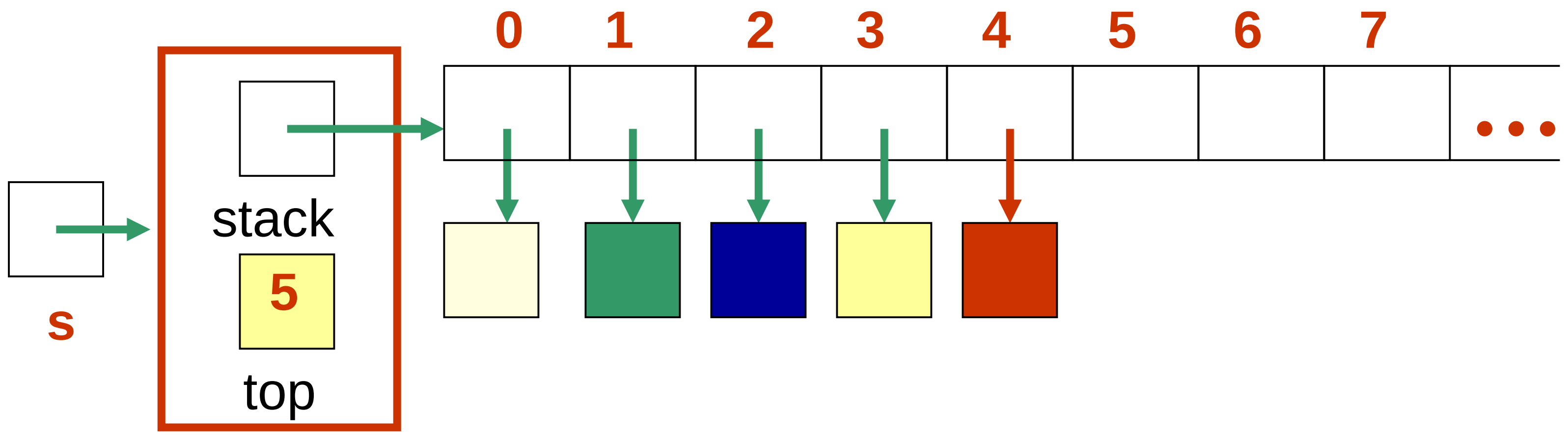
Introduction to Exceptions in Java

Recall the Array Implementation of a Stack



Incorrect implementation of the pop operation in class ArrayStack

```
public T pop( ) {
    top--;
   T result = stack[top];
   stack[top] = null;
   return result;
}
```

Let the Main Method be This

```
public class TestStack {
   public static void main (String[] args) {
      ArrayStack<String> s = new ArrayStack<String>(5);
      String line;
      line = s.pop();
            . . .
      }
   }
}
```

The Program Crashes

```
public class TestStack {

   public static void main (String[] args) {
      ArrayStack<String> s = new ArrayStack<String>();
      String line;
      line = s.pop();
         . . .
      }
   }
```

When we run this program we get this error message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
            at ArrayStack.pop(ArrayStack.java:87)
            at TestStack.main(TestStack.java:6)
```

Exceptions

- ***Exception***: an abnormal or erroneous situation at runtime
- Examples:
    - Array index out of bounds
    - Division by zero
    - Illegal input number format
    - Following a null reference

Exceptions

- These erroneous situations ***throw an exception***
- Exceptions can be thrown by the *Java virtual machine* or by the *program*

The Error Messages Contain Useful Information

```
public class ArrayStack<T> implements StackADT<T> {
    .
    .
    .
    public T pop( ) {
        top--;
      87: T result = stack[top];
      stack[top] = null;
      return result;
    }
```

The error message tells us where the error is:

```
Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: -1
        at ArrayStack.pop(ArrayStack.java:87)
        at TestStack.main(TestStack.java:6)
```

The Error Messages Contain Useful Information

```
public class TestStack {

   public static void main (String[] args) {
      ArrayStack<String> s = new ArrayStack<String>();
      String line;
      6: line = s.pop();
          . . .
      }
   }
```

The error message tell sus where the error is:

```
Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: -1
            at ArrayStack.pop(ArrayStack.java:87)
            at TestStack.main(TestStack.java:6)
```

We Can Fix the Problem by Using Java Exceptions

```
public T pop( ) {
   if (isEmpty( ))
        what should we do?
   top--;
   T result = stack[top];
   stack[top] = null;
   return result;
}
```

We Can Fix the Problem by Using Java Exceptions

```
public T pop( ) throws EmptyCollectionException {
   if (isEmpty( ))
        throw new EmptyCollectionException(“Stack” );
   top--;
   T result = stack[top];
   stack[top] = null;
   return result;
}
```

Notice the specification of where exceptions are thrown in the Stack ADT

```
public interface StackADT<T> {
  //  Adds one element to the top of this stack
  public void push (T dataItem);
  //  Removes and returns the top element of this stack
  public T pop( ) throws EmptyCollectionException;
  //  Returns the top element of this stack
  public T peek( ) throws EmptyCollectionException;
  //  Returns true if this stack is empty
  public boolean isEmpty( );
  //  Returns the number of elements in this stack
  public int size( );
  //  Returns a string representation of this stack
  public String toString( );
}
```

Ignorig Exceptions

