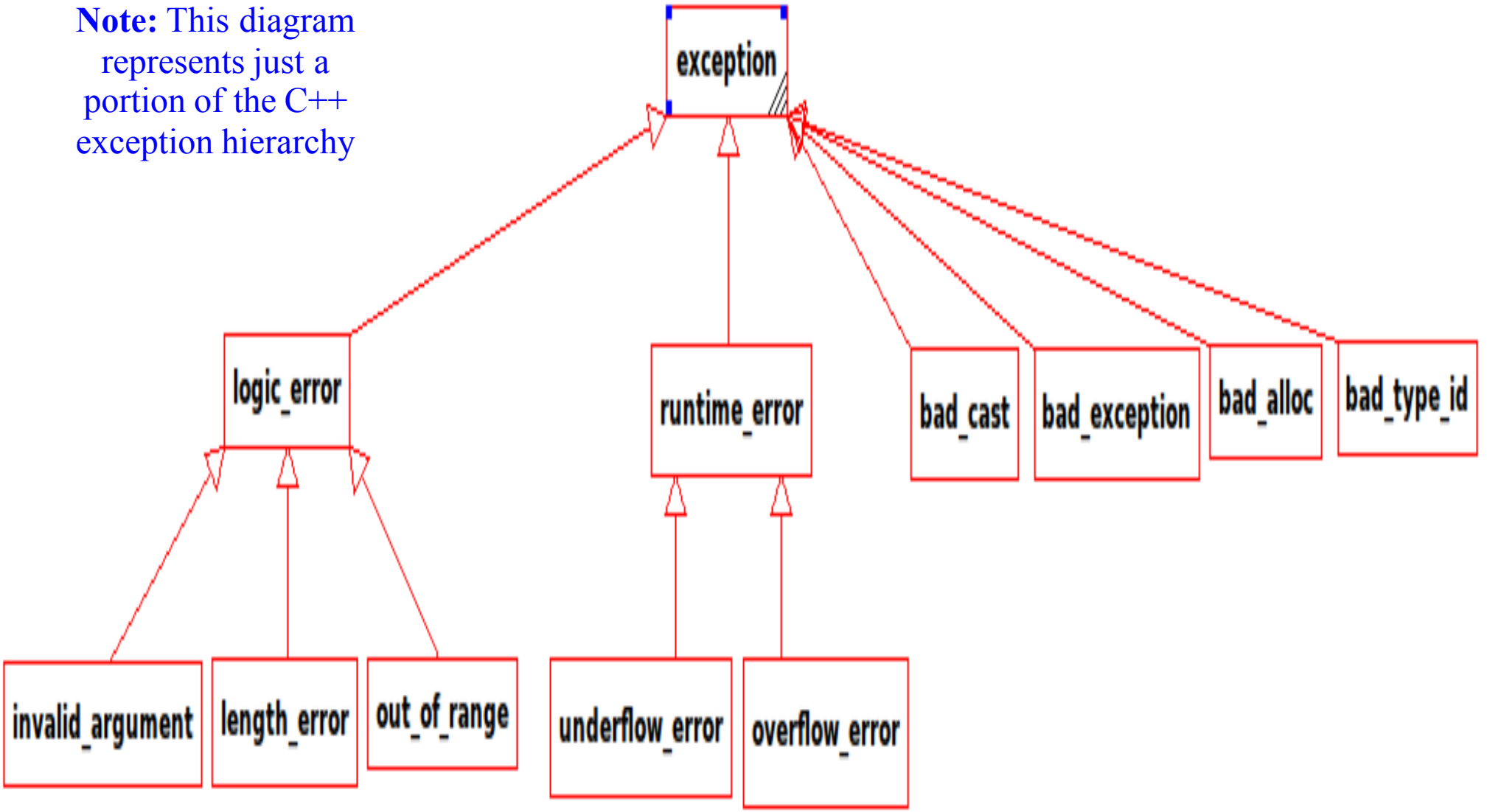
Exception Hierarchy

Both C++ and Java have exception hierarchies in which various exception classes are sub classes of some base exception or error class

