Exception

"A run-time event that disrupts the normal flow of control."

Robustness (cf. *Correctness*)
Portability

- Compiler checks *syntax*, flags *type mismatches*, *uninitialized variables*, etc.
- ***Exception mechanism*** enables *flagging* and *graceful handling* of *violation of semantic constraints* of the *language* or of the *application* at run-time. E.g.,
  - Array index out of range, Storage error, etc.
  - Physical system problems such as *High Temperature*, *Disk crash*, etc.
  - Programming errors such as *Identifier* not found in *Table* etc.

Exception *vs* Error Code

Returning ***error code*** to signal abnormal condition is unsuitable because:

- Error handling code and normal processing code mixed up.
- A "caller" can *ignore* ***it***.
  - Not conducive to portability and robustness.
  - Difficult to pinpoint source/location of error from results that are only indirectly related to it.
- For a sequence of nested calls, the intermediate procedures must explicitly check and propagate ***it***.
  - Using *global variable* or *goto*s is not appropriate either.

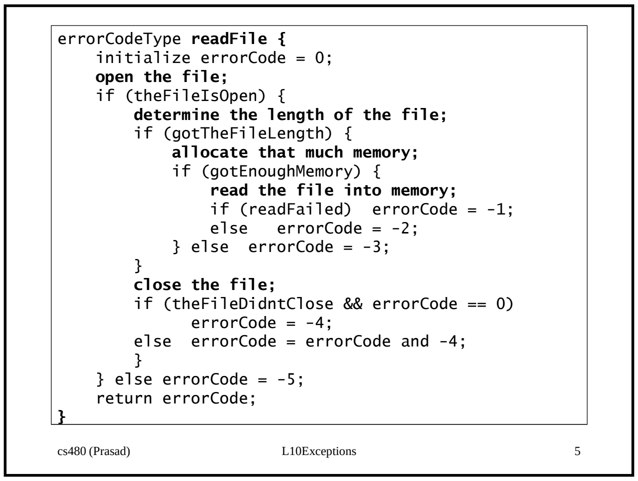
Advantage 1: Separating Error Handling Code from "Regular" Code

CODE

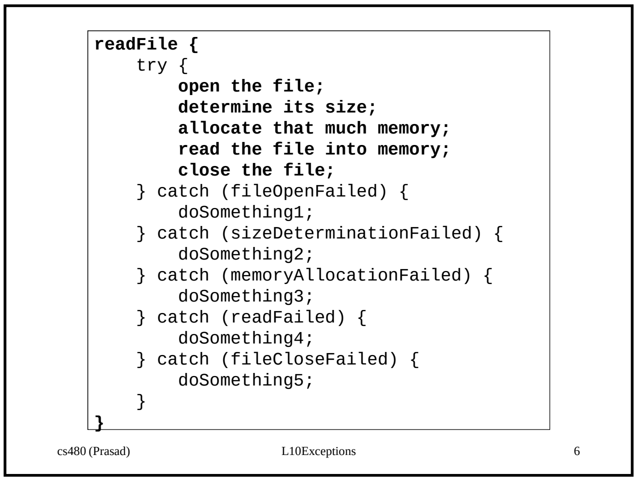
```
readFile {
 open the file;
 find its size;
 allocate   memory;
 read file into memory;
 close the file;
}
```