```
public class TestStack {
   public static void main (String[] args) {
      ArrayStack<String> s = new ArrayStack<String>();
      String line;
      line = s.pop();
            . . .
      }
   }
}
```

If we ignore the exception thrown by method pop, when running this program we get this error:

```
Exception in thread “main” EmptyStackException: The Stack is empty.
            at ArrayStack.pop(ArrayStack.java.76)
            at TestStack.main(TestStack.java:6)
```

Catching and re-Throwing Exceptions

The calling method can either catch an exception or it can re-throw it.

- The method catches the exception if it knows how to deal with the error.
- Otherwise the exception is re-thrown.

Catching Exceptions

```
public class TestStack {

   public static void main (String[] args) {
      ArrayStack<String> s = new ArrayStack<String>();
      String line;
      try {
         line = s.pop();
            . . .
      }
      catch (EmptyStackException e) {
         // Error handling code
         System.out.println (e.getMessage());
      }
      . . .
   }
}
```

Catching Exceptions

- How try-catch works:
  - When the try-catch statement is executed, the statements in the try block are executed
  - If no exception is thrown:
    - Processing continues as normal
  - If an exception is thrown:
    - Program enters in "panic mode" and control is immediately passed to the first catch clause whose specified exception corresponds to the class of the exception that was thrown

Catching Exceptions

- If an exception is *not* caught and handled inside the method where it occurs:
  - Control is immediately returned to the *method that invoked the method* that produced the exception
  - If that method does not handle the exception (via a try statement with an appropriate catch clause) then control returns to the method that called it …
- This process is called ***propagating the exception***

Catching Exceptions

- Exception propagation continues until
  - The exception is caught and handled
  - Or until it is propagated out of the *main* method, resulting in the termination of the program

Catch Blocks

- A single catch block can handle more than one type of exception.
- This can reduce code duplication.
- In the catch clause we specify the types of exceptions that the block can handle and separate each exception type with a vertical bar (|).

A Try-Catch Example with Multiple Catch Statements

The try-catch syntax:

```
try {
    code
}
catch(exception1 e) {statements}
catch(exception2 e) {statements}
catch(exception3|exception4 e){statements}
```

Re-Throwing Exceptions

```
public class TestStack {
  private static void helper (ArrayStack<String> s) throws
                                         EmptyStackException {
      String line = s.pop();
      . . .
  }

  public static void main (String[] args) {
    ArrayStack<String> s = new ArrayStack<String>();
    String line;
    try {
       helper(s);
          . . .
    }
    catch (EmptyStackException e) {
       System.out.println ("Stack empty"+e.getMessage());
    }
    . . .
}
```

Method *helper* re-throws the exception thrown by pop()

If the EmptyStackException were not caught here, the program would crash

Java Exceptions

- In Java, an exception is an ***object***
- There are Java predefined ***exception classes,*** like
  - ArithmeticException
  - IndexOutOfBoundsException
  - IOException
  - NullPointerException

Some Java Error and Exception Classes

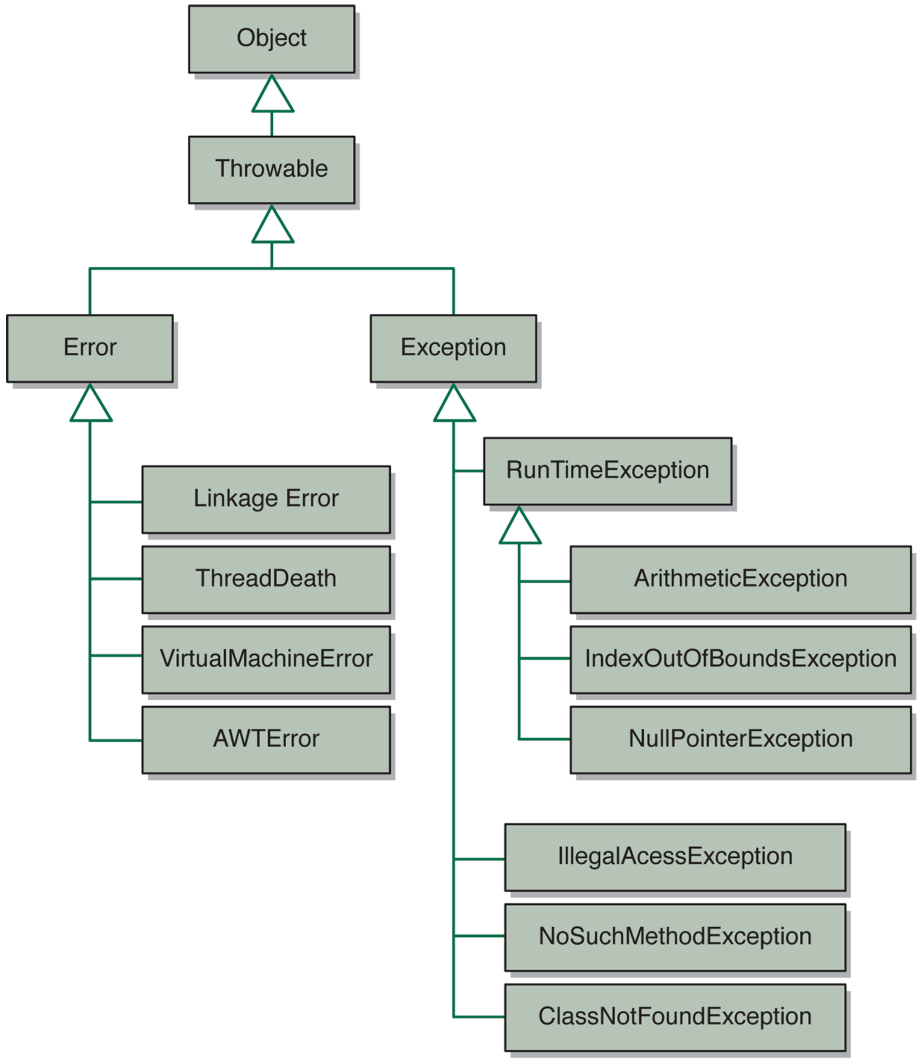


FIGURE 2.7 Part of the `Error` and `Exception` class hierarchy

Runtime Errors

- Java differentiates between ***runtime errors*** and ***exceptions***
  - Errors are unrecoverable situations, so the program must be terminated
    - Example: running out of memory

Exceptions

Exception: an abnormal or erroneous situation at runtime

Examples:

- Division by zero
- Array index out of bounds
- Null pointer exception

Exceptions can be thrown by the program or by the java virtual machine, for example for this statement