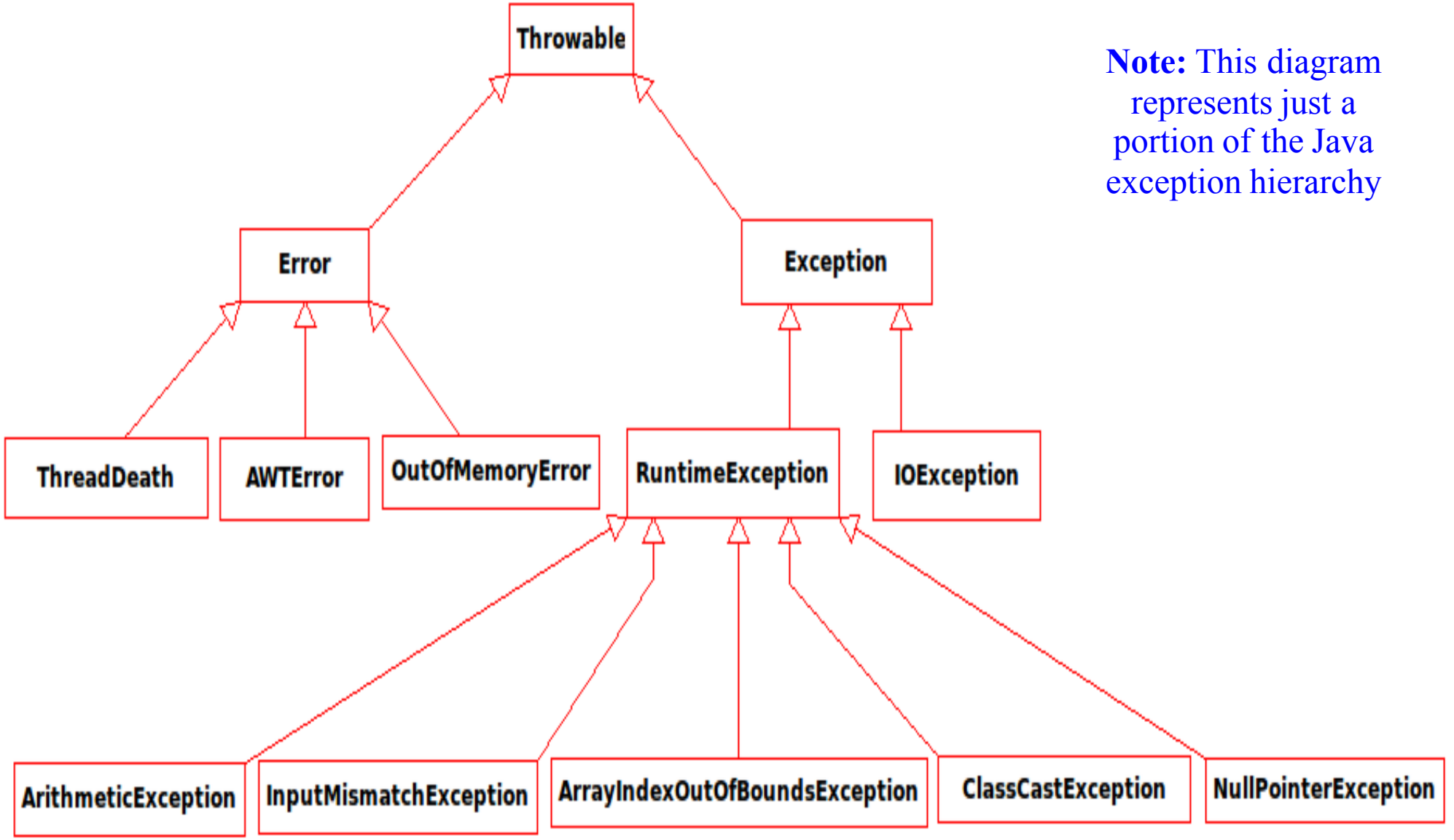
- Objects of the child class exceptions are also considered exceptions of their respective parent classes
- For instance, you could code a catch handler to catch **exception** objects in C++ or **Exception** objects in Java, and this would catch all exceptions of the type of the parent as well as those of type of the child classes