ABNORMALITIES

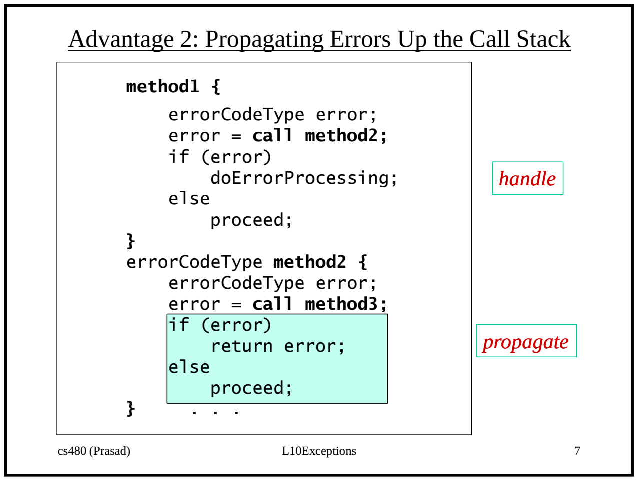
- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?



```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed)  errorCode = -1;
                else   errorCode = -2;
            } else  errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0)
              errorCode = -4;
        else  errorCode = errorCode and -4;
        }
    } else errorCode = -5;
    return errorCode;
}
```



```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething1;
    } catch (sizeDeterminationFailed) {
        doSomething2;
    } catch (memoryAllocationFailed) {
        doSomething3;
    } catch (readFailed) {
        doSomething4;
    } catch (fileCloseFailed) {
        doSomething5;
    }
}
```

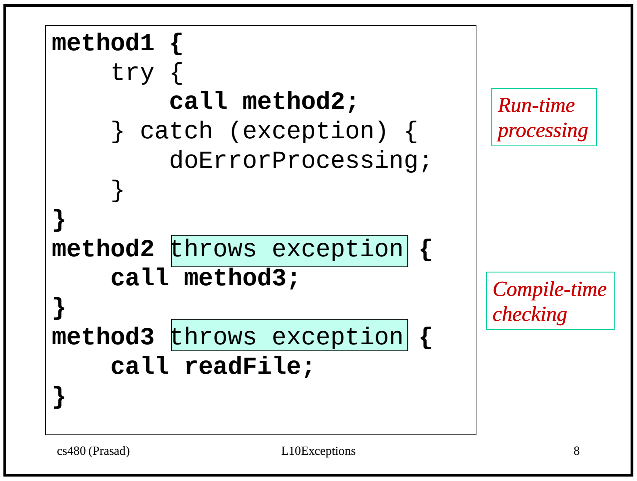


Advantage 2: Propagating Errors Up the Call Stack

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}    . . .
```

handle

propagate



```
method1 {
    try {
        call method2;
    } catch (exception) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```

Run-time processing

Compile-time checking

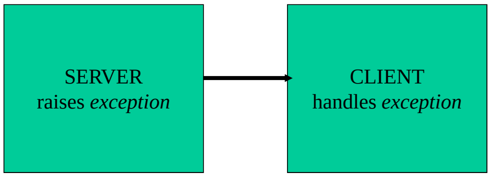
Construct

```
    try {                 ...
         throw  new  Exception("test");
             ...
    } catch (InterruptedException e)  {
             ...
    } catch   (Exception e)  {
             ...
    } finally {     ...
    }
```

◆ In Java, exceptions are “object-oriented”.

- A generated exception is an instance of a class. It typically holds information about the *point of creation* and the *reason for creation.*
- Exceptions are grouped and well-integrated into the class subclass hierarchy. The type (class) of an exception is used to determine the handler.
- A `catch`-statement associates an exception handler with an exception object.
- As `catch`-statements are processed sequentially, they are ordered by “specificity”. That is, a handler for an instance of an exception class must always *precede* an handler for an instance of its superclass, to get the desired behavior.

