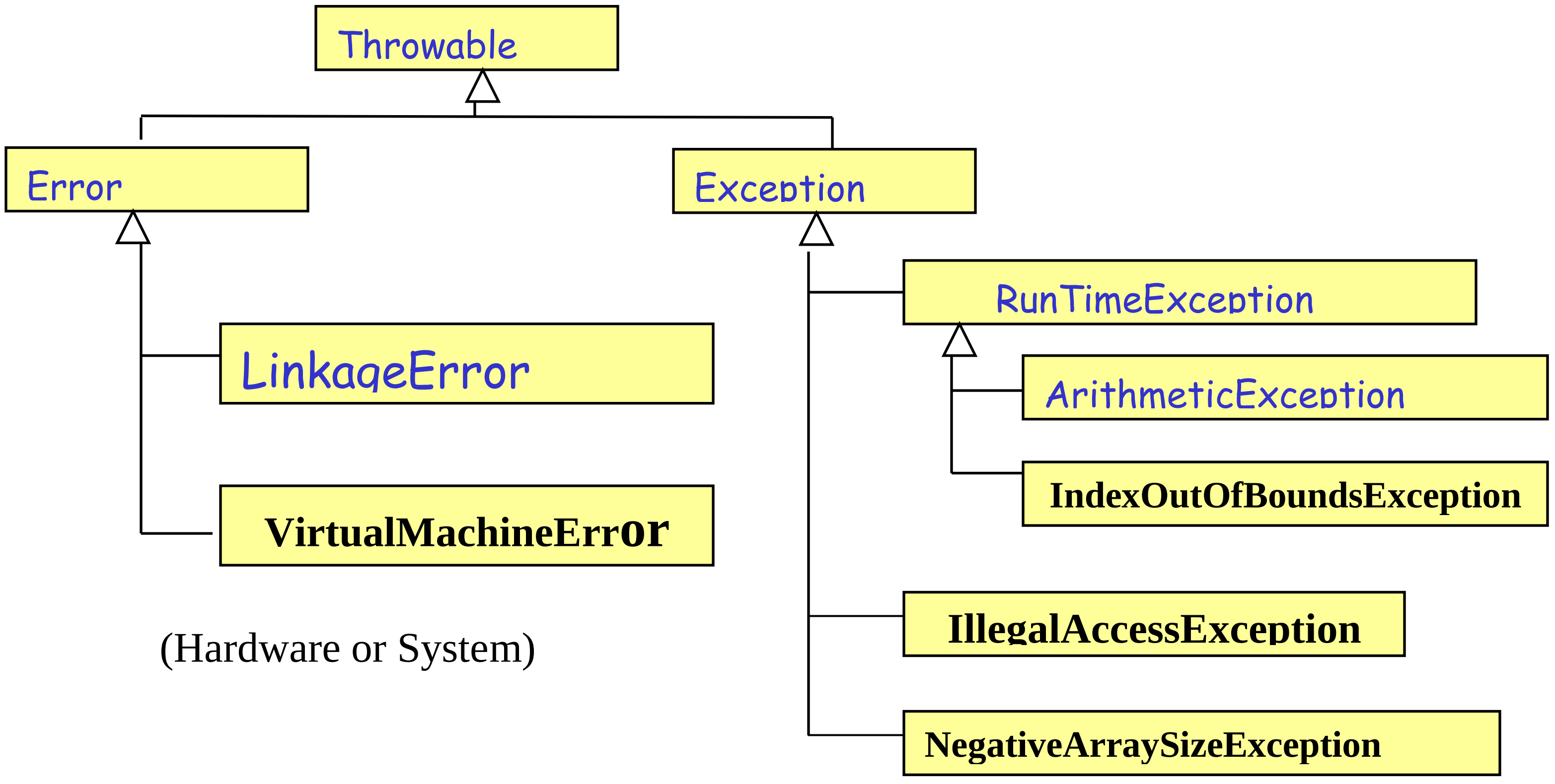
Exception Handling in Java

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions

Exceptions are represented as classes in Java.



