Assertions and Exception Handling in Java

Used to detect and handle unusual or exceptional conditions

One way of detecting errors is assertions

- assert (that something must be true);
  - –Your program crashes when the assertion is false
  - –Use assertions sparingly.
  - –Assertions are to detect incorrect and unrecoverable events that are the programmer's fault
  - –Assertions test things that should never happen
- exceptions allow an action to be taken
  - –Use exceptions when the event is unpredictable, but recoverable.
  - –The fault is with the user of the software and not the program.
- Exceptions detect unusual events, assertions detect bugs

Where an assertion might be used

- Consider a program that opens a file, checks that the file was successful opened, and then stores a *handle* (i.e. a way to access the file such as a pointer or reference) to the file
- Sometime later another routine grabs the handle and attempts to write the file
- The file pointer is null
  - An assertion can be used to catch this.
  - The fault is almost certainly the programmer's
  - An assertion can be used at runtime to detect this
  - Shipped, non-alpha, non-beta software should not have active assertions.

Where an exception might be used

- Consider a program that opens a file, checks that the file was successful opened, and then stores a *handle* (i.e. a way to access the file such as a pointer or reference) to the file
- The file to be opened is specified by a file name.
- Suppose the file is to be read, should already exists, but no file by that name exists.
  - This is almost certainly a user error, not a programmer error
  - It would be nice to give the user a second chance or at least be informed of the error
  - It is also unpredictable when this will happen
  - An exception is a good way to check for this

exceptions are a way to detect and possibly correct unusual conditions

- always check the return value of function calls
  - –memory allocation
  - –opening a file
  - –sending data through network
- Exceptions are a good way of checking for problems without doing too much damage to the flow of control of the normal program logic.

Java Assertions

```
public class AssertionTest3 {

   public static void main(String argv[ ]) throws IOException {
      System.out.print("Enter your marital status: ");
      int c = System.in.read();
      switch ((char) c) {
         case 's':
         case 'S': System.out.println("Single"); break;
         case 'm':
         case 'M': System.out.println("Married"); break;
         case 'd':
         case 'D': System.out.println("Divorced"); break;
         default: assert !true : "Invalid Option"; break;
      }

   }
}
```

Compile using the "-source 1.4" (or higher option) with some versions of javac

Assertions are disabled by default, use the "-ea" option to turn them on

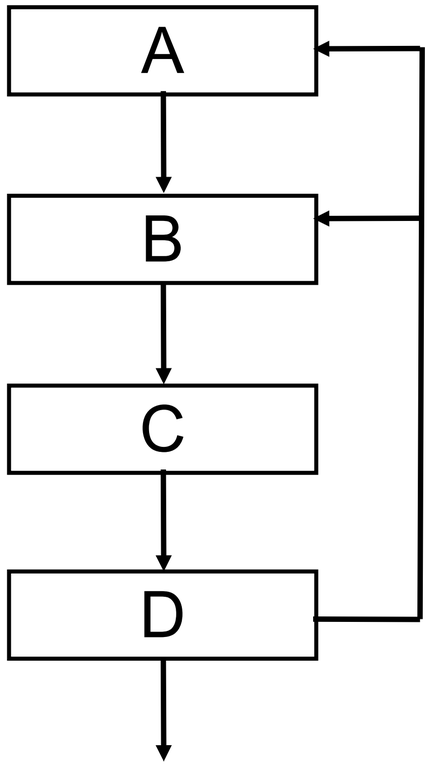
Handle problems when they happen

- A typical approach is to check before proceeding:

```
retval = func(parameters);
if (retval != 0) {
   handle the condition
}
```

- However, sometimes the immediate caller does not know how to handle the problem. This is especially common for reused and reusable code.

What about errors deep in a chain of calls?



