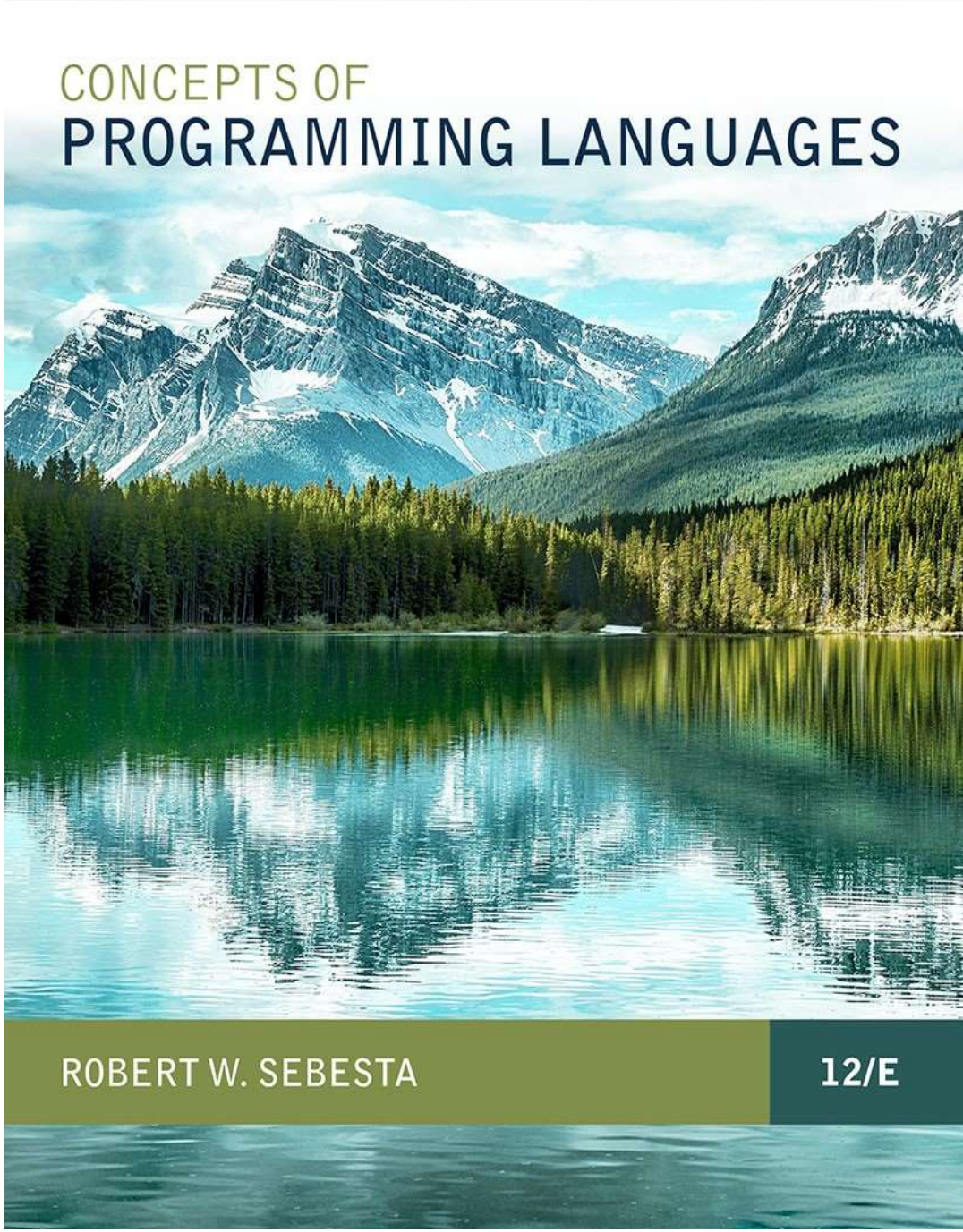
Lecture 10 (Chapter 14)

