

Exceptions

Overview

- Why Exceptions?
- Working with methods that throw exceptions
- The try-catch-finally block
- The class Exception and its subclasses
- Checked vs Unchecked Exceptions
- Defining your own Exceptions

Why Exceptions?

- Exceptions allow the programmer to treat error conditions outside the main logic flow
- Most programming languages (without exceptions) handle errors by passing return codes as error indicators

What's an Exception

- A signal that indicates an *exceptional condition* (something unexpected) has happened in your program
- To *throw an exception* is to signal that an exceptional condition has occurred
- To *catch an exception* is to handle the exception - to take whatever action is necessary
 - sometimes you can't do anything



Working with methods that throw exceptions

Exception Example 1 the FileInputStream class

API:
public FileInputStream (String s) throws FileNotFoundException;

Your code:
String s = "myfile.dat";
FileInputStream fis = new FileInputStream (s);

we MUST deal with the possibility of an Exception

may throw exception

```
try {  
    FileInputStream fis = new FileInputStream (s);  
} catch (FileNotFoundException e) {}
```

Exception Example 2 the Thread class

Thread static method API
public static void sleep (long millis)
throws InterruptedException;

User code:
Thread.sleep(1000);

will not compile unless...

Exception Example 2 [catching the exception]

Thread static method
public static void sleep (long millis)
throws InterruptedException;

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {}
```

Exception Example 2 [catching the exception]

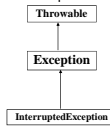
Thread static method

public static void sleep (long millis)

```
try {
    Thread.sleep(1000);
} catch (Exception e) {}
```

throws InterruptedException;

OR .. can catch a superclass of the Exception class



Handling Exceptions...

C:\class\Programs\Exceptions\ExceptTest.java:6:
Exception java.io.FileNotFoundException must be caught, or it must be declared in the throws clause of this method.

- Catch the exception in our method
- OR
- List the exception in our own method header

quick way to play with code that throws exceptions

```
class ThreadSleepTest {
    public static void main (String [] args)
        throws InterruptedException {
        for (int i=0; i< 10; i++) {
            if (i== 5) Thread.sleep(1000);
            System.out.print(i + " .. ");
        }
    }
}
```

```
>java ThreadSleepTest
0.. 1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

one second pause

```
class ThreadSleepTest2 {
    public static void main (String [] args) {
```

```
        try {
            for (int i=0; i< 10; i++) {
                if (i== 5) Thread.sleep(2000);
                System.out.print(i + " .. ");
            }
        } catch (InterruptedException e) {
        }
    }
}
```

try-catch the complete story

```
try {
    // code that might
    // throw an exception
} catch (ExceptionType variable) {
    // handle the exception if thrown
} finally {
    // .. always do this
}
```

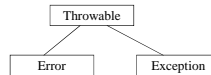
Try..Catch..Finally Combo Rules

- You cannot use try alone
- A try block requires catch or finally or both
- A catch or finally block requires a try block
- The finally block is always executed as part of the Java control flow - even if you use a return or break to avoid a finally block

The Exception Hierarchy

The Exception Hierarchy

A Java exception is an object that is an instance of some subclass of Throwable



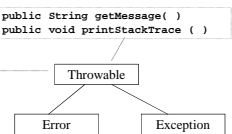
Errors are exceptions that result from system problems (e.g. Java Virtual Machine)

Errors are almost always unrecoverable - should not be caught

Conditions that may be caught and handled
Often recoverable

The Exception Hierarchy

includes a String message that is inherited by all subclasses



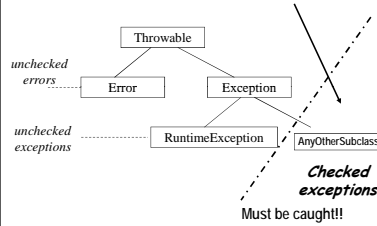
A user defined exception should subclass Exception

What Exceptions Must be Caught?

