seed
beyond the obvious

Core Java

Exception Handling

Objectives

- Exception Handling in Java overview
- Identify the need for Exception Handling
- try/catch block
- Exception hierarchy
- RuntimeExceptions
- Flow control in try/catch blocks
- finally block
- Throwing multiple exceptions

seed
beyond the obvious

Objectives

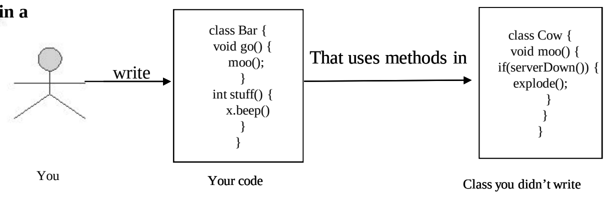
- Catching multiple exceptions
- Exceptions are polymorphic
- Throwing Exception
- Exception rules
- Describe the cascading of exceptions
- Distinguish between throw & throws keyword
- Create User defined Exceptions
- Assertion

Exception Handling in Java overview

- Exception : an OO way of handling errors.
- Keeps problem solving & error handling code different.
- Thus, program is less complex.
- Exceptions in Java are actual objects, instances of classes that inherit from class Throwable.
- These objects encapsulate the error information.
- Created when a abnormal situation arises in your java program.

Need for Exception Handling

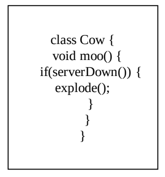
You want to call a method in a Class that you didn't write



```
class Bar {
  void go() {
    moo();
  }
  int stuff() {
    x.beep()
  }
}
```

```
class Cow {
  void moo() {
  if(serverDown()) {
    explode();
    }
  }
}
```

That method does something risky, something that might not work at runtime

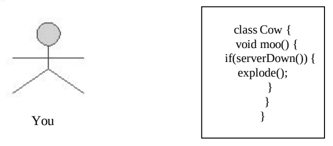


```
class Cow {
  void moo() {
  if(serverDown()) {
    explode();
    }
  }
}
```

Class you didn't write

Need for Exception Handling

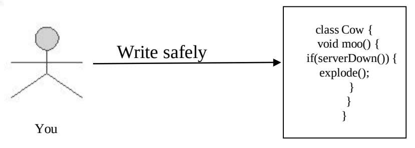
You need to know that the method You are calling is risky



```
class Cow {
  void moo() {
  if(serverDown()) {
    explode();
    }
  }
}
```

Class you didn't write

You then write code that can handle the failure if it does happen. You need to be prepared, just in case



```
class Cow {
  void moo() {
  if(serverDown()) {
    explode();
    }
  }
}
```

Class you didn't write

try/catch block

- Compiler needs to know that you know you are calling a risky method

```
import javax.sound.midi.*;

public class Musictest {   // this is the first one

   public void play() {

              Sequencer player = MidiSystem.getSequencer();

   }
}
```

- Example:- IOStreamDemoWithError.java

try/catch block

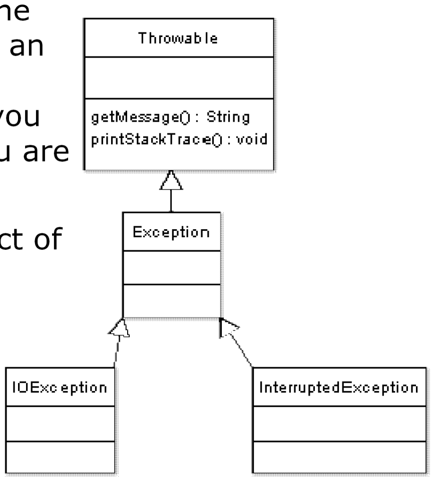
```
import javax.sound.midi.*;

public class Musictest {   // this is the first one

   public void play() {

     try {

              Sequencer player = MidiSystem.getSequencer();

          } catch (MidiUnavailableException ex) {
               ex.printStackTrace();
          }
   }
}
```

- Put the risky thing in a **try** block
- Make a **catch** block for what to do if the exceptional situation happens.
- Example:- IOStreamDemo.java

Exception

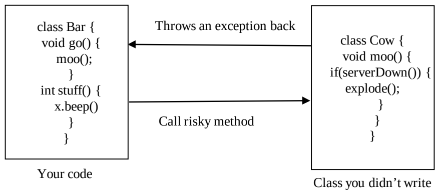
- A try/catch block tells the compiler that you know an exceptional thing could happen in the method you are calling, and that you are prepared to handle it
- An exception is an object of type **Exception**

```
try {
        //something risky
} catch (Exception ex) {
        try to recover
}
```



Exception

- If it is your code that catches the exception, then whose code throw it???