Exception Handling



ISBN 0-321-49362-1

Chapter 14 Topics

- Introduction to Exception Handling
- Exception Handling in C++
- Exception Handling in Java

Introduction to Exception Handling

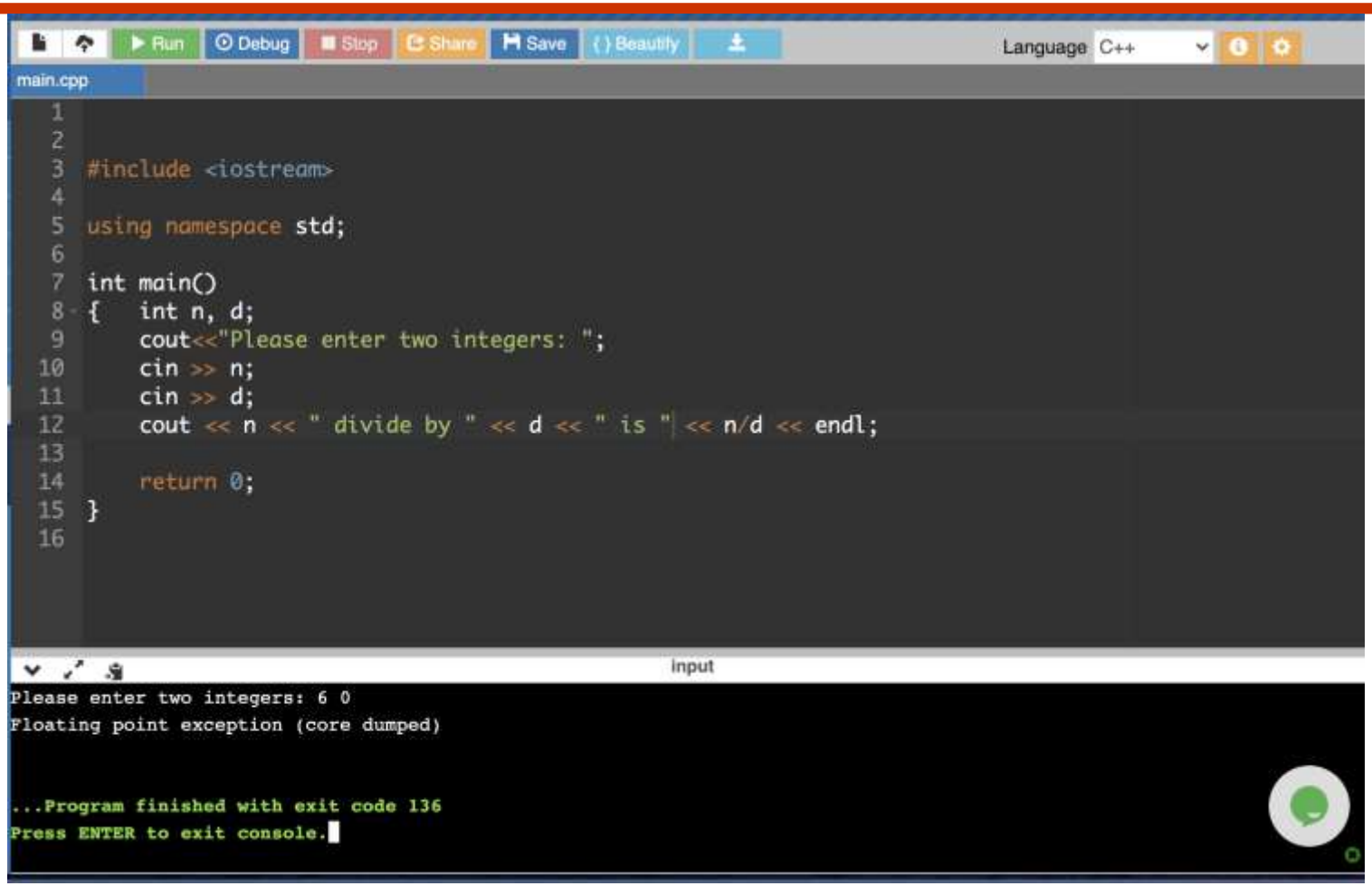
- Early languages no language constructs for exception handling
- In a language without exception handling
  - When an exception occurs (e.g. divide by zero) control goes to the operating system, where an error message is displayed and the program is terminated
  - A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions (user defined, software detected), e.g.
    - Use built-in function to detect exceptions

      if (fin.eof()) { … end of file error … }
    - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram, e.g.

