Exceptions

Objectives

At the end of this module you should be able to:

- Describe exceptions & understand their importance
- Describe the Java exception hierarchy
- Declare method signatures with **`throws`**
- Define an application exception hierarchy
- Use the **`try-throw-catch`** construct
- Use nested **`try`** blocks
- Use the **`finally`** clause
- Understand rethrowing exceptions

Exceptions

- Java incorporates an exception-handling mechanism into the language structure.
- Exceptions are objects that represent what went wrong.
  - Could be an exceptional case
  - Could be the expected negative result of a behavior
  - Could be the unexpected negative result of a behavior
  - If handled properly, many are recoverable
- Exceptions are standard Java objects with a specific type of hierarchy

Exceptions (cont.)

- An exception is not synonymous with a bug
  - Programming faults (bugs)
  - System faults like a down network (not a bug)
- Exceptions can be managed, which means either:
  - Code responds to an exception so a problem can be fixed and then processing can continue
  - Shutting the application down gracefully in order to do as little damage as possible

Reporting A Problem

- Try to perform the interaction

```
result = getResult();
```

- The interaction is determined to be a failure

```
 if(result != expectedResult) {
```

- An exception object is thrown to describe the failure

```
throw new DidntWorkException(“It Broke”);
```

- Exceptions are handled “further up”
- Note: always use `throw new XyzException()`
  - Stack trace information is prepared including the line number where `new` is executed

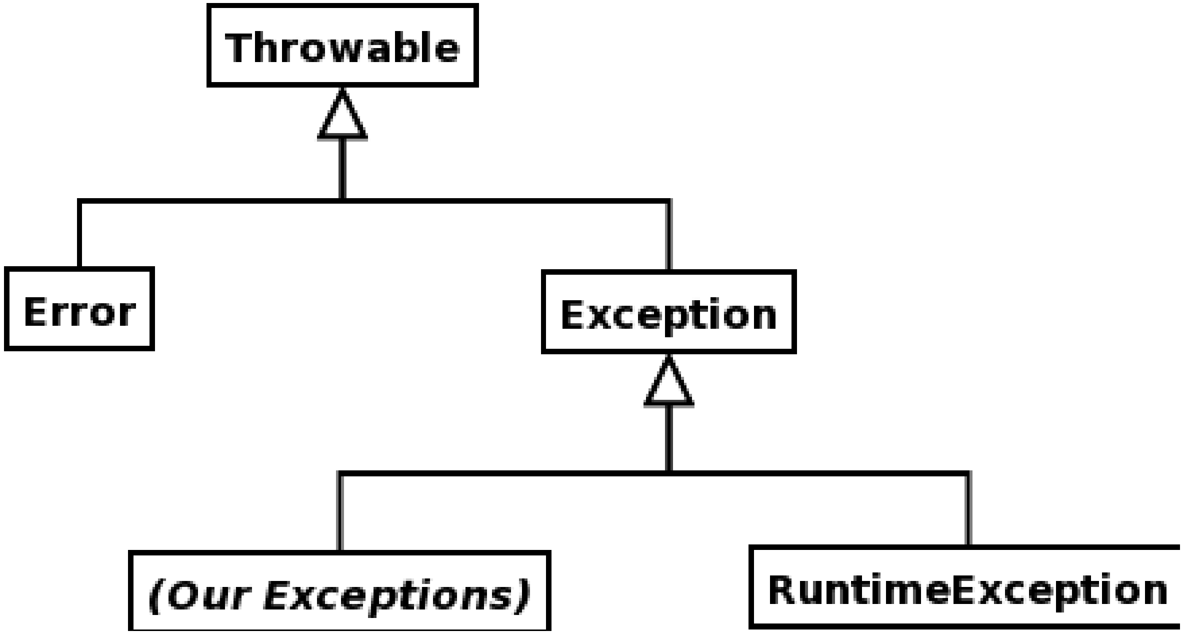
Exception Classification

- All exceptions are Java objects
- Exceptions are specific types of Java objects
  - Subclasses of **`java.lang.Throwable`**
    - Typically you won't work directly with **`Throwable`**
  - Will cause execution flow to be redirected
- Two subclasses of **`Throwable`**:
  - Error—environmental issue; probably won't recover from
    - `OutOfMemoryError, StackOverflowError`
  - Exception—programming/environmental issue; might try to recover
    - `NullPointerException, IOException`

Exception Classification (cont.)