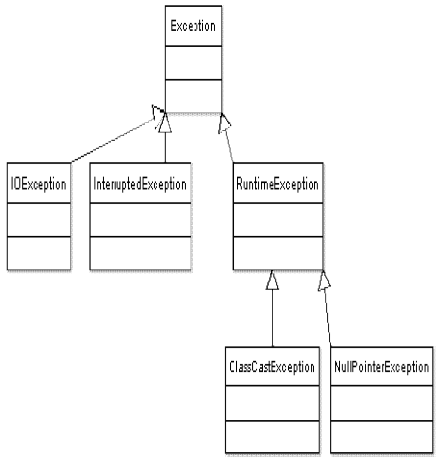
```
class Bar {
  void go() {
    moo();
  }
  int stuff() {
    x.beep()
  }
}
```

```
class Cow {
  void moo() {
    if(serverDown()) {
      explode();
    }
  }
}
```

- One method will catch what another method throws. An exception is always thrown back to the caller
- The method that throws has to declare that it might throw the exception

RuntimeExceptions

- The compiler checks for everything except **RuntimeExceptions**.
- Exceptions that are not subclasses of **RuntimeException** are checked for by the compiler. They are called "checked" exceptions
- **RuntimeException** occurs because user made a programming error i.e. code was not very robust.
- Includes problems as:
  - A bad cast
  - A out-of-bounds array access
  - A null pointer access



seed
beyond the obvious

Flow control in try/catch blocks

```
try {
            Foo a = x.doRiskyThing();
            int b = f.getNum();
     } catch (Exception ex) {
            System.out.println("failed");
   }
   System.out.println("success");
```

- If the **try** succeeds – doRiskyThing() does not throw an exception
  - The code in the **catch** block never runs
  - The code below the **catch** block runs
- If the **try** fails – doRiskyThing() does throw an exception
  - Rest of the **try** block doesn't run
  - The **catch** block runs, then the method continues on.

seed
beyond the obvious

finally block

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

```
try {
      turnWaterHeaterOn()
      x.boil();
    } catch (BoilingException ex) {
      ex.printStackTrace();
   } finally {
         turnWaterHeaterOff();
   }
   System.out.println(“success”);
```

- A finally block lets you put all your important cleanup code in one place instead of duplicating it.

seed
beyond the obvious

Throwing multiple exceptions

```
public class Account {
    public void withdrawMoney(int m) throws InsufficientBalanceException,
    TransactionFailureException {
          //Code that does withdrawing
    }
}
```

- A method can **throw** multiple exceptions if it needs. But a method’s declaration must declare all the checked exceptions it can throw
- How to catch multiple exceptions???

seed
beyond the obvious

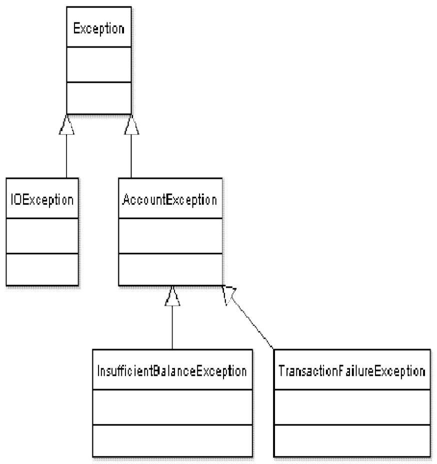
Catching multiple exceptions

- *public class AccountTest {*
```
        public void withdraw() {
      Account act = new Account(1234);
      try {
            act.withdrawMoney(1000);
      } catch(InsufficientBalanceException ex) {
                //recovery code here
      } catch(TransactionFailureException tex) {
                //recovery code here
       }
    }
}
```
- Stack the **catch** blocks under the **try**, one after the other

Exceptions are polymorphic

- You can **DECLARE** exceptions using supertype of the exceptions you throw
  public void withdrawMoney(int m) throws AccountException

- You can **CATCH** exceptions using a supertype of the exception thrown
```
try {
     act.withdrawMoney(1000);
   } catch(AccountException ex) {
     //recovery code here
   }
```

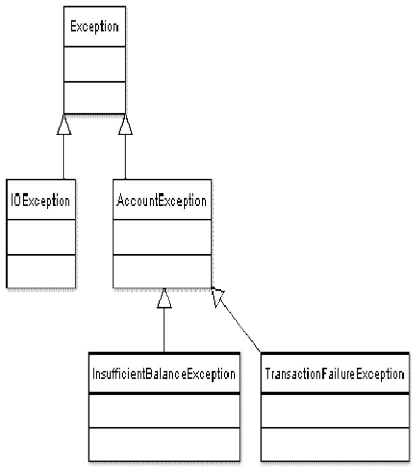


Exceptions are polymorphic

- Just because you **CAN** catch everything with one big super polymorphic catch, doesn't always mean you **should**.

```
try {
      act.withdrawMoney(1000);
    } catch(Exception ex) {
      //recovery code here
    }
```