      int main () { … if (no error) return 0; else return 1; }
    - Pass an exception handling subprogram to all subprograms
    - …

How did programmers survive at early days when there wasn’t any language supported exception handling mechanism?

Scenario 1: When exception occurs control goes to OS where error msg displayed and program terminated



C++: Divide by zero exception

Scenario 2: Programmer adds protection

```
if (d == 0 )
        cout << "divide by zero error\n";
else
        cout << "quotient is: " << n/d << "\n";
cout << "now we continue";
```

Compare with previous case, now in case d is 0, a message will be displayed and program will continue to execute.

Scenario 3: Aided with built-in function

```
//fin: a file stream opened successfully
while (!fin.eof()) {  //while not end of file
        data = fin.read();
        …
}
```

!eof() protects read() from hitting an exception when no more data to read. Similar strategy could be used for preventing other exceptions such as file not exist when opening.

Without built-in functions such as eof(), programmers are not able to code the protection.

Scenario 4: errors occurred in subprograms

```
double my_div (double n, double d, int & flag) {
        if (d == 0) {flag = 1; return 0.0;  }
        else { flag = 0; return n/d;}
}

int main () {
 int error;
 …
 result = my_div (5, 0, error);    //a status flag from the subprogram
 if (error) { return 1; } else { … display result …}
 …
}
```

Now, almost every newer language comes with language supported exception handling construct.

Basic Concepts

- In a language with exception handling
  - Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing
- An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called *exception handling*
- The exception handling code unit is called an *exception handler*
- An exception is *raised* when its associated event occurs