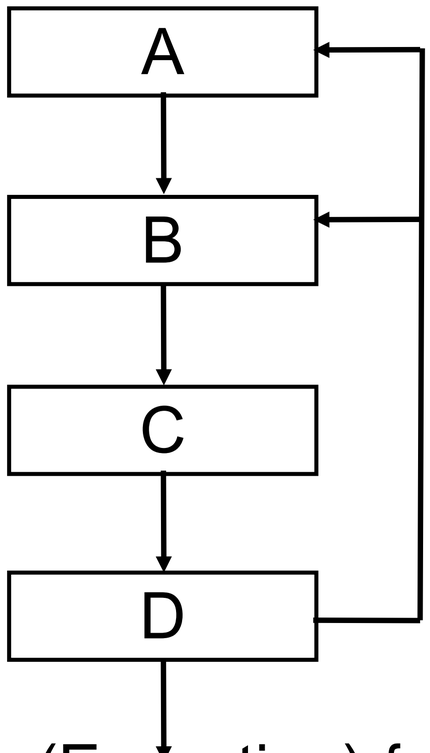
try to open a file that does not exist

- A gets a file name, calls B; B calls C; C calls D.
- D cannot open the file
  - –Should D create the file?
  - –Can D even make that decision?
- Maybe A would prefer the user to give a different file name.
- Maybe, B wants to simply skip the file.

⇒ need a way to inform a caller that is several control flow actions away

Concepts of Exception Handling

```
try {
```




```
} catch (Exception) {
    handle exception;
}
```

- A, B, or C can throw an exception and it will be caught by

try { ... } catch

- If an exception is not caught, the program terminates.

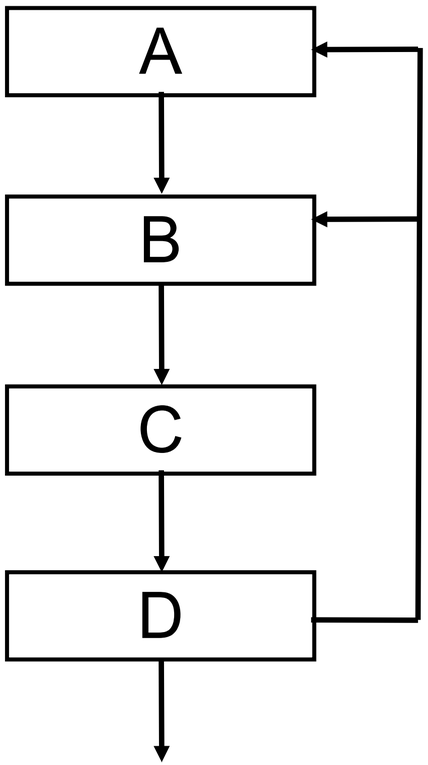
throw-catch

- When an unexpected situation occurs and the current function cannot handle it, it ***throws*** an exception.
- If no local *try-catch* surrounding the *throw* the exception is sent to the immediate caller. If it does not ***catch*** this exception, the exception is thrown to the next caller in the call stack.
- If an exception is not caught anywhere in the call stack, the program terminates (no where else to throw it)
- Should a calling function be aware of exceptions thrown by the called function?
  - Java requires syntax to make it explicit that the programmer wants the exception re-thrown.

```
public class ArrayInit {

   public static void foo( ) {
      bar( );
   }

   public static void bar( ) {
      int[ ] iAry = new int[ ] {0, 1, 2, 3};
      System.out.println(iAry[4]);
   }

   public static void main(String[] args) {
      foo( );
   }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
        at ArrayInit.bar(ArrayInit.java:9)
        at ArrayInit.foo(ArrayInit.java:4)
        at ArrayInit.main(ArrayInit.java:13)
```

Catching an Exception



- An exception is caught by the closest caller in the runtime call stack.
- If both A and B catch the same exception and D throws the exception, B will catch it.
- The handling code **can throw** the exception **again** to its call stack; the handling code can also throw a **different** exception.
- An exception may pass **parameters** through the call stack.

