Java By Abstraction: Chapter 11

Exception Handling

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

What are Exceptions?

There are three sources that can lead to exceptions:

The End User

Garbage-in, garbage-out

The Programmer

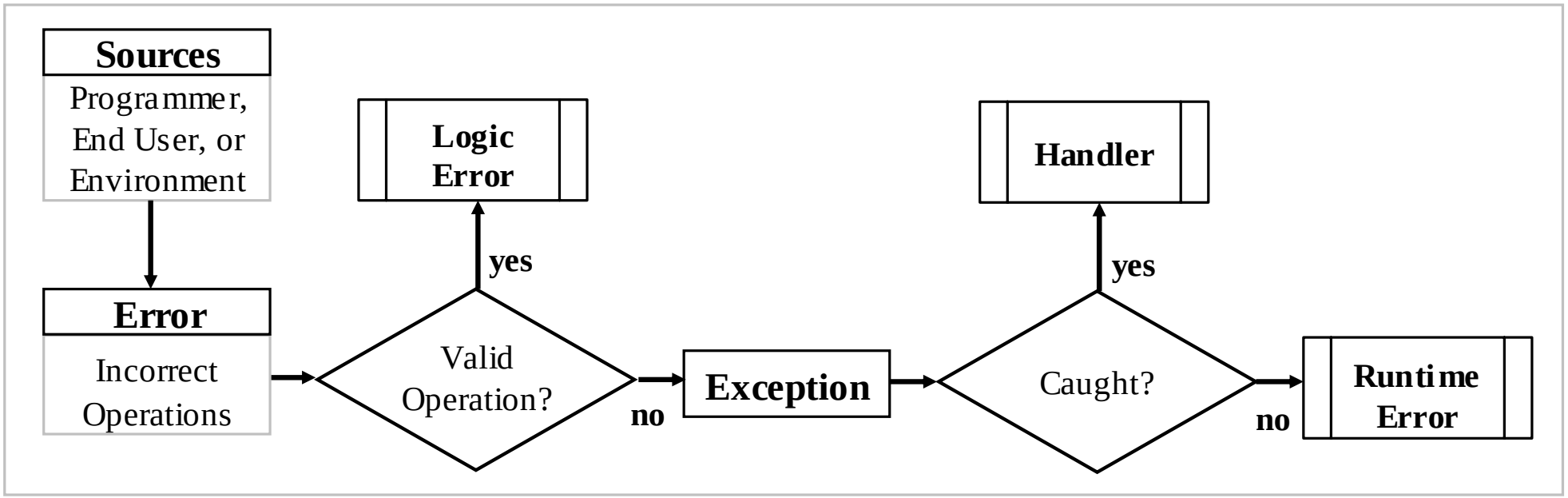
Misunderstanding requirements and/or contracts

The Environment

The VM, the O/S, the H/W, or the network

Exception Handling in General

- An error source can lead to an incorrect operation
- An incorrect operations may be valid or invalid
- An invalid operation throws an exception
- An exception becomes a runtime error unless caught



Example One

Given two integers, write a program to compute and output their quotient.

```
output.println("Enter the first integer:");
int a = input.nextInt();
output.println("Enter the second:");
int b = input.nextInt();

int c = a / b;
output.println("Their quotient is: " + c);
```

Example One

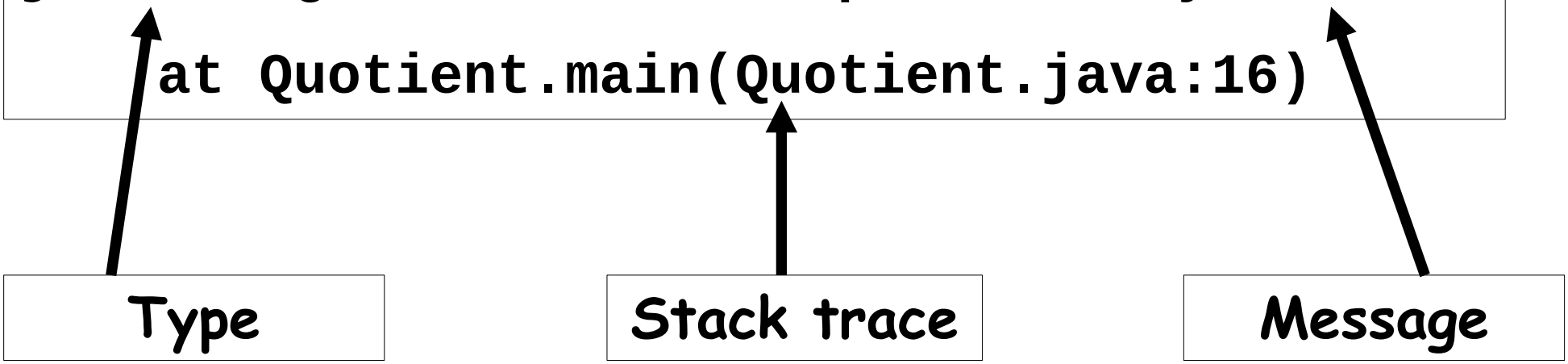
```
Enter the first integer:
8

Enter the second:
0

Exception in thread "main"
java.lang.ArithmeticException: / by zero

    at Quotient.main(Quotient.java:16)
```