The `try` and `catch` Statement

To process an exception when it occurs, the line that throws the exception is executed within a *try block*

A try block is followed by one or more *catch* clauses, which contain code to process an exception

Each catch clause has an associated exception type. When an exception occurs, processing continues at the first catch clause that matches the exception type

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}

// etc...
```

The `finally` block

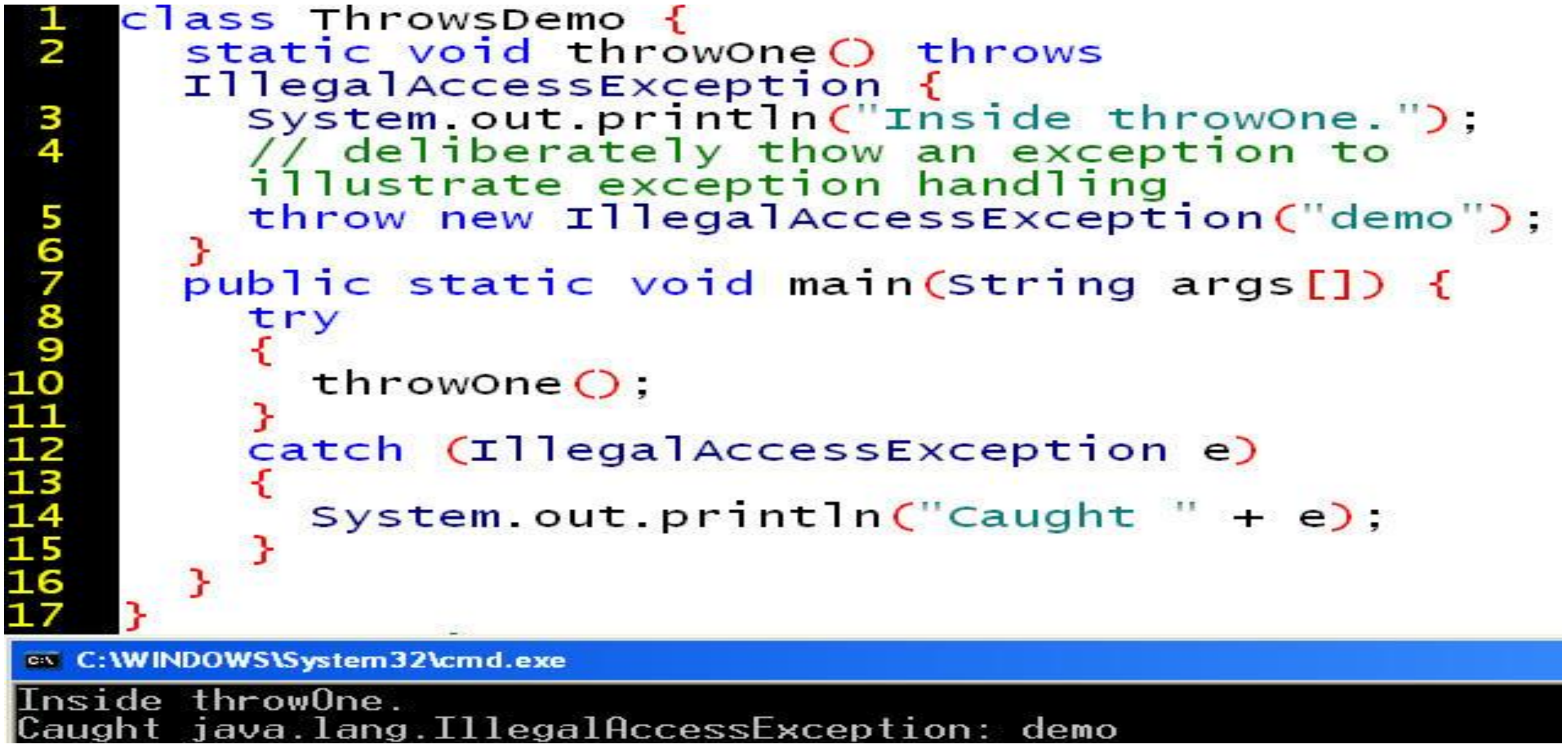
A finally block is where you put code that must run regardless of an exception.