- There are two types of Exceptions
- Checked—direct subclasses of `Exception`
  - Compiler requires this be handled in code
  - Typically *recoverable* application-level issues
    - E.g. `FileNotFoundException`
- Unchecked—`RuntimeException`
  - Not checked by compiler (hence “unchecked”)
  - Typically programming bugs “that shouldn’t happen”
    - `NullPointerException`
    - `ArraryIndexOutOfBoundsException`
  - Don’t try to fix at runtime, fix the bug!

Classification of Exceptions in Java



Java Exception Hierarchy

Exception Handling, Option 1

```
try {
  //delicate code
} catch (ExceptionType e) {
  //recovery
} finally {
  //final clean up
}
```

- **`try`**
  - Contains code that might fail
  - Flow control jumps from **`try`** to **`catch`** if an exception occurs
- **`catch`**
  - Contains the handling/recovery code
  - Executed only if a detected exception occurs
- **`finally`**
  - Always executed--use for final clean up
  - Have one **`finally`** block per **`try`**

Exception Handling 1 (cont.)

- It is permitted to have multiple catch blocks
- When designing multiple exception handlers consider
  - Exceptions that might arise
  - Class hierarchy of those exceptions
- These govern the order of the catch blocks

Specific

↓

Generic

```
try {
  //some network code
} catch (IOException ioe) {
  //do some IO recovery
} catch (Exception e) {
  //do some generic recovery
} finally {
  //do clean up
}
```

try-throw-catch Example

```
public class ExceptionsExample {
   public static void main(String[] args) {
     ExceptionsExample testObj = new ExceptionsExample();
     testObj.exec(args[0]);
   }
   public void exec(String option) {
     try {
      if (option.equals("fail")) {
        throw new Exception();
       }
       if (option.equals("access")) {
         throw new IllegalAccessException();
       }
       System.out.println("No Exception Thrown");
     } catch (IllegalAccessException e) {
       System.err.println("IOExcepton caught");
     } catch (Exception e) {
       System.err.println("Exception caught");
     }
   }
 }
```

Exception Handling, Option 2

- Sometimes this method cannot handle the problem.
- So, the method is quit, and the exception is passed to the caller.
- For *checked exceptions*, the method must declare this possibility.

```
public void mightBreak() throws BrokenException {
    // do stuff
    if (itBroke) {
      throw new BrokenException(“it Broke”);
    }
    // rest of method
```

Implementing an Exception Hierarchy

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class BankExceptions {
  public static void main(String[] args) {
    // here is the try block
    try {
      throw new ATMException();
    } catch (ATMException e) {
      System.err.println("Caught ATMException");
    } catch (BankException e) {
      System.err.println("Caught BankException");
    }
  }
}
// Output is: Caught ATMException
```

Implementing an Exception Hierarchy (cont.)

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class BankExceptions2 {
  public static void main(String [] args) {
    try {
      throw new ATMException ();
    } catch(BankException e) {
      System.err.println("Caught BankException");
    }
  }
}
// Output is: Caught BankException
```

Implementing an Exception Hierarchy cont.

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class BankExceptions3 {
  public static void main(String[] args) {
    try {
      throw new ATMException();
    } catch (BankException e) {
      System.err.println("Caught BankException");
    } catch (ATMException e) {
      System.err.println("Caught ATMException");
    }
  }
}
// This code will not compile.
// catch(ATMException e) would never be reached
```

Exception API

- Functionality of `exception` is all inherited from **`Throwable`**
- Interesting **`java.lang.Throwable`** APIs
  - **`getMessage`**
  - **`getStackTrace`**
  - **`initCause`**
  - **`printStackTrace`**
  - **`toString`**

Reporting an Exception Stack Trace