Modularity and Exception



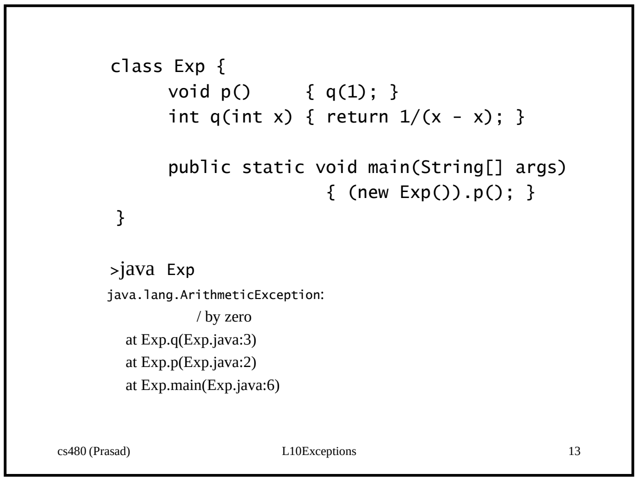
SERVER
raises *exception*

CLIENT
handles *exception*

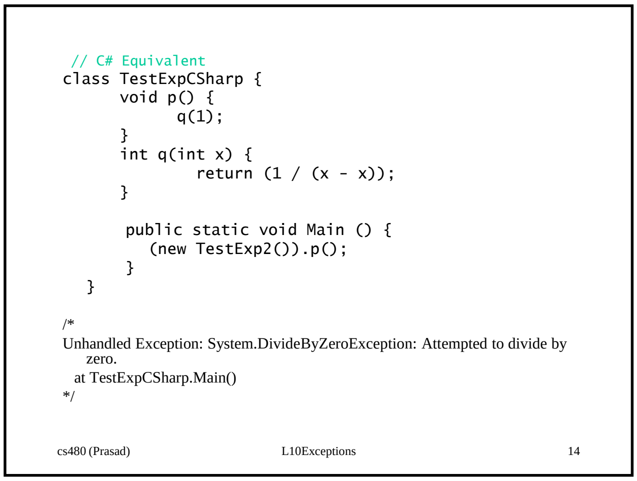
Context-sensitive Handling (*client-server*)

E.g., “*Identifier not found in table*” can elicit any one of the following actions depending on the context of use:

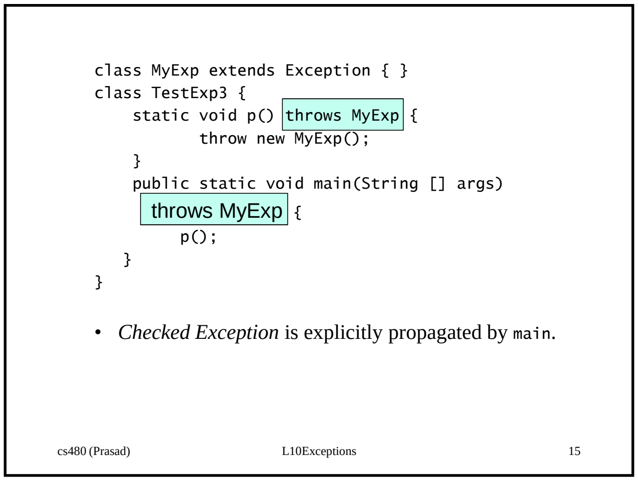
- Return a default value.
- Undefined/unbound variable error.
- Add the identifier to the table.
- Prompt user to reenter identifier.



```
class Exp {
      void p()     { q(1); }
      int q(int x) { return 1/(x - x); }

      public static void main(String[] args)
                      { (new Exp()).p(); }
 }

>java Exp
java.lang.ArithmeticException:
               / by zero
   at Exp.q(Exp.java:3)
   at Exp.p(Exp.java:2)
   at Exp.main(Exp.java:6)
```

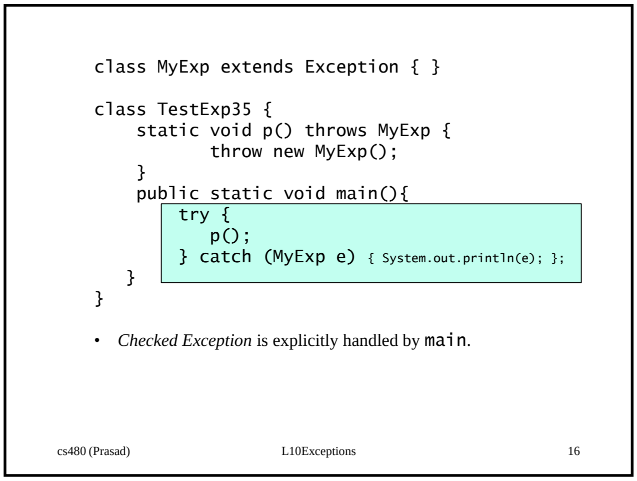


```
 // C# Equivalent
class TestExpCSharp {
      void p() {
            q(1);
      }
      int q(int x) {
              return (1 / (x - x));
      }

       public static void Main () {
          (new TestExp2()).p();
       }
   }

/*
Unhandled Exception: System.DivideByZeroException: Attempted to divide by
    zero.
  at TestExpCSharp.Main()
*/
```