C++ Exception Hierarchy

Note: This diagram represents just a portion of the C++ exception hierarchy



Java Exception Hierarchy



Note: This diagram represents just a portion of the Java exception hierarchy

Creating New Exception Classes

You can define your own exception classes by inheriting from the existing exception classes

C++

```
#include <stdexcept>
using namespace std;

class MyException : public runtime_error
{
        public:
                //default constructor
                MyException()
                        //pass message to parent
                        // class constructor
                        : runtime_error("This  is my exception!")
                {

                }
};
```

Java

```
public class MyException extends Exception
{
        //default constructor
        public MyException()
        {
                //pass message to parent
                // class constructor
                super("This is my exception!");
        }
}
```

Try Block

This block is used to enclose normal processing code that might generate an exception

- Each try block must be followed by one or more catch block (in Java – if you have a finally block then the catch block is optional)
- Try blocks may be nested

Try Block

C++

```
int amt;

try
{
	cout << "Now in " <<
	"
CalculateRetailContainersNeeded()"
<< endl;

	amt = wcSize / rcSize;
}
```

Java

```
int amt;

try
{
	System.out.println("Now in
" + "
CalculateRetailContainersNeeded()"
);

	amt = wcSize / rcSize;
}
```

Catch Handlers

C++

```
catch(int)
{
        cout << "Divisor cannot be
zero\n";
        cout << "Setting divisor to 1\n";

        rcSize = 1
}
catch(string)
{
        cout << "Invalid input\n";
}
catch(MyException  &e)
{
        cout << "There was a problem\n";
        cout << e.what();
}
catch(...)
{
        cout << "An unspecified error
occurred\n";
}
```

Java

```
catch(ArithmeticException  e)
{
        System.out.println("Divisor
cannot be zero");
        System.out.println(e.getMessage()
);
        RcSize = 1;
}
catch(InputMismatchException  e)
{
        System.out.println(e.getMessage()
);
}
catch(MyException  ex)
{
        System.out.println(ex.getMessage(
));
}
catch(Exception  e)
{
        System.out.println("Exception:  "+
         e.getMessage());
}
```

Catch All Exceptions

C++:

```
catch (...)
 {

 }
```

Java:

```
catch(Exception e)
 {

 }
```

Warning – never use catch all as the first catch block, since catch blocks are checked in the order they appear - No other catch would be checked after a catch all

Explicitly Throwing Exceptions

In C++ exceptions have to be explicitly thrown by the programmer in code

Java allows the programmer to explicitly throw exceptions, but it can also automatically throw some exceptions, for example, a division by zero at runtime causes Java to automatically throw an **ArithmeticException**

Explicitly Throwing Exceptions

- In Java only exception objects can be thrown, while in C++ any data type, value or object can be thrown
- In both languages, the keyword to throw an exception is **throw**

Explicitly Throwing Exceptions

C++

```
try
{
        cout << "Now in " <<
        " CalculateRetailContainersNeeded()"  << endl;
        if (rcSize == 0)
                throw -1;
        return wcSize / rcSize;
}
catch(int)
{
        cout << "Divisor cannot be zero\n";
        cout << "Setting divisor to 1\n";
        rcSize = 1;
}
catch(...)
{
        cout << "An unspecified error occurred\n";
}
```