Exception Handling in Java

try-catch-finally

```
try {
    ...
} catch (exception_type1 id1) {
    ...
} catch (exception_type2 id2) {
    ...
} catch (exception_type3 id3) {
    ...
} finally {
    // code here executes, both when an exception is caught, when an exception does not occur,
    // or when an exception occurs and is not one of the caught exceptions.
```

```java
import java.lang.*;

public class B {

  public B( ) { }

  public void foo(int i) throws MyException, MyException1 {
    if (i == 0) {
      throw new MyException1( );
    } else if (i == 1) {
      throw new MyException( );
    }
  }
}
```

```java
import java.lang.*;

class MyException extends Exception {
  public MyException( ) {
    super();
  }
}
```

```java
import java.lang.*;

class MyException1 extends Exception {
  public MyException1( ) {
    super();
  }
}
public class T {

  /* basic reference operations */

  public static void main(String[] args) throws MyException1 {
    B b = new B( );
    for (int i = 2; i > -1; i--) {
      try {
        System.out.println("trying with "+i);
        b.foo(i);
        System.out.println("no exception thrown with "+i);
      }
      catch (MyException e) {System.out.println("MyException caught"+i);}
      finally {System.out.println("finally executed "+i);}
    }
  }
}
```

```
import java.lang.*;

class MyException1 extends Exception {
  public MyException1( ) {
    super();
  }
}

public class T {

    /* basic reference operations */

   public static void main(String[] args) throws MyException1 {
      B b = new B( );
      for (int i = 2; i > -1; i--) {
         try {
            System.out.println("trying with "+i);
            b.foo(i);
            System.out.println("no exception thrown with "+i);
         }
         catch (MyException e) {System.out.println("MyException caught"+i);}
         finally {System.out.println("finally executed "+i);}
      }
   }
}
```

```
import java.lang.*;

public class B {

   public B( ) { }

   public void foo(int i) throws MyException, MyException1 {
      if (i == 0) {
         throw new MyException1( );
      } else if (i == 1) {
         throw new MyException( );
      }
   }
}

import java.lang.*;

class MyException extends Exception {
  public MyException( ) {
    super();
  }
}
```

In Java, most exceptions that may be thrown by a method must be declared.

Serves as documentation and lets the compiler know what calling methods should catch or re-throw

All user-defined exceptions extend Exception, i.e. inherit from it.

Note that this could cause problems with multiple inheritance restrictions, but really exception classes should be specialized.

1. gather the relevant data about the error
2. make it available to the handler

```
import java.lang.*;

class MyException1 extends Exception {
  public MyException1( ) {
    super();
  }
}

public class T {

    /* basic reference operations */

   public static void main(String[] args) throws MyException1 {
      B b = new B( );
      for (int i = 2; i > -1; i--) {
         try {
            System.out.println("trying with "+i);
            b.foo(i);
            System.out.println("no exception thrown with "+i);
         }
         catch (MyException e) {System.out.println("MyException caught"+i);}
         finally {System.out.println("finally executed "+i);}
      }
   }
}
```

```
trying with 2
no exception thrown with 2
finally executed 2
trying with 1
MyException caught1
finally executed 1
trying with 0
finally executed 0
Exception in thread "main" MyException1
        at B.foo(B.java:9)
        at T.main(T.java:10)
```

Remember from the previous slide that foo throws an exception when *i = 1* and *i = 0*.

Multiple Exceptions

```
// ExceptionUsage4.java
class MyException extends Exception {
   private String me_message;
   public MyException(String msg) {
      me_message = message;
   }
   public String toString( ) {
      return me_message;
   }
}
class Err extends Exception {
   private int e_value;
   public Err(int i) {
      e_value = i;
   }
}
class ExceptionUsage4 {
   static void f( ) throws MyException {
      throw new MyException("hello");
   }
   static void g( ) throws Err( ) {
      throw new Err(71);
   }
   static void h( ) throws MyException, Err {
      f( );
      g( );
   }

   public static void main(String{ } args) {
      try {
         h( );
      } catch (MyException meobj) {
         System.out.println("caught MyException "+meobj);
      } catch (Err eobj) {
         System.out.println("caught Err " + eobj);
      }

      try {
         g( );
      } catch (Err eobj) {
         System.out.println("caught Err "+ eobj);
      }
      /*
       * g( ) cannot throw MyException
        catch (MyException meobj) {
           System.out.println("caught MyException ");
        }
      */
   }
```

caught MyException hello

caught Err 71

caught Err 71

```
static void h( ) throws MyException, Err {
   f( );
   g( );
}

public static void main(String{ } args) {
   try {
      h( );
   } catch (MyException meobj) {
      System.out.println("caught MyException "+meobj);
   } catch (Err eobj) {
      System.out.println("caught Err " + eobj);
   }

   try {
      g( );
   } catch (Err eobj) {
      System.out.println("caught Err "+ eobj);
   }  //  g( ) cannot throw MyException
    catch (MyException meobj) {
      System.out.println("caught MyException ");
    }
   */
}
```

Error statement if the comment on the previous slide was made compilable code would be something like:

```
T.java:13: exception MyException is never thrown in body of corresponding try
statement
         catch (MyException e) {System.out.println("MyException caught"+i);}
         ^
1 error
```

```
class Err extends Exception {
   private int e_value;
   public Err(int i) {
      e_value = i;
   }
}

static void f( ) throws Err {
   throw Err(57);
}

public static void main(String{ } args) {
   try {
      f( );
   } catch (Exception e) {
      System.out.println("caught Exception "+e);
   } catch (Err eobj) {
      System.out.println("caught Err " + eobj);
   }
}
```

Which catch clause catches the exception?