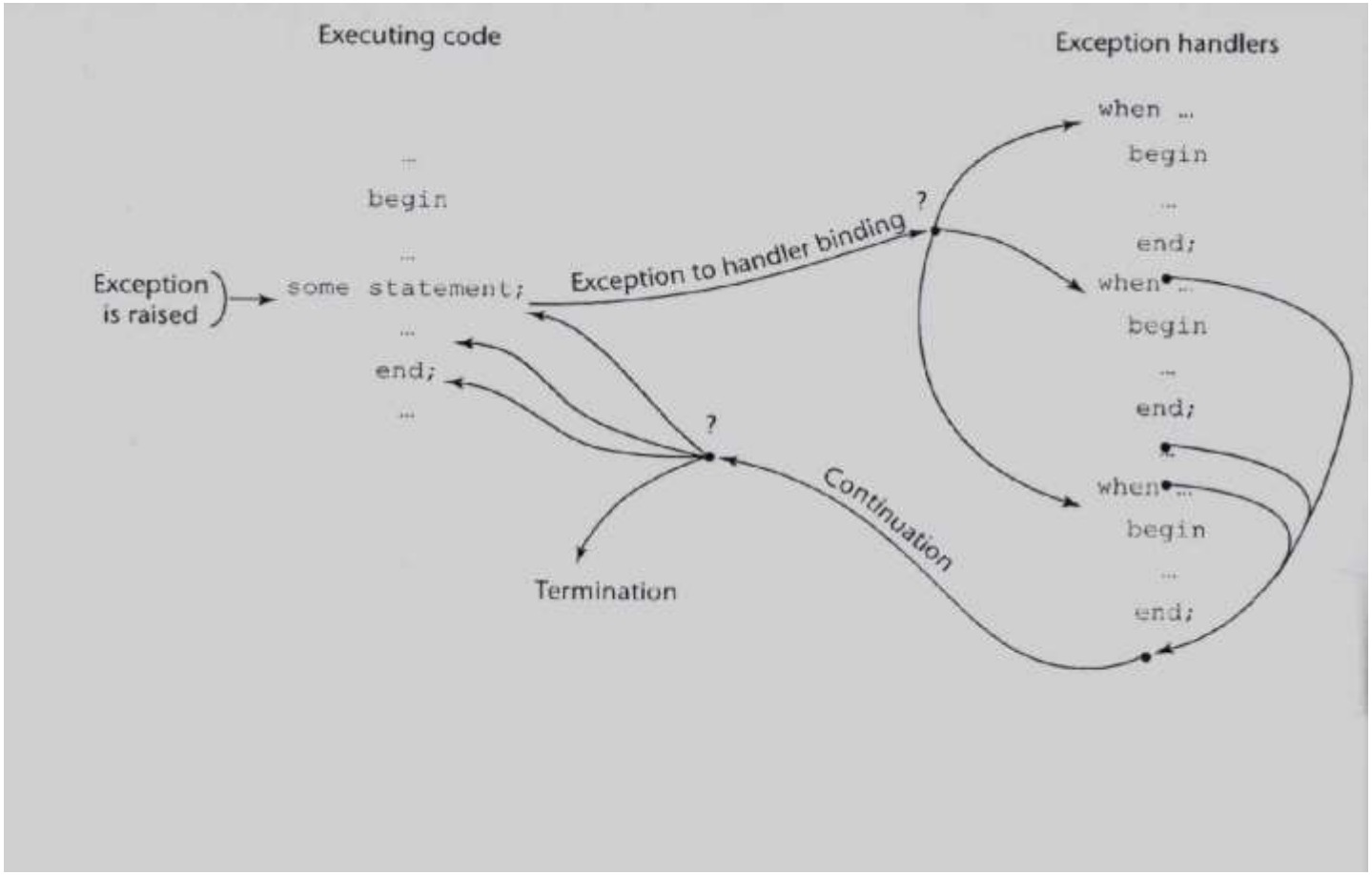
Exception Handling: Advantages

- Traditionally error detection code is tedious to write
- Exception handling encourages programmers to consider many different possible errors
- Exception propagation allows a high level of reuse of exception handling code

Design Issues

- How and where are exception handlers specified and what is their scope?
- How is an exception bound to an exception handler?
- Where does execution continue, if at all, after an exception handler completes its execution?
- Is some form of finalization provided?
- How are user-defined exceptions specified?
- Are there any predefined exceptions?
- Should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that can be handled?

Exception Handling Control Flow



Exception Handling in C++

- Added to C++ in 1990
  - Design is based on that of CLU, Ada, and ML
- Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

The throw and catch of Exceptions

- Exceptions are all raised explicitly by the statement:

  throw [*expression*];

  - A throw without an operand can only appear in a handler; when it appears, it simply re-raises the exception
  - The type of the expression disambiguates the intended handler
- catch is the name of all handlers
  - the formal parameter of each catch must be unique
  - The formal parameter need not have a variable
    - It can be simply a type name to distinguish the handler it is in from others
  - The formal parameter can be used to transfer information to the handler
  - The handler catch (…) will catch any unhandled exception

```
double division(int a, int b) {
  if( b == 0 ) {
    throw ”Div by zero";
  }
  if (a == 0) {
    throw 0.0;
  return (a/b);
}
```

```
int main () {
   …
   cin >> a;
   cin >> b;

  try {
    z = division(x, y);
    cout << z << endl;
  }
  catch (const char* msg) {
   cerr << msg << endl;
  }
  catch (double d) {
   cerr << “not like” << d << ”as result”;
  }
  catch (…) { cerr << “other errors”; }
            //e,g. error in cin >> a;
  return 0;
}
```

Control Flow

- If a throw statement is executed, control goes to the catch block
  - If a handler is matched, after a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers;
  - If no handler is matched, the exception is unhandled
- Unhandled exceptions
  - An unhandled exception is propagated to the caller of the function in which it is raised
  - This propagation continues to the main function
  - If no handler is found, control goes to the OS (to call default exception handler provided by the OS)

Control Flow: Python vs. C++

```
try :
        A
except myException :
        B
else :
        C
finally :
        D
```

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

else block: executed when no exception occurs
finally block: always executed.