```
class BankException extends Exception {
  BankException(String msg) { super(msg); }
}
public class BankExceptions4 {
  public static void main(String [] args) {
    try {
      throw new BankException ("I'm a BankException");
    } catch(BankException e) {
      System.err.println(e.getMessage());
      e.printStackTrace();
    }
  }
}
// Output is
// I'm a BankException
// BankException: I'm a BankException
// at BankExceptions4.main(BankExceptions4.java:7)
```

Nesting

- Java's exception mechanism supports nesting
- You can have
  - **`try-catch`** blocks in **`try`** blocks
  - **`try-catch`** blocks in **`catch`** blocks
  - **`try-catch`** blocks in **`finally`** blocks

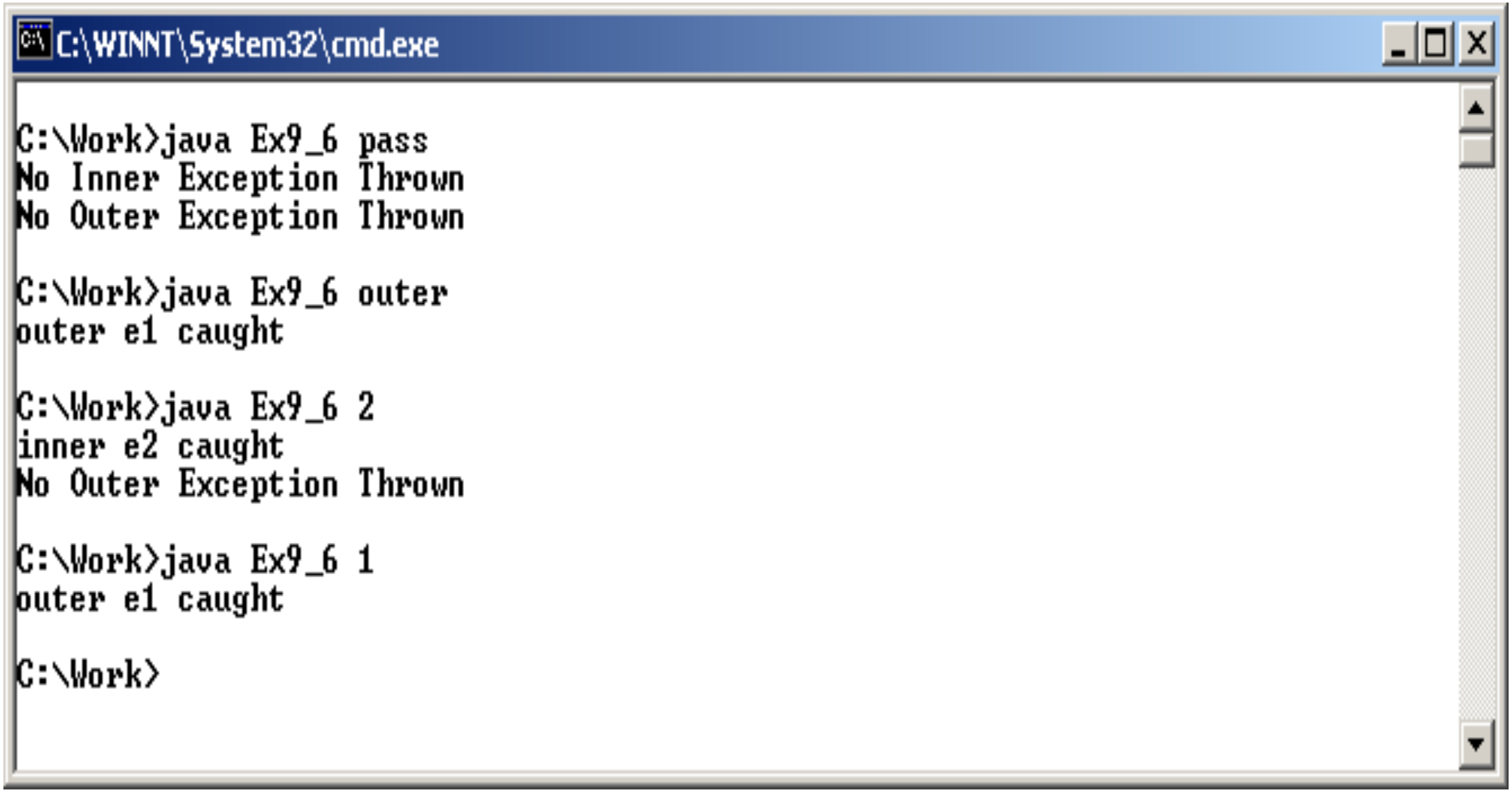
Nested `try` Blocks