The answer is the first one (Exception e)

The rule is to traverse the catches in syntactic order and execute the first catch that has a type match.

Therefore, always order *catches* from most to least derived, as below:

```
public static void main(String{ } args) {
   try {
      f( );
   } catch (Err eobj) {
      System.out.println("caught Err " + eobj);
   } catch (Exception e) {
      System.out.println("caught Exception "+e);
   }
}
```

Catch and re-Throw

```
class ExceptionUseEx {

   public static void f( ) throws MyException {
      throw new MyException("hello");
   }

   public static void g( ) throws MyException {
      try {
         f( );
      } catch (MyException e) {
         System.out.println("catching and re-throwing in g "+e);
         throw new MyException("ece30862");
      }
   }

   public static void main(String[ ] args) {
      try {
         g( );
      } catch (MyException e) {
         System.out.println("caught MyException in main "+e);
      }
   }
}
```

catching and throwing a new exception

caught MyException in main ece30862

```
class ExceptionUseEx {

   public static void f( ) throws MyException {
      throw new MyException("hello");
   }

   public static void g( ) throws MyException {
      try {
         f( );
      } catch (MyException e) {
         System.out.println("catching and re-throwing in g "+e);
         throw e;
      }
   }

   public static void main(String[ ] args) {
      try {
         g( );
      } catch (MyException e) {
         System.out.println("caught MyException in main "+e);
      }
   }
}
```

catching and re-throwing

caught MyException in main hello

Exceptions and File IO and an example of how finally is useful

Java - JavaException/src/ExceptionUsage5.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

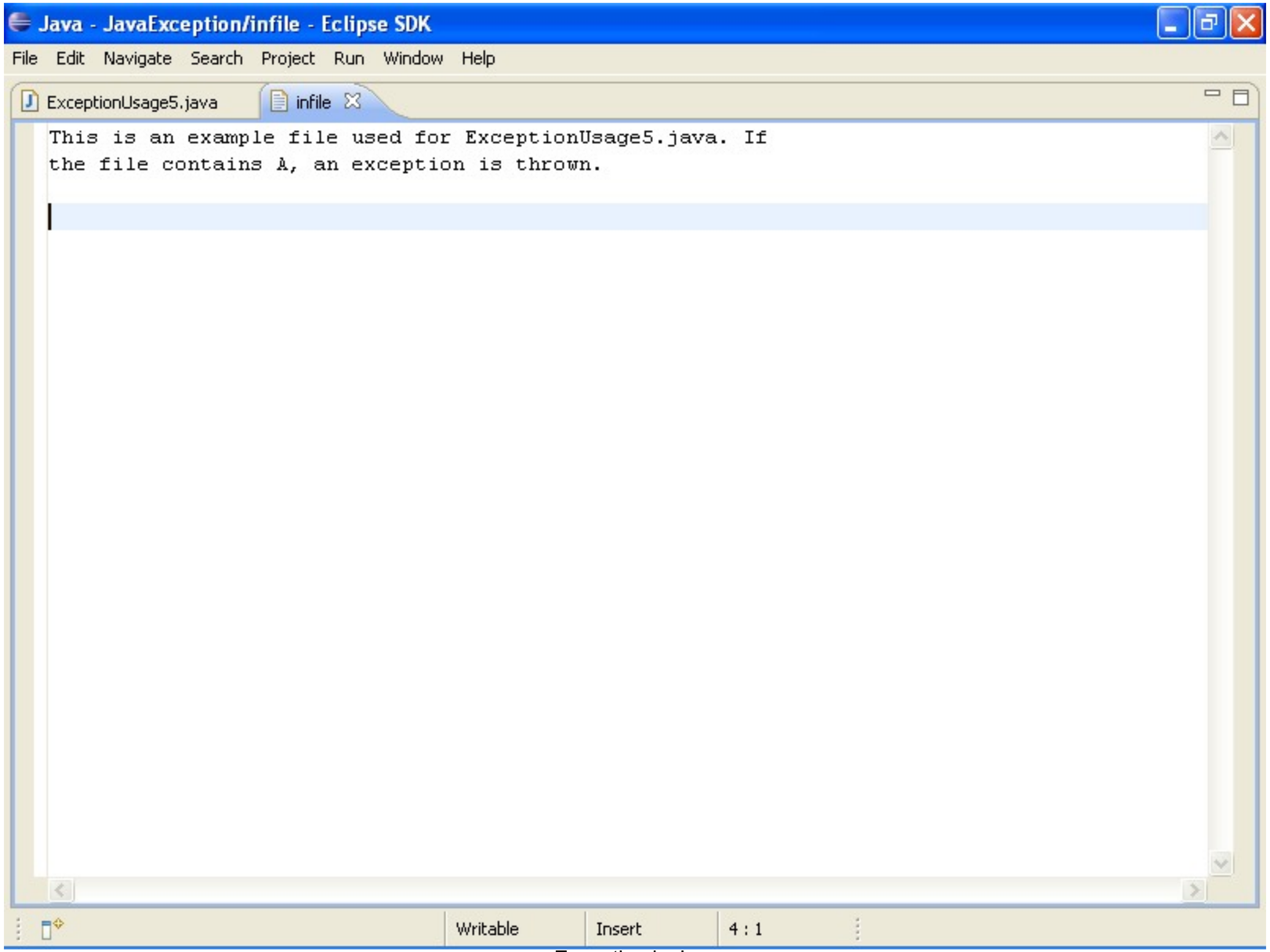
ExceptionUsage5.java | infile

```
//ExceptionUsage5.java
import java.io.*;
class ExceptionUsage5 {
    static void foo() throws Exception { throw new Exception(); }
    static void bar() throws Exception {
        FileReader input = null;
        try {
            input = new FileReader( "infile" );
            int ch;
            while ( ( ch = input.read() ) != -1 ) {
                if ( ch == 'A' ) {
                    System.out.println( "found it" );
                    foo();
                }
            }
        } finally {
            if ( input != null ) {
                input.close();
                System.out.println("input stream closed successfully");
            }
        }
        System.out.println( "Exiting bar()" );
    }
    public static void main( String[] args ) {
        try {
            bar();
        } catch( Exception e ) {
            System.out.println( "caught exception in main" );
```