In this case:

- The error source is the end user.
- The incorrect operation is invalid
- The exception was not caught

Anatomy of an Error Message

```
Enter the first integer:
8

Enter the second:
0

Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Quotient.main(Quotient.java:16)
```



The Delegation Model

- We, the client, delegate to method *A*
- Method *A* delegates to method B
- An invalid operation is encountered in B
- If B handled it, no one would know
- Not even the API of B would document this
- If B didn't, it delegates the exception back to *A*
- If *A* handled it, we wouldn't know
- Otherwise, the exception is delegated to us
- We too can either handle or delegate (to VM)
- If we don't handle, the VM causes a runtime error

The Delegation Model Policy

Handle or Delegate Back

Note:

- **Applies to all (components and client)**
- **The API must document any back delegation**
- **It does so under the heading: "Throws"**

Example Two, Part 1

Given a string containing two slash-delimited substrings, write a program that extracts and outputs the two substrings.

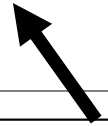
```
int slash = str.indexOf("/");
String left = str.substring(0, slash);
String right = str.substring(slash + 1);
output.println("Left substring: " + left);
output.println("Right substring: " + right);
```

Example Two, Part 1

Here is a sample run with str = "14-9"

```
int slash = str.indexOf("/");
String left = str.substring(0, slash);
String right = str.substring(slash + 1);
output.println("Left substring: " + left);
output.println("Right substring: " + right);
```

```
java.lang.IndexOutOfBoundsException:
String index out of range: -1
at java.lang.String.substring(String.java:1480)
at Substring.main(Substring.java:14)
```



The trace follows the delegation from line 1480 within substring to line 14 within the client.

Example Two, Part 1

Here is the API of substring:

```
String substring(int beginIndex, int endIndex)
Returns a new string that…

Parameters:
beginIndex - the beginning index, inclusive.
endIndex - the ending index, exclusive.

Returns:
the specified substring.

Throws:
IndexOutOfBoundsException - if the beginIndex
is negative, or endIndex is larger than the
length of this String object, or beginIndex is
larger than endIndex.
```

Try-Catch Block

```
try
{  ...
   code fragment
   ...
}
catch (SomeType e)
{  ...
   exception handler
   ...
}
program continues
```

Try-Catch Block Example

Redo the last example with exception handling

```
try
{
   int slash = str.indexOf("/");
   String left = str.substring(0, slash);
   String right = str.substring(slash + 1);
   output.println("Left substring: " + left);
   output.println("Right substring: " + right);
}
catch (IndexOutOfBoundsException e)
{
   output.println("No slash in input!");
}
output.println("Clean Exit."); // closing
```

Try-Catch with Multiple Exceptions

```
try
{  ...
}
catch (Type-1 e)
{  ...
}
catch (Type-2 e)
{  ...
}
...
catch (Type-n e)
{  ...
}
program continues
```

Example Two, Part 2

Given a string containing two slash-delimited integers, write a program that outputs their quotient. Use exception handling to handle all possible input errors.

Note that when exception handling is used, do not code defensively; i.e. assume the world is perfect and then worry about problems. This separates the program logic from validation.