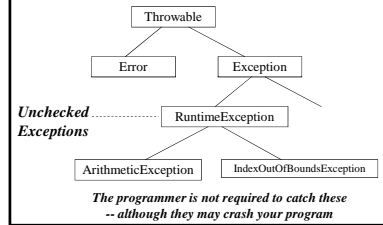
those that the compiler checks that you've handled



Checked Exceptions



UnChecked Exceptions



Throwing Exceptions

Throwing Unchecked Exceptions

- Unchecked Exceptions
 - A method may throw any unchecked exception
 - No requirement to declare anything in the method declaration

```

void foo() {
    if (...)
        throw new ArithmeticException ();

    if (...)
        throw new IndexOutOfBoundsException ();
}
    
```

Throwing Checked Exceptions

- Checked Exceptions
 - A method may throw any checked exception as long as either:
 - the exception class is declared in the method throws clause
 - a superclass of the exception class is declared in the method throws clause

```

void foo (int i) throws IOException {
    if (i == 0)
        throw new IOException();
}
    
```

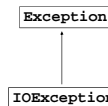
you must declare the type of any checked exception you throw ... OR

```

void foo (int i) throws Exception {
    if (i == 0)
        throw new IOException();
}
    
```

OR... you may declare a superclass of the exception type you throw

not good practice but allowed by Java's rules for subclass substitution



throw - catch parameter matching

```

void bar () {
    try {
        foo(4);
    } catch (IOException e) {
        // header
    }
}

void foo (int i) throws IOException {
    if (i == 0)
        throw new IOException();
}
    
```

the catch clause MUST specify a compatible type with throws header

throw - catch parameter matching

```
void bar () {
  try {
    foo(4);
  }
  catch (Exception e) {
  }
}

void foo (int i) throws IOException {
  if (i == 0)
    throw new IOException();
}
```

this will compile but is poor programming practice!

throw - catch parameter matching

```
void bar () {
  try {
    foo(4);
  }
  catch (IOException e) {
  }
}

void foo (int i) throws Exception {
  if (i == 0)
    throw new IOException();
}
```

compiler will not allow this!

Compiler: Exception java.lang.Exception must be caught

Multiple Catch Blocks..

```
try {
  someObject.test();
  anotherObject.foo();
} catch (InterruptedException e1) {
  // do something with e1
} catch (IOException e2) {
  // do something with e2
} catch (NullPointerException e3) {
  // do something with e3
}
```

checked in order until match is found!

Multiple Catch Blocks..GOTCHA!

```
try {
  someObject.test();
  anotherObject.foo();
} catch (Exception e1) {
  // do something with e1
} catch (IOException e2) {
  // do something with e2
} catch (NullPointerException e3) {
  // do something with e3
}
```

matches ALL Exceptions and subclasses of Exception

Multiple Catch Blocks

```
try {
  someObject.test();
  anotherObject.foo();
} catch (IOException e1) {
  // do something with e1
} catch (NullPointerException e2) {
  // do something with e2
} catch (Exception e3) {
  // do something with e3
}
```

list most specific Exceptions first

list most general Exceptions last

Creating Your Own Exception Classes

Creating Your Own Exception Class

- extend the class Exception
- take advantage of the Exception class constructor
 - Exception (String s)
- the string can be retrieved by sending getMessage() to the exception object

```
public class MyException extends Exception {
  // constructor
  public MyException (String s) {
    super(s); // calls the superclass constructor
  }
}
```

throwing your exception

You throw an exception by combining the keyword throw with object creation:

```
throw new MyException("MyException ZZ");
```

this can be a very specific message

```
public class A {
  public static void main(String args []) {
    A myObj = new A();
    try {
      myObj.test();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
  void test () throws MyException {
    throw new MyException("hey");
  }
}

class MyException extends Exception {
  MyException (String s) {
    super(s); // calls the superclass constructor
  }
}

// you must call the superclass' constructor
```

How should you handle Exceptions?

```
FileInputStream in =  
new  
FileInputStream("employee.dat");
```



Ask the compiler,
AND/OR
Look up the method
declarations

```
public FileInputStream (String name) throws  
FileNotFoundException;
```

requires a design decision how to handle if NOT a subclass of RuntimeException

Options

- Do nothing .. ignore it
- Print a message and bail out
- Pass the buck
- Get some new information and retry
- Pass an application-specific exception back to the caller