Default handler: Python vs. C++

Python:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    except:
        print(“Catch’em all!”)
    else:
        print("result is", result)
    finally:
        print("finally clause")
```

C++:

```
try {
    z = division(x, y);
  }

  catch (…) { cerr << “other errors”; }

  return 0;
}
```

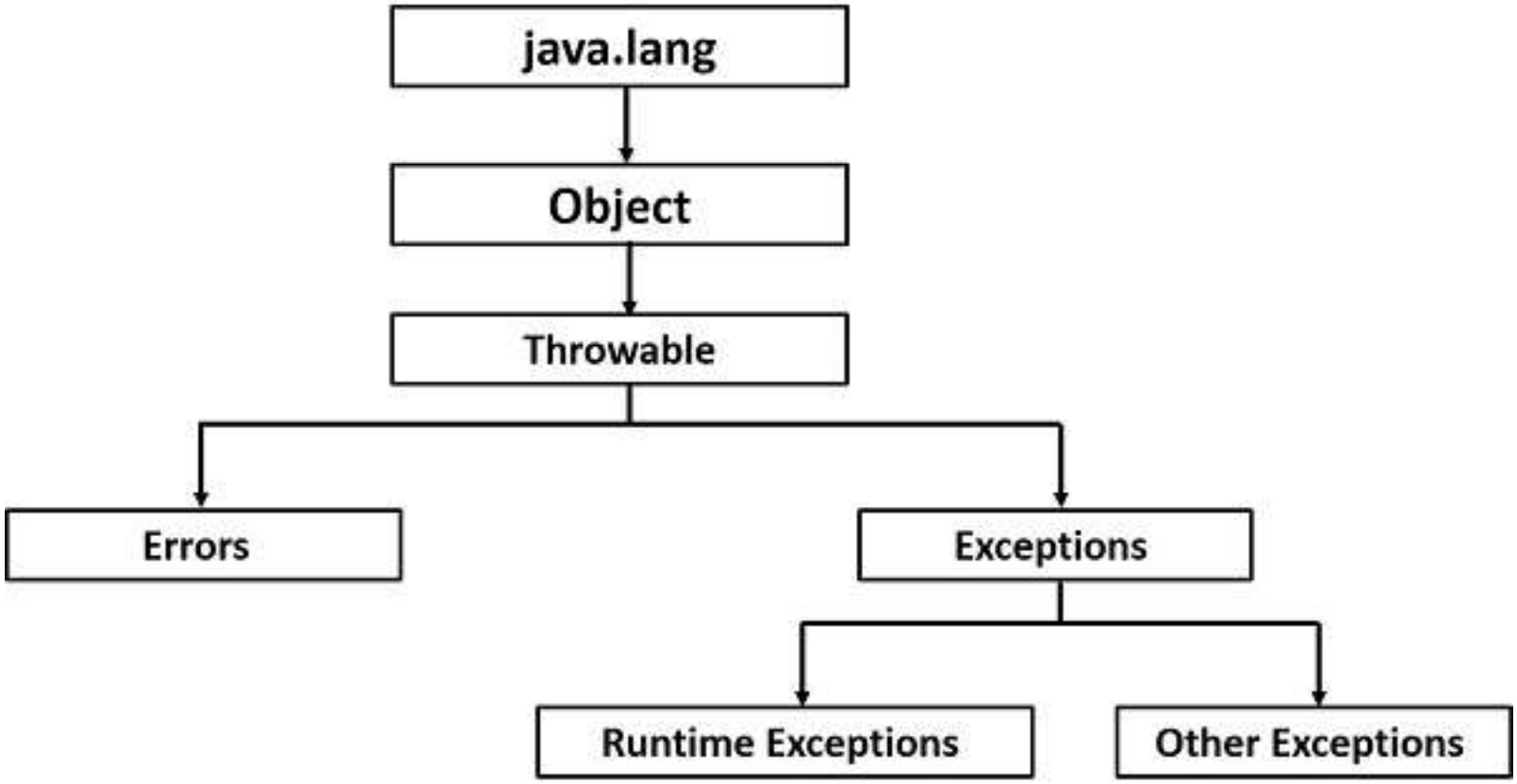
What about Java?

Features and Evaluation

- Exceptions are neither specified nor declared
- Binding exceptions to handlers through the type of the parameter certainly does not promote readability
- Functions can list the exceptions they may raise
  - Without a specification, a function can raise any exception (the throw clause)
- There are no predefined exceptions
  - All exceptions are user-defined
  - a list of standard exceptions defined in <**exception**>
- It is odd that exceptions are not named and that hardware- and system software-detectable exceptions cannot be handled

Exception Handling in Java

- Based on C++, but more in line with OOP philosophy
- All exceptions are objects of classes that are descendants of the *Throwable* class
- The Java library includes two subclasses of Throwable :
  - Error
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - Exception
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses
      - IOException e.g., EOFException
      - RuntimeException e.g., ArrayIndexOutOfBoundsException


java.lang
Object
Throwable
Errors
Exceptions
Runtime Exceptions
Other Exceptions

Reference

Java Exception Handlers

- Every catch requires a named parameter and all parameters must be descendants of Throwable
- Syntax of try clause is exactly that of C++
- Exceptions are thrown with throw, as in C++
- An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it
  - If no handler found, it propagates to the caller as in C++
  - It is suggested to add a handler (as the last catch clause) to catch all exceptions
    - Simply use an Exception class parameter
- An exception can be handled and rethrown by including a throw in the handler (a handler could also throw a different exception)

Checked and Unchecked Exceptions

- Unchecked Exceptions
  - Exceptions of class Error and RunTimeException and all of their descendants (all others are checked exceptions)
- Checked Exceptions
  - Checked exceptions are checked at compile-time.
    - It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.
- throw vs. throws (Example next slide)
  - A method that calls a method with a particular checked exception in its throws clause has three alternatives:
    - Catch and handle the exception
    - Catch the exception and re-throw an exception (could be a different exception) listed in its own throws clause
    - Declare the same exception in its throws clause and do not handle it

throw vs. throws