```
try {
  // Dangerous activities that might throw A or B }
catch(A a1) {
  // Handler for situation A}
catch(B b1) {
  // Handler for situation B}
finally {
  // Activities that happen every time
}
```

The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to insure that resource is *always* recovered.

Declaring that a method throws exceptions

- The keyword **throws** indicates that a method can throw an exception that it does not handle. This declaration does permit another method to handle the exception.
- All Java methods use the **throw** statement to throw an exception. The throw statement requires a single argument: a *throwable* object.



```
class ThrowsDemo {
  static void throwOne() throws
  IllegalAccessException {
    System.out.println("Inside throwOne.");
    // deliberately thow an exception to
    illustrate exception handling
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try
    {
      throwOne();
    }
    catch (IllegalAccessException e)
    {
      System.out.println("Caught " + e);
    }
  }
}
```

C:\WINDOWS\System32\cmd.exe

```
Inside throwOne.
Caught java.lang.IllegalAccessException: demo
```

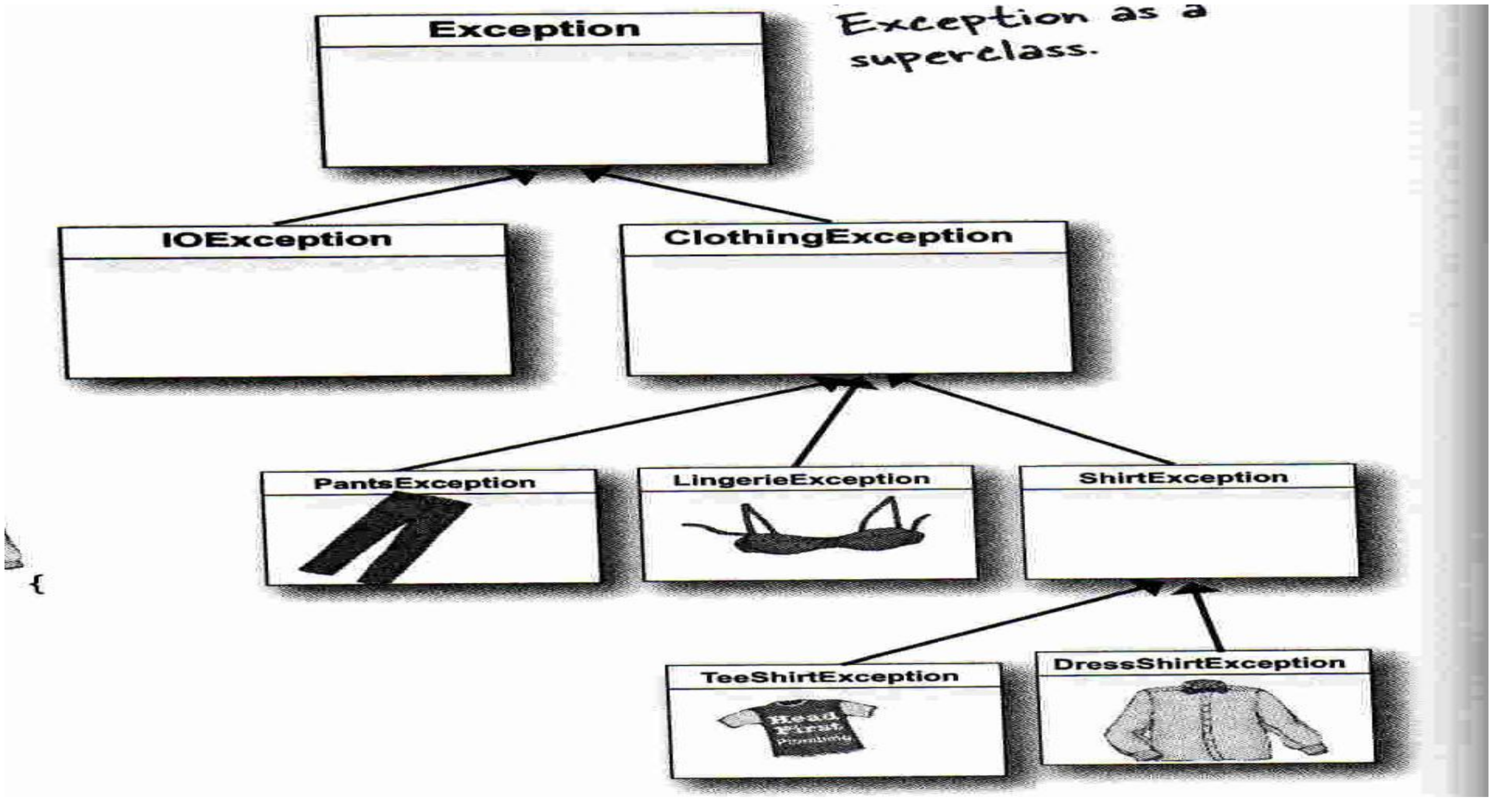
A method can throw more than one exception.