```
class e1  extends Exception{}
class e2 extends Exception{}

public class Ex9_6 {
  public static void main(String[] args) {
    Ex9_6 testObj = new Ex9_6();
    testObj.exec(args[0]);
  }
  public void exec(String option) {
    // here is the outer try block
    try {
      if (option.equals("outer"))
        throw new e1();


        . . .
```

Nested `try` Blocks

```
  // inner try block
  try {
   if (option.equals("1"))
    throw new e1();
   if (option.equals("2"))
    throw new e2();
   System.out.println("No Inner Exception Thrown");
  } catch (e2 e) {
   System.err.println("inner e2 caught");
  }
  System.out.println("No Outer Exception Thrown");
 } catch (e1 e) {
  System.err.println("outer e1 caught");
 } catch (Exception e) {
  System.err.println("outer e2 caught");
 }
}
```

Nested `try` Blocks



```
C:\Work>java Ex9_6 pass
No Inner Exception Thrown
No Outer Exception Thrown

C:\Work>java Ex9_6 outer
outer e1 caught

C:\Work>java Ex9_6 2
inner e2 caught
No Outer Exception Thrown

C:\Work>java Ex9_6 1
outer e1 caught

C:\Work>
```

The `finally` Block Again

- All **`finally`** blocks are always executed
  - Whether the exception is thrown or not
  - Whether the exception is handled or not
  - Whether the exception came from a nested block or not
- Exceptions:
  - **`finally`** might not *complete* if another exception arises in the middle of processing the block
  - A call to **`System.exit()`** will also abandon current processing
  - Turning the power off or killing the VM process can also prevent **`finally`** from completing

Rethrowing Exceptions

- Sometimes a low level exception cannot be handled but it is not descriptive to the caller
  - Consider catching the exception, then throwing a new, application-level exception that is more descriptive

  -
    ```
    try {
      doCreditCardNetworkOperations();
    } catch (SocketTimeoutException ne) {
      // network not available, try again later…
      throw new RetryCreditCardLaterException(ne);
    }
    ```
- Notice that the exception constructor allows nesting of original exception (the “Cause”) inside the new semantic exception

Custom Exceptions

- In many cases you will want to create application specific exceptions
- Extend the `Exception` class
  - Subclass an existing exception type
  - Choose something that is a reasonable generalization of the problem if possible:
    - IOException
  - Otherwise choose between `RuntimeException`, `Error`, and `Exception`
- Maintain the reason message and cause
  - Invoke superclass constructors to manage this

`try-finally` Block

- A **`try`** block can be used without a **`catch`** clause.
- This is often desired when a resource is opened and we want to guarantee that it will be closed.
- Java guarantees that the **`finally`** block will always be executed, even if there is an uncaught exception.

```
try{
        Open Stream
}finally{
        Close Stream
}
```

Rules for Overloading Methods

- Overloading methods must be entirely compatible with the method they replace
  - Liskov substitution principle
- Overloading method must not break compiler checks regarding exceptions
  - May not throw checked exceptions that were not declared for the base method from an overloading method
- This also applies to interface implementation methods
  - Generalized methods often declare exceptions they do not actually throw

Rules for Overriding Methods (cont.)

- An overridden method cannot throw any exceptions that are not already being declared in the **`throws`** clause of the method.
- Doing so would break polymorphism and could possibly break an existing program that is not expecting a thrown exception.
- An overridden method is allowed to declare an exception that is a subclass of a declared exception for the overridden method.
- A **`catch`** clause will catch a super class exception of a subclass; therefore it is legal to throw a subclass’s exception.

Summary

In this module, we covered:

- Exceptions and why they are unavoidable as a rule
- Declaring method signatures with **`throws`**
- The Java exception hierarchy
- An application exception hierarchy
- The **`try-throw-catch`** construct
- Nested **`try`** blocks
- The **`finally`** clause
- Rethrowing exceptions