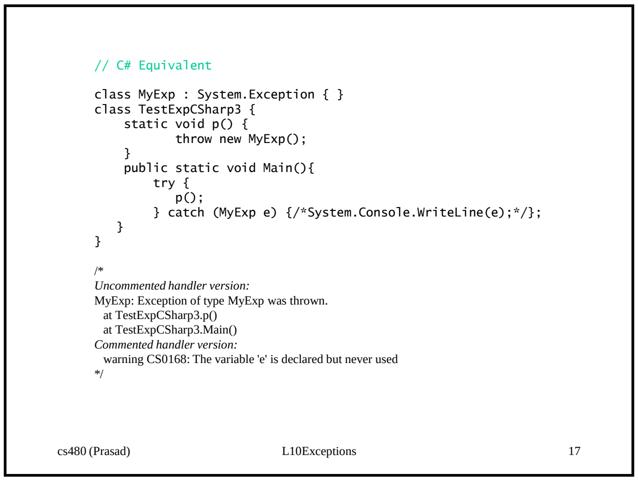
```
class MyExp extends Exception { }
class TestExp3 {
    static void p() throws MyExp {
           throw new MyExp();
    }
    public static void main(String [] args)
     throws MyExp {
         p();
   }
}
```

- *Checked Exception* is explicitly propagated by `main`.



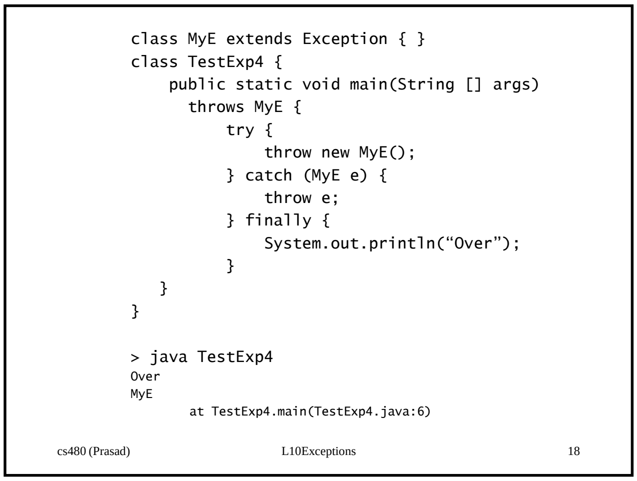
```
class MyExp extends Exception { }

class TestExp35 {
    static void p() throws MyExp {
           throw new MyExp();
    }
    public static void main(){
        try {
            p();
        } catch (MyExp e) { System.out.println(e); };
    }
}
```

- *Checked Exception* is explicitly handled by `main`.

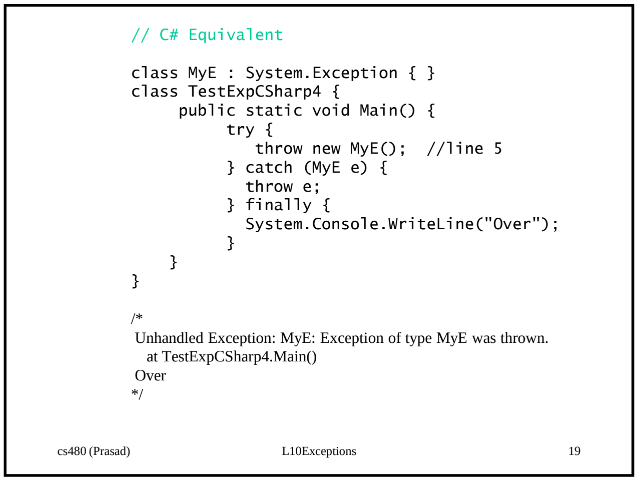


```
// C# Equivalent

class MyExp : System.Exception { }
class TestExpCSharp3 {
    static void p() {
           throw new MyExp();
    }
    public static void Main(){
        try {
           p();
        } catch (MyExp e) {/*System.Console.WriteLine(e);*/};
   }
}
```