Use finally to make sure the file is closed if *foo* is called and an exception is caught.

Since the exception is not caught in a catch clause it will be thrown up to the caller of bar.

Writable | Smart Insert | 1 : 1


Java - JavaException/infile - Eclipse SDK
File Edit Navigate Search Project Run Window Help
ExceptionUsage5.java
infile
This is an example file used for ExceptionUsage5.java. If
the file contains A, an exception is thrown.
Writable
Insert
4 : 1

Java - JavaException/src/ExceptionUsage5.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

ExceptionUsage5.java | infile

```
//ExceptionUsage5.java (modified)
import java.io.*;
class ExceptionUsage5 {
    static void foo() throws Exception { throw new Exception(); }
    static void bar() throws Exception {
        FileReader input = null;
        try {
            input = new FileReader( "infile" );
            int ch;
            while ( ( ch = input.read() ) != -1 ) {
                if ( ch == 'A' ) {
                    System.out.println( "found it" );
                    foo();
                }
            }
        } catch (IOException ioe) {
            System.out.println("IOException occurs " + ioe);
        } finally {
            if ( input != null ) {
```

Console

<terminated> ExceptionUsage5 [Java Application] C:\Program Files\java\jre1.6.0_03\bin\javaw.exe (May 25, 2008 5:10:17 PM)

```
found it
input stream closed successfully
caught exception in main
```

Writable | Smart Insert | 34 : 1

YHL

Overhead of Exception Handling

- In Java, invoking a function that throws exceptions 50% of the time doubles the running time over no exceptions.
- Do not overuse exceptions. They should be *exceptional.*
  - If the cost of handling exceptions is something you are worrying about, you are almost certainly overusing them.
- In many cases, if the errors can be detected by the return code

```
retval = function(...);
if (retval < 0) { ... /* handle error */ }
```

  then, you should do so without using try-catch.
- Don't use exceptions **just** as a way of letting someone else deal with your errors!
- Detect problems early, instead of using exceptions. For example, check whether network connection is valid before sending data (and catch exception when the sending fails).
- Catch an exception early and prevent its propagation.