  - You won't automatically know what went wrong



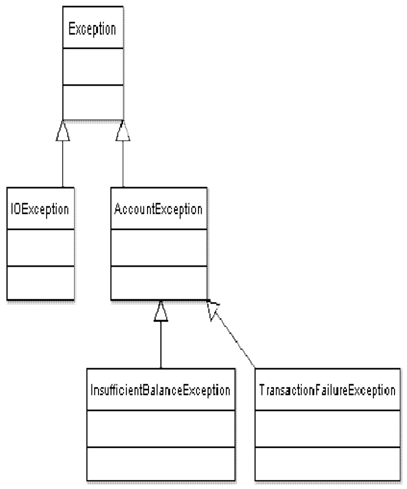
seed beyond the obvious

Exceptions are polymorphic

- Write a different catch block for each exception that you need to handle uniquely

```
try {
        act.withdrawMoney(1000);
    } catch(InsufficientBalanceException ex) {
        //recovery from insufficient bal
    } catch(TransactionFailureException tex) {
       //recovery from transaction failure
    } catch(AccountException aex) {
       //recovery from all others
    }
```

- Multiple catch blocks **must** be ordered from smallest to biggest

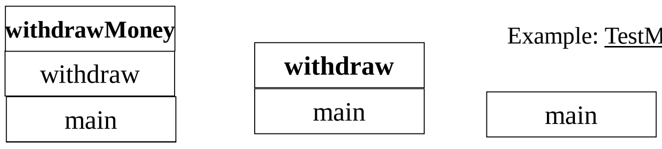


seed beyond the obvious

Throwing Exception

- If you don’t want to handle an exception, you can declare it by **throws**

```
public void withdraw() throws AccountException{
        Account act = new Account(1234);
         act.withdrawMoney(1000);
}
```



Example: TestMyException.java

The JVM shuts down

- main() calls withdraw()
- withdraw() calls withdrawMoney()
- withdrawMoney() runs and throws an AccountException

- withdrawMoney() pops off the stack immediately and the exception is thrown back to withdraw()
- withdraw() does not have a try/catch so…

- withdraw() pops off the stack immediately and the exception is thrown back to main(), which in turn throws it to JVM

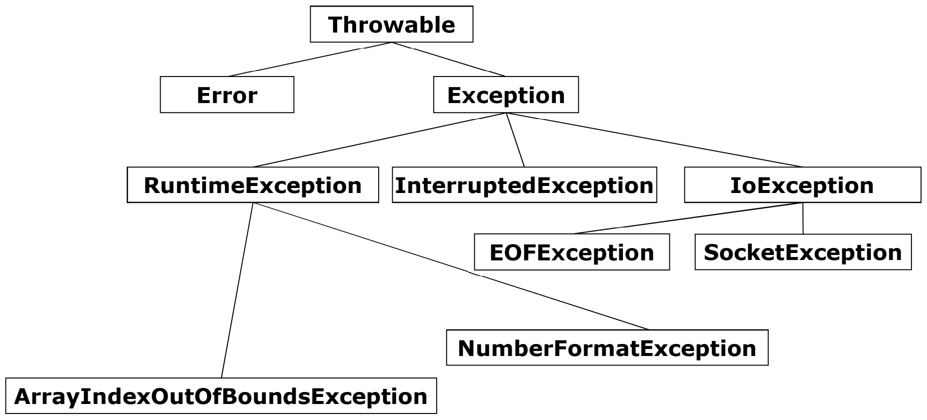
seed beyond the obvious

Exception rules

- You can’t have a catch or finally without a try
- You can’t put code between the try and the catch
- A try must be followed by either a catch or finally
- A try with only a finally (no catch) must still declare the exception
- If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method

seed beyond the obvious

Exception Hierarchy



The class "Throwable" and some of its Subclasses.

Errors

- Are internal errors in the java Runtime environment
- Describe internal errors & resource exhaustion in JVM.
- Rare & usually fatal.
- Used by Java run-time system to indicate errors having to do with run-time environment itself.
  - OutOfMemoryError
  - StackOverflowError

User Defined Exceptions

```
class MyException extends Exception {
   MyException() { }
   MyException(String s)
   {
        super(s);
   }
}
```

User Defined Exceptions

```
class ExceptionDemo {
public void myMethod() throws MyException
{
   ….
   if(<some condition>){
        throw new MyException();
   }
……
}
```

User Defined Exceptions

```
    public static void main(String args[ ]) {
          try{
                    ExceptionDemo e=new
          ExceptionDemo();
                    e.myMethod();
          }
          catch(MyException e) {
                    // handle  the exception
          }
}
```

Assertion

- When ***implementing*** and ***debugging*** a class, it is sometimes useful to state conditions that should be true at a particular point in a method. These conditions called assertions, help ensure a program’s validity by catching potential bugs and identifying possible logic errors during development.
- Java includes two versions of the **assert** statement for validating assertions programmatically. The assert statement evaluates a boolean expression and determines whether it is true or false

Assertion

- The first form of assert statement is
  - **assert *expressrion;*** this statement evaluates expression and throws an **AssertionError** if expression is false
- The second form is
  - ***assert expression1:expression2;*** this statement evaluates ***expression1*** and throws an **AssertionError** with ***expression2*** as the error message if ***expression1*** is false
- Example:-

  AssertTest.java

seed beyond the obvious

Assertion

- Note:-
  - By default assertions are disabled when executing a program because they reduce performance and are unnecessary for the program's user.
  - To enable assertion at runtime use the following version of java command
    - ***java –ea AssertTest***

seed beyond the obvious