/*
Uncommented handler version:
MyExp: Exception of type MyExp was thrown.
  at TestExpCSharp3.p()
  at TestExpCSharp3.Main()
Commented handler version:
  warning CS0168: The variable 'e' is declared but never used
*/

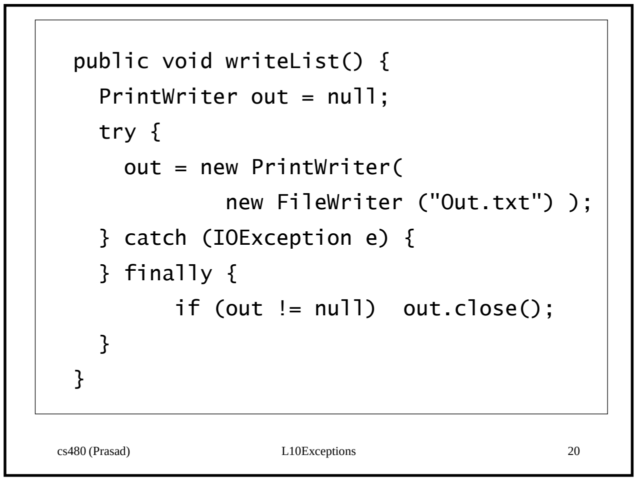


```
class MyE extends Exception { }
class TestExp4 {
     public static void main(String [] args)
       throws MyE {
          try {
              throw new MyE();
          } catch (MyE e) {
              throw e;
          } finally {
              System.out.println(“Over”);
          }
    }
}

> java TestExp4
Over
MyE
        at TestExp4.main(TestExp4.java:6)
```



```
// C# Equivalent

class MyE : System.Exception { }
class TestExpCSharp4 {
     public static void Main() {
          try {
             throw new MyE();  //line 5
          } catch (MyE e) {
            throw e;
          } finally {
            System.Console.WriteLine("Over");
          }
     }
}
```

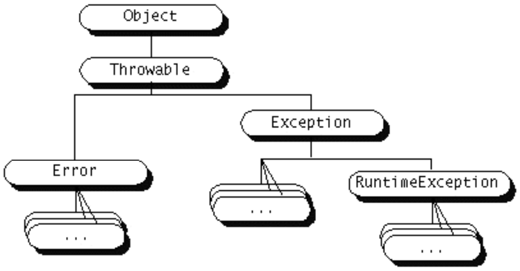
/*
Unhandled Exception: MyE: Exception of type MyE was thrown.
  at TestExpCSharp4.Main()
Over
*/



```
public void writeList() {
  PrintWriter out = null;
  try {
    out = new PrintWriter(
            new FileWriter ("Out.txt") );
  } catch (IOException e) {
  } finally {
        if (out != null)  out.close();
  }
}
```

Subclasses of `Throwable`

- `Error`
  - *unchecked*: recovery difficult or impossible
    - E.g., `OutOfMemoryError`, etc.
- `Exception` (language/user-defined)
  - *unchecked*: too cumbersome for programmer to declare/handle.
    - E.g., `NullPointerException`, `ClassCastException`, etc.
  - *checked* : requires programmer to provide an handler or propagate exception ***explicitly***.
    - E.g., `java.io.IOException`, etc.


Object
Throwable
Error
Exception
RuntimeException
...
...
...