```
public class Laundry {
    public void doLaundry() throws PantsException, LingerieException {
        // code that could throw either exception
      }
   }

 public class Foo {
       public void go() {
           Laundry laundry = new Laundry ();
       try {
               laundry.doLaundry();

          } catch(PantsException pex) {
                   // recovery code

         } catch(LingerieException lex) {
                   // recovery code
           }
  }
```

Exceptions are polymorphic

You can declare exceptions using a supertype of the exceptions you throw.

```
public void doLaundry() throws ClothingException //ClosingException lets you throw
                                                 //any subclass of ClothException.
```

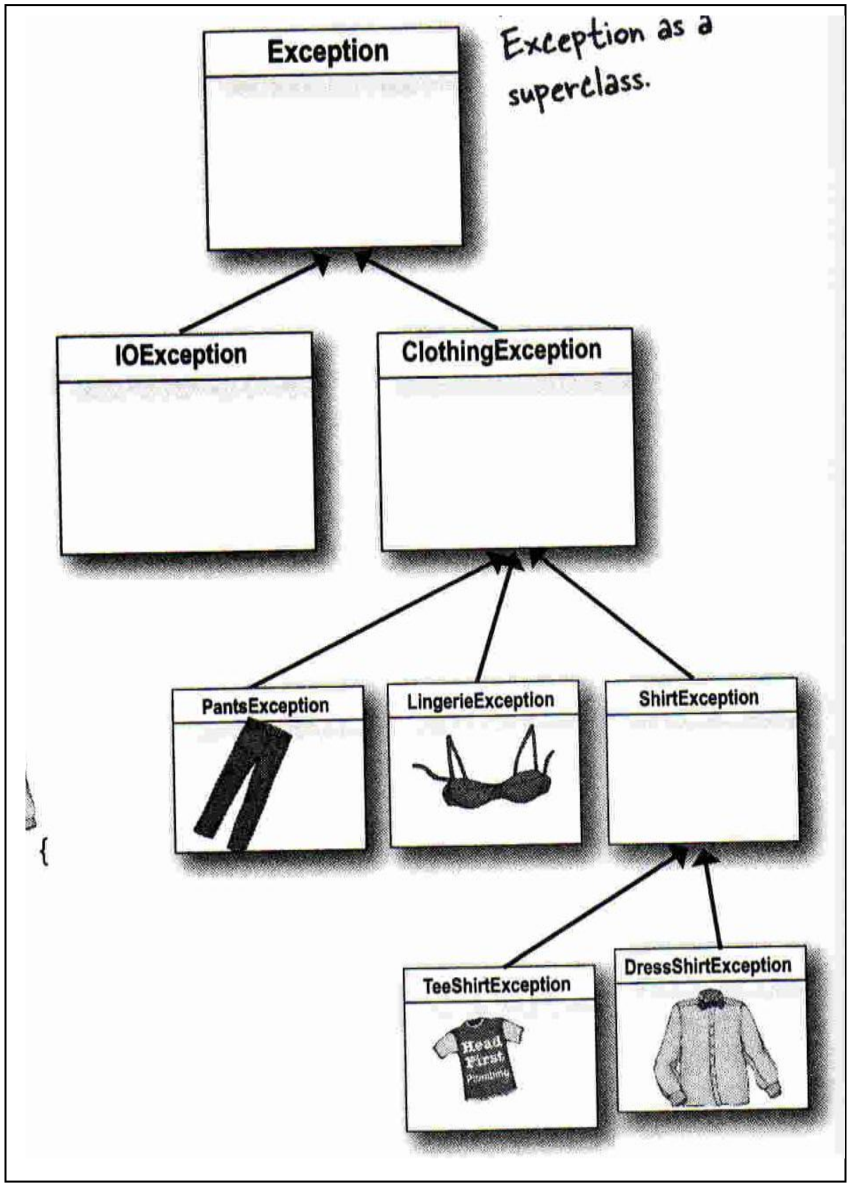


You can Catch exceptions using a supertype of the exception thrown.