```
void myMethod() {
   try {
      //throwing arithmetic exception using throw
      throw new ArithmeticException("Something went wrong!!");
   }
   catch (Exception exp) {
      System.out.println("Error: "+exp.getMessage());
   }
}

class Example {
   public static void main(String args[]) throws IOException
   {                                   //IOException will not be handled in main()
      FileInputStream fis = null;
      fis = new FileInputStream("B:/myfile.txt");
       …
      fis.close();
   }
```

The finally Clause

- Can appear at the end of a try construct

```
finally {
...
}
```

- Purpose: To specify code that is to be executed, regardless of what happens in the try construct

```
class TestFinallyBlock{
  public static void main(String args[]) {
  try {
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e) {
        System.out.println(e);
   }
   finally {
        System.out.println(”always executed");
   }
   System.out.println("rest of the code...");
  } //end of main
} //end of class
```

Evaluation

- The types of exceptions makes more sense than in the case of C++
- The throws clause makes clear indication for (unhandled) exception propagation.
  - However, confusion between throw and throws
- The finally clause is often useful
- The Java interpreter throws a variety of exceptions that can be handled by user programs
  - A large set of built-in exceptions (Reference)
- Complexity in checked and unchecked exceptions
- pros and cons?

Summary

- History
  - Ada, one of the first languages with comprehensive exception handling features, provides extensive exception-handling facilities with a large set of built-in exceptions.
- C++ includes no predefined exceptions
- Exceptions are bound to handlers by connecting the type of expression in the throw statement to that of the formal parameter of the catch function
- Java exceptions are similar to C++ exceptions except that a Java exception must be a descendant of the Throwable class. Additionally Java includes a finally clause