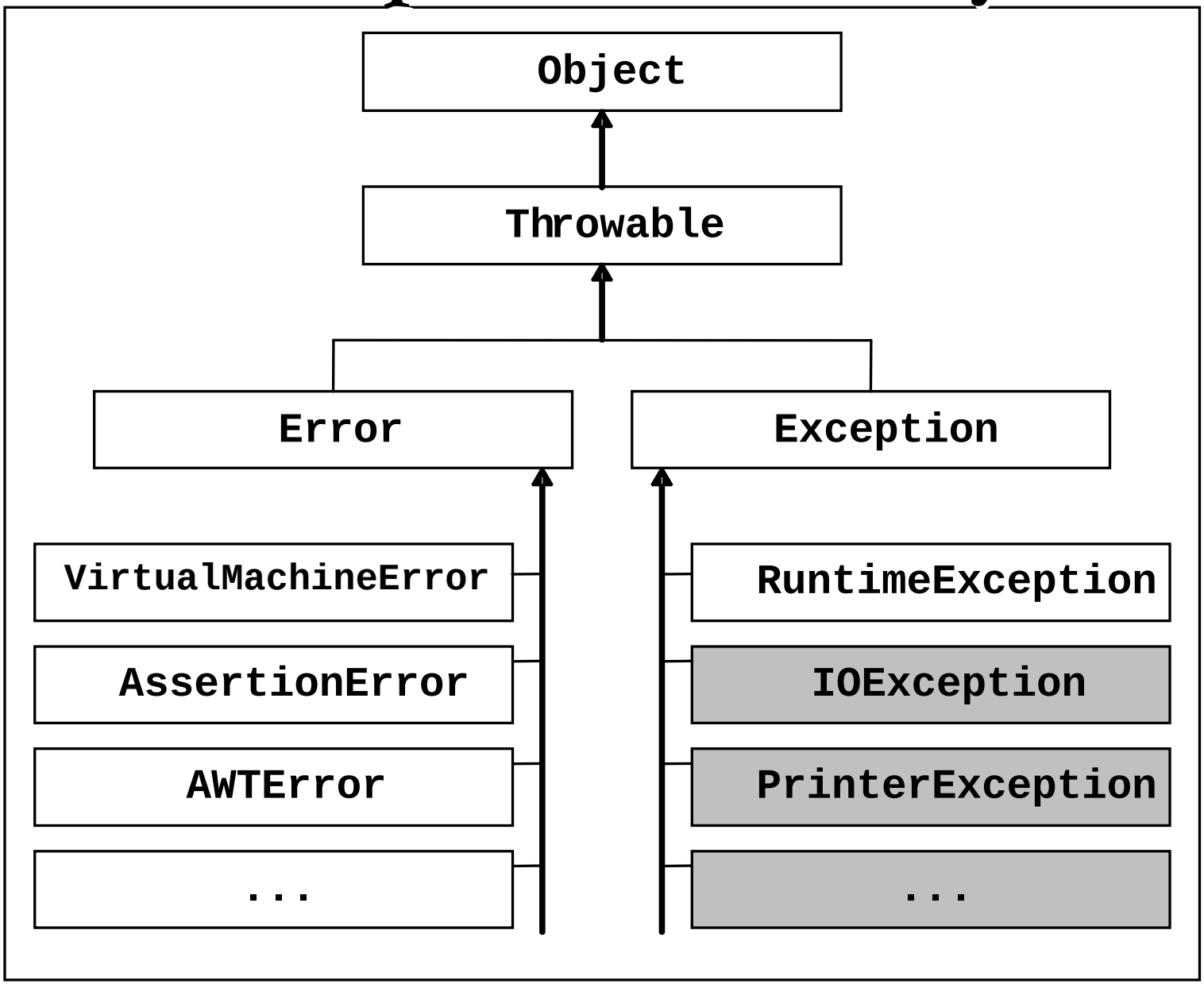
Example Two, Part 2

```
try
{
   int slash = str.indexOf("/");
   String left = str.substring(0, slash);
   String right = str.substring(slash + 1);
   int leftInt = Integer.parseInt(left);
   int rightInt = Integer.parseInt(right);
   int answer = leftInt / rightInt;
   output.println("Quotient = " + answer);
}
catch (?)
{

}
```

Example Two, Part 2

```
catch (IndexOutOfBoundsException e)
{
   output.println("No slash in input!");
}
catch (NumberFormatException e)
{
   output.println("Non-integer operands!");
}
catch (ArithmeticException e)
{
   output.println("Cannot divide by zero!");
}

output.println("Clean Exit."); // closing
```

Exception Hierarchy



Exception Handling

- They all inherit the features in Throwable
- Can create them like any other object:
  Exception e = new Exception();
- And can invoke methods on them, e.g.
  getMessage, printStackTrace, etc.
- They all have a toString
- Creating one does not simulate an exception. For that, use the throw keyword:

```
Exception e = new Exception("test");
throw e;
```

Example Two, Part 3

Write an app that reads a string containing two slash-delimited integers, the first of which is positive, and outputs their quotient using exception handling. Allow the user to retry indefinitely if an input is found invalid.

As before but:

- **What if the first integer is not positive?**
- **How do you allow retrying?**

Example Two, Part 3

```
for (boolean stay = true; stay;)
{
   try
   {
      // as before
      if (leftInt < 0) throw(??);
      …
      output.println("Quotient = " + answer);
      stay = false;
   }
   // several catch blocks
}
```

Checked Exceptions

- App programmers can avoid any RuntimeException through defensive validation
- Hence, we cannot force them to handle such exceptions
- Other exceptions, however, are "un-validatable", e.g. diskette not inserted; network not available…
- These are "checked" exceptions
- App programmers *must* acknowledge their existence
- How do we enforce that?
- The compiler ensures that the app either handles checked exceptions or use "throws" in its `main`.