```
try {
            laundry.doLaundry();

} catch (ClothingException cex) {
```

Can catch any ClothingException subclass

```
try {
            laundry.doLaundry();

} catch (ShirtException cex) {
```



Exception: Handle it in another place in the program

If it is not appropriate to handle the exception where it occurs, it can be handled at a higher level

Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the outermost level.

Inner try blocks

You can have inner try blocks for specialized exceptions, and reserve the outmost try block for the exception handling to use if all the previous attempts fail.

```
class NestedTryBlocks{
   public static void main(String args[])    {
      int numer[] = {4, 8, 16, 32, 64, 128, 256, 512};
      int denom[] = {2, 0, 4,    4,   0,     8};
        try{
        for (int i=0; i<numer.length; i++) {
            try {
               System.out.println(numer[i] + "/"+ denom[i] + "is" +   numer[i]/denom[i]);
            }
            catch (ArithmeticException exc)
            {
               System.out.println(" cannot divide  by zero");
            }
          }
        }
        catch (ArrayIndexOutOfBoundsException exc)
        {
           System.out.println (" No matching element found");
        }
      }
   }
```

```
4/2is2
 cannot divide  by zero
16/4is4
32/4is8
 cannot divide  by zero
128/8is16
 No matching element found
```

Rethrowing an exception

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the enclosing try block have an opportunity to catch the exception.

```
catch(Exception e) {
  System.out.println("An exception was thrown");
  throw e;}
```

```
public class Rethrowing {
  public static void f() throws Exception {
      System.out.println("originating the exception in f()");
     throw new Exception("thrown from f()");
  }

 public static void g() throws Throwable {
   try {
       f();
   } catch(Exception e) {
       System.out.println("Inside g()”, e");
       throw e;
   }
  }

public static void main(String[] args) throws Throwable {
   try {
      g();
   } catch(Exception e) {
        System.out.println("Caught in main");
   }
  }
}
```

```
originating the exception in f()
Inside g()”, e
Caught in main
```

Consider the following Java code:

```
try {
         System . out . println (1);
      try {
                  System . out . println (2);
                  f ();
                  System . out . println (3);}
         catch (E1 e1) {
                  System . out. println (4);}
         finally {
                  System . out. println (5);}
}
catch (E2 e2) {
         System . out . println (6);}
```

Indicate the output of this code for the cases where (a) there is no exception, (b) f throws exception E1, (c) f throws exception E2, and (d) f throws exception E3. (Note: E3 is not a subclass of either E1 and E2.)

Answer:
(a) 1, 2, 3, 5
(b) 1, 2, 4, 5
(c) 1, 2, 5, 6
(d) 1, 2, 5, Exception

Why Finally?

A Finally block will be executed after a try block if no exception has been thrown or after a catch if an exception was thrown. This means that a finally block can be used as 'clean up' - a place to close files or connections etc, whether an exception is thrown or not.

- If a try block throws an exception and the catch block propagates the exception (throws it again), the finally clause will still execute.

- **If the finally clause executes a return statement, it overides a thrown exception** (so the exception will not be thrown; instead the return will occur).