```
i = size / 0;
```

The virtual machine will throw an exception

Declaring Exception Classes

```
public class EmptyStackException extends
                                        RuntimeException {

   public EmptyStackException (String mssg) {
        super (mssg);
   }
}
```

```
public class ExceptionExample {
        private static int x = 1;
        private static String s = “”;
        public static void main(String[] args) {
                try {
                        method1(2);
                        method1(1);
                        x = x + 3;
                }
                catch (Exception1 e) {x = 0;}
                catch (Exception2 ex) {x = x + 5;}
                System.out.println(x+”, “+s);
        }
        private static void method1(int param) throws Exception1, Exception2 {
                try {
                        if (param == 1) method2(“hello”);
                        else method2(s);
                        ++x;
                }
                catch (Exception1 e) {
                        System.out.println(e.getMessage());
                        s = “hi”;
                }
        }
```

```
private static void method2(String str) {
        if (str.length() > 0) ++x;
        else
            throw new Exception1(“Empty string”);
        s = “hello”;
        throw new Exception2(“Long string”);
}
```

Execute by hand this code. What is printed by the program?

Checked and Unchecked Exceptions

- Checked exceptions are checked by the compiler
- Unchecked exceptions are not

Example: Checked Exception

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader(“test.txt");
            BufferedReader fileInput = new BufferedReader(file);
            System.out.println(fileInput.readLine());
            fileInput.close();
        }
        catch (FileNotFoundException e) { ... }
        catch (IOException e) { ... }
    }
}
```

Example: Checked Exception

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader(“test.txt");
        BufferedReader fileInput = new BufferedReader(file);
        System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```

The compiler gives the error:

```
Main.java:5: error: unreported exception FileNotFoundException; must be
caught or declared to be thrown
                    FileReader file = new FileReader(“test.txt”);
```

Example: UnChecked Exception

```
class Main {

    public static void main(String[] args) {
        int x = 10;
        int y = 0;
        int z = x /y;
    }
}
```

The compiler does not give an error even though we are dividing by zero.

A Try-Catch Example with Multiple Catch Statements and a *finally* Block

The try-catch-finally syntax:

```
try {
    code
}
catch(exception1 e) {statements}
catch(exception2 e) {statements}
catch(exception3|exception4 e){statements}
finally {statements}
```

Finally Block

The finally block always executes when the try block exits, whether an exception was thrown or not (even if the exception was not caught by any of the catch statements!)

The finally block is executed even if there is a return statement inside the try or catch blocks or if a new exception is thrown.

No *finally* Block

```
PrintWriter out;
try {
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    out.println(“Data”);
    int x = 5 / 0;
}
catch(FileNotFoundException e) {…}
catch(IOException e) {…}
if (out != null) out.close();
```

The exception caused by dividing 5 by 0 will not be caught, so the statement out.close() will not be executed, so the file will not be closed, and the data will not be stored in it.

Code with *finally* Block

```
PrintWriter out = null;
try {
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    out.println(“Data”);
    int x = 5 / 0;
}
catch(FileNotFoundException e) {…}
catch(IOException e) {…}
finally  {
  if (out != null) out.close();
}
```

The file will be closed.