Do Nothing

```
try {  
FileInputStream in =  
new FileInputStream("employee.dat");  
} catch (FileNotFoundException e) {}
```

compiler is happy but
programmer may spend hours
trying to debug the program

Print and Bail

```
try {  
FileInputStream in =  
new FileInputStream("employee.dat");  
}  
catch (FileNotFoundException e) {  
System.out.println ("can't find  
employee.dat file");  
System.exit(1);  
}
```



Print and Rethrow

```
try {  
FileInputStream in =  
new FileInputStream("employee.dat");  
}  
catch (FileNotFoundException e) {  
System.out.println ("can't find employee.dat file");  
throw e;  
}
```



Pass the Buck

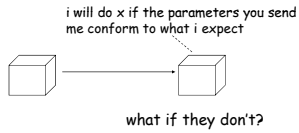
```
public void foo ()  
throws FileNotFoundException {  
FileInputStream in =  
new FileInputStream("employee.dat");  
}
```

Retry Option -- put try/catch in a loop

```
int tries = 0;  
while (tries < 3) {  
try {  
String s = getNameFromUser();  
FileInputStream in =  
new FileInputStream(s);  
} catch (FileNotFoundException e) {  
tries++;  
}  
} // end while loop
```

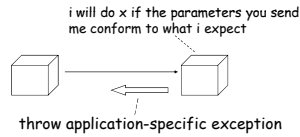
Application Specific Exceptions

- Components by "contract"



Application Specific Exceptions

- Components by "contract"



Throw Application-Specific Exception

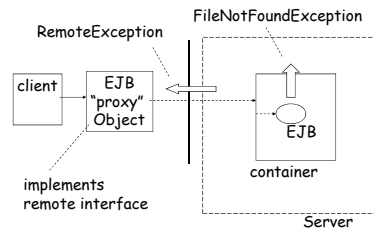
```
boolean success = false;
int tries = 0;
while (tries < 3) {
  try {
    String s = getNameFromUser();
    DataInputStream in = new DataInputStream (new
      FileInputStream(s));
    success = true;
  } catch (FileNotFoundException e) {
    tries++;
  }
} // end while

if (!success)
  throw new UsersJerkException ("user can't supply
  a correct file name");
```

Changing the Exception

- Often seen JDK technique when working with multi-tiered systems
- Seen in Enterprise Java Beans
 - server component framework
 - clients do not see the actual exception that may be thrown when EJB tries to perform a service

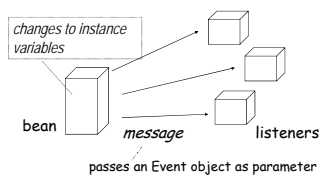
Client talks to "proxy" object



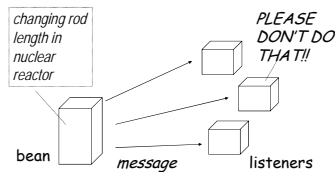
Exceptions as a Communication Mechanism

seen in Java Beans...

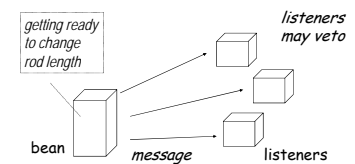
Beans can notify Listener objects of changes



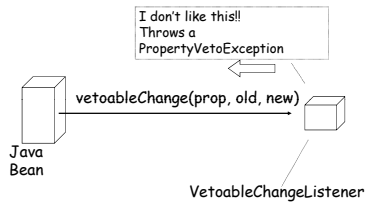
JDK designers needed a way to let listeners veto an object's change



Java Beans can have Constrained Properties and Vetoable Listeners



can VETO by throwing an Exception



Bean code

```
public void setCreditLimit (int newval) {  
    try {  
        notifyListeners("credit", credit, newval);  
        credit = newval;  
    }  
    catch (PropertyVetoException e) { }
```



Summary

- Exceptions are a useful way to structure the normal control flow of an application from the exceptional conditions that may occur
- Checked exceptions must be understood and dealt with by the programmer
- Options are
 - handle in a catch block
 - pass the buck (and declare that you throw)