```
class ExQ {
  public static void main(String[] args) {
          try {
                  System.out.println("Start");
                  ExQ x = new ExQ();
                  x.aMethod();
                  System.out.println("After method");}
        catch (Exception error) {
                  System.out.println("main’s catch");}
        finally {
                  System.out.println("main’s finally");}
                  System.out.println("End");
  }

  public void aMethod() throws Exception {
        try {
                  System.out.println("In aMethod");
                  throw new Exception();}
        catch (Exception error ) {
                  System.out.println("aMethod’s catch");
                  throw new Exception();}
        finally {
                  System.out.println("aMethod’s finally");
        }
  }
}
```

```
Start
In aMethod
aMethod’s catch
aMethod’s finally
main’s catch
main’s finally
End
```

What will be the output of compiling and executing the main method of the following class?

```
class ExQ {
  public static void main(String[] args) {
        try {
              System.out.println("Start");
              ExQ x = new ExQ();
              x.aMethod();
              System.out.println("After method");}
        catch (Exception error) {
              System.out.println("main’s catch");}
        finally {
              System.out.println("main’s finally");}
              System.out.println("End");}

  public void aMethod() throws Exception {
        try {
              System.out.println("In aMethod");
              throw new Exception();}
        catch (Exception error ) {
              System.out.println("aMethod’s catch");
              throw new Exception();}
        finally {
              System.out.println("aMethod’s finally");
              return;
              }}}
```

```
Start
In aMethod
aMethod’s catch
aMethod’s finally
After method
main’s finally
End
```

Exception Handling in C++

• Added to C++ in 1990
• Design is based on that of Ada, and ML

C++ Exception Handlers

```
try {
      throw an actual parameter
}
catch (formal parameter) {
      -- handler code   }
catch (formal parameter) {
      -- handler code
}
```

- A throw expression accepts one parameter which is passed as an argument to the exception handler.
- The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is catch.

Ex1.

```
int main () {
    try  {
         throw 20;  }
    catch (int e)  {
          cout << "An exception occurred. Exception Nr. " << e << endl;
    }
    return 0;
}
```

Ex2.

```
int main () {
    char myarray[10];
    try  {
          for (int n=0; n<=10; n++)    {
          if (n>9) throw "Out of range";
                  myarray[n]='z';    }
          }
     catch (char * str)  {
          cout << "Exception: " << str << endl;
    }
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Xtest(int test) {
    cout << "Inside Xtest, test is: " << test << \n";
    if (test) throw test;
}

int main( ) {
try { // start a try block
    Xtest(0);
    Xtest(1);
    Xtest(2);
}
catch (int i) { // catch an error
    cout << "Caught one! Number is: ";
    cout << i << "\n";
}
cout << "end";
return 0; }
```

This program displays:

```
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught one! Number is: 1
end
```

A **try** block could be in a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset.

```
void Xhandler(int test) {
  try {
      if (test) throw test;}
  catch(int i) {
      cout << "Caught one! Ex. #: " << i << "\n";
}}

int main( ) {
    cout << "start";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout << "end";
    return 0; }
```

This program displays:

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught one! Ex. #: 3
End
```

Throwing Objects

```
class CDtorDemo {
   public:
      CDtorDemo(){
           cout << "Constructing CDtorDemo.\n"; }
      ~CDtorDemo(){
           cout << "Destructing CDtorDemo.\n"; }

      void MyFunc() {
           CDtorDemo D;
             cout<< "In MyFunc(). Throwing CTest exception.\n";
             throw CTest(); }

int main() {
     try {
        cout << "In try block, calling MyFunc().\n";
        MyFunc(); }
     catch( CTest E ) {
         cout << "In catch handler.\n";
         cout << "Caught CTest exception type: ";
     }
      return 0; }
```