Java

```
try
{
        System.out.println("Now  in " +
        " CalculateRetailContainersNeeded()");
        if (rcSize == 0)
                throw new Exception();
        return wcSize / rcSize;
}
catch(ArithmeticException  e)
{
        System.out.println("Divisor  cannot be zero");
        System.out.println("Setting  divisor  to 1");
        rcSize = 1;
}
catch(Exception  e)
{
        System.out.println(
                "An unspecified  error  occurred");
        System.out.println(e.getMessage());
}
```

Explicitly Throwing Exceptions

C++

In C++ we have to test for the divisor being zero and if so explicitly throw an exception

```
try
{
        cout << "Now in " <<
        " CalculateRetailContainersNeeded()"  << endl;
        if (rcSize == 0)
                throw -1;
        return wcSize / rcSize;
}
catch(int)
{
        cout << "Divisor cannot be zero\n";
        cout << "Setting divisor to 1\n";
        rcSize = 1;
}
catch(...)
{
        cout << "An unspecified error occurred\n";
}
```

In Java, though we can also Explicitly throw an exception, it is not always necessary as some exceptions like "ArithmeticException" are thrown automatically when divide by zero is attempted

Java

```
");
        if (rcSize == 0)
                throw new Exception();
        return wcSize / rcSize;
}
catch(ArithmeticException  e)
{
        System.out.println("Divisor  cannot be zero");
        System.out.println("Setting  divisor  to 1");
        rcSize = 1;
}
catch(Exception  e)
{
        System.out.println(
                "An unspecified  error occurred");
        System.out.println(e.getMessage());
}
```

Re-throwing Exceptions

Once an exception is processed by a catch block it is considered as being handled

- What if the method containing the catch handler wants to propagate the exception up through the call stack?

Re-throwing Exceptions

Re-throwing an exception allows a method to send a just-handled exception up to a calling method to indicate that the exception occurred

- For example, if main() calls A() and A() calls B(), and B() handles an exception, it can rethrow that exception so it A() too can handle it, and A() in turn can rethrow it to main() for processing there.

Re-throwing Exceptions

Re-throwing an exception is done inside the catch block / catch handler

- Keyword format C++:

throw;

(note that in C++ the throw statement is by itself to indicate a re-thrown exception)

- Keyword format Java:

throw ex;

(where ex is the exception object caught)

Re-throwing Exceptions

C++

```
try
{
        cout << "Now in " <<
        " CalculateRetailContainersNeeded()"  << endl;
        if (rcSize == 0)
                throw -1;
        return wcSize / rcSize;
}
catch(int)
{
        cout << "Divisor cannot be zero\n";
        cout << "Setting divisor to 1\n";
        RcSize = 1;
        throw;
}
catch(...)
{
        cout << "An unspecified error occurred\n";
        throw;
}
```

Java

```
try
{
        System.out.println("Now in " +
        " CalculateRetailContainersNeeded()");
        if (rcSize == 0)
                throw new Exception();
        return wcSize / rcSize;
}
catch(ArithmeticException e)
{
        System.out.println("Divisor cannot be zero");
        System.out.println("Setting divisor to 1");
        RcSize = 1;
        throw e;
}
catch(Exception e)
{
        System.out.println(
                "An unspecified error occurred");
        System.out.println(e.getMessage());
        throw e;
}
```

Exception Specifiers/Declaration

State that a method can throw a specific exception or set of exceptions

Exceptions that can be thrown are specified after the argument list but before the method body using the keywords **throw** in C++ or **throws** in Java

C++:

```
void method( ) throw (int, string) {   }
```

Exception Specifiers/Declaration

Java:

void method( ) throws SQLException, IOException { }

- C++ and Java differ on how exception specifiers are treated by the language compilers
- Become part of the signature of the method in C++ but not in Java

Exception Specifiers/Declaration

C++

```
//this method throws an int, a string and a double
//notice that the exceptions are not handled
//(meaning there is no try-catch) it is just declared
//so if exceptions occurred they will have to be
//handled by the calling method

//perform exposure based on range specified
void method1(int pRangeSetting)
        throw (int, string, double)
{
        int exposure;

        if (pRangeSetting < 0)
                throw -1;
        if (pRangeSetting == 0)
                throw “error-exposure is zero”;
        if (pRangeSetting > 99)
                throw 99.0;

        exposure = pRangeSetting  * 33;

        performExposure(exposure);
}
```

Java

```
//this method uses the built in System.in.read()
//method to get a single y or n response from the
//user. The method automatically throws an
//IOException if something goes wrong. This
//exception is a checked exception so must
//be declared since we dont have try-catch here

//Get response (y/n) from user
public char getAnswer() throws IOException
{
        char ans; //single char needed to store 'y' or 'n'

        System.out.println("Enter  y or n:");
        ans = ' ';

        //loop until y or n is entered by user
        while (ans != 'y' && ans != 'Y'
                && ans != 'n' && ans != 'N')
        {

                ans = (char) System.in.read();
        }
        return ans;
}
```

Handling Exceptions which Occur in called Methods

In the previous slide, both the C++ and Java methods had exception specifiers declaring the type of exceptions which could be thrown

- However these exceptions were not handled by those methods
- In such a case, a calling method can choose to handle any exceptions raised/thrown by the called method
- Or it can propagate those exceptions upward to its caller too

Handling Exceptions which Occur in called Methods

C++

```
void method2()
{
        try
        {
                int ps;
                cout << “Please enter setting”;
                cin >> ps;
                method1(ps);
        }
        catch(int)
        {
                cout << “error – negative number”;
        }
        catch(string)
        {
                cout << “error – zero is illegal setting”;
        }
        catch(double)
        {
                cout << “error – setting greater than 99”;
        }
}
```

Java

```
public void processResponse()
{
        try
        {
                char c;
                c = getAnswer();
        }
        catch(IOException exc)
        {
                System.out.println(
                        "An input error occurred");
                System.out.println(exc.getMessage());
        }
}
```

In these methods, we have handled the exceptions thrown by the called method (method1() in C++ and getAnswer() in Java.

Handling Exceptions which Occur in called Methods

C++

```
void method2()
        throw (int, string, double)
{
        int ps;
        cout << "Please enter setting";
        cin >> ps;
        method1(ps);
}
```

Java

```
public void processResponse()
        throws IOException
{
        char c;
        c = getAnswer();


}
```

In these versions of the same methods, we have not handled the exceptions that may be thrown/raised by the called methods,ward but instead we have just allowed them to be propagated upward

Java's Finally Block

Because both C++ and Java use the termination model of exception handling it is possible that some code will not get executed when an exception is thrown

- For instance, a file or database may have been opened or memory allocated, then the exception occurs and is handled before the code to close the file/database or release the memory was executed.

Java's Finally Block

This causes resource leaks

- Java tries to correct this by providing a finally block which always gets executed whether or not an exception is thrown. (except when System.exit() is used)
- C++ does not have a corresponding finally block

Java's Finally Block

Java

Note:

- In C++ every try block must be followed by at least one catch block
- In Java the catch block is optional if there is a finally block
- Also in Java, the finally block is optional if there is a catch block
- However there must be one or the other or else the Java compiler will generate an error

```
try
{
        //normal code here
}
catch(Exception e)
{
        //optional catch handler here
}
finally
{
        //optional finally block here
}
```

Quirks and Problems with Exception Handling

- Exception handling can lead to **surprisingly irregular execution of code**
- Exception handling can **hide underlying errors** by trapping the exceptions thrown but not sufficiency distilling or handling the error that caused the exception

Quirks and Problems with Exception Handling

- Encourages **lazy programmers**
- Exception handling can **cause resource leaks** a resource is acquired and used by a method but an exception occurs before the resource is released

When not to use Exception Handling

Never used exception handling for normal flow of control in a program