Error Handling with Exceptions

Introduction

- Catch an error at compile time
- Handle the rest errors at run time
  - Allow the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly
- Java provides a consistent error-reporting model using exceptions to increase the robustness of your code
  - Exceptions are events that occur during the execution of programs that disrupt the normal flow of instructions
- The goals for exception handling
  - Simplify the creation of large, reliable programs using less code than currently possible
  - To do so with more confidence that your application doesn't have an unhandled error

Concepts

- **The word "exception" is meant in the sense of "I take exception to that. "**
  - **At the point where the problem occurs, you might not know what to do with it**
  - **You must stop, and somebody, somewhere, must figure out what to do**
  - **Don’t have enough information in the current context to fix the problem**
  - **Hand the problem out to a higher context where someone is qualified to make the proper decision**
- **Benefit of exceptions**
  - **Reduce the complexity of error-handling code**
  - **You only need to handle the problem in one place, in the so-called *exception handler***

Basic exceptions

- An *exceptional condition* is a problem that prevents the continuation of the current method or scope
  - A normal problem: you have enough information in the current context to somehow cope with the difficulty
  - An exceptional condition: you cannot continue processing because you don't have the information necessary to deal with the problem in the current context
  - E.g., Division
- When you throw an exception, several things happen
  - The exception object is created in the same way that any Java object is created: on the heap, with *new*
  - The current path of execution is stopped and the reference for the exception object is ejected from the current context
  - The *exception-handling mechanism* takes over and begins to look for an appropriate place (*exception handler*) to continue executing the program

Exception arguments

- You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context
  - This is called throwing *an exception*

```
if(t == null)
  throw new NullPointerException();
```

- There are two constructors in all standard exceptions
  - The *default constructor*
  - Take a string argument so that you can place pertinent information in the exception

```
throw new NullPointerException("t = null");
```

- You can throw any type of *Throwable*, which is the *exception root class*

Catching an exception

- **Set up a *try block* to capture an exception**

```
try {
  // Code that might generate exceptions
}
```

- **With exception handling, you put everything in a *try* block and capture all the exceptions in one place**

- **Much easier to write and read**
  - **The goal of the code is not confused with the error checking**

Exception handlers

- **The thrown exception must end up someplace, called *exception handler***

```
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...
```

- **Each *catch* clause (exception handler) is like a little method that takes one and only one argument of a particular type**
- **If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception**

Termination vs. resumption

- There are two basic models in exception-handling theory: *termination & resumption*
- Java supports *termination*
  - The error is so critical that there's no way to get back to where the exception occurred
- The alternative is called *resumption*
  - The exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time
- Historically, programmers using operating systems that supported presumptive exception handling eventually ended up using termination-like code and skipping resumption

Creating your own exceptions

- **The Java exception hierarchy cannot foresee all the errors you might want to report**
  - **Create your own to denote a special problem that your library might encounter**

- **To create your own exception class, you must inherit from an existing exception class**

Creating your own exceptions

- **The most trivial way to create a new type of exception is just to let the compiler create the default constructor for you**
- **The most important thing about an exception is the *class name***

```
1  class SimpleException extends Exception {}
2
3  public class InheritingExceptions {
4    public void f() throws SimpleException {
5      System.out.println("Throw SimpleException from f()");
6      throw new SimpleException();
7    }
8    public static void main(String[] args) {
9      InheritingExceptions sed = new InheritingExceptions();
10     try {
11       sed.f();
12     } catch(SimpleException e) {
13       System.out.println("Caught it!");
14     }
15   }
16 }
```

Creating your own exceptions (Cont.)

- You can also create an exception class that has a constructor with a ***String*** argument
  - ***printStackTrace( )*** produces information about the sequence of methods that were called to get to the point where the exception happened

```
1  class MyException extends Exception {
2    public MyException() {}
3    public MyException(String msg) { super(msg); }
4  }
5
6  public class FullConstructors {
7    public static void f() throws MyException {
8      System.out.println("Throwing MyException from f()");
9      throw new MyException();
10   }
11   public static void g() throws MyException {
12     System.out.println("Throwing MyException from g()");
13     throw new MyException("Originated in g()");
14   }
15   public static void main(String[] args) {
16     try {
17       f();
18     } catch(MyException e) {
19       e.printStackTrace(System.out);
20     }
21     try {
22       g();
23     } catch(MyException e) {
24       e.printStackTrace(System.out);
25     }
26   }
27 }
```

```
/* Output:
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:9)
        at FullConstructors.main(FullConstructors.java:17)
Throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:13)
        at FullConstructors.main(FullConstructors.java:22)
*///:~
```

The exception specification

- In Java, you're encouraged to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method
- The exception specification uses an additional keyword, ***throws***, followed by a list of all the potential exception types
  - So your method definition might look like this

```
void f() throws TooBig, TooSmall, DivZero { //...
```

- If you say

```
void f() { //...
```

  - No exceptions are thrown from the method
  - Except for the exceptions inherited from ***RuntimeException***, which can be thrown anywhere without exception specifications

The exception specification (Cont.)

- If the code within your method causes exceptions, but your method doesn't handle them
- The compiler will detect this and tell you
  - either handle the exception or indicate with an exception specification that it may be thrown from your method
- Java enforces exception specifications from top to bottom
  - Guarantees that a certain level of exception correctness can be ensured *at compile time*
- There is one place you can lie: You can claim to throw an exception that you really don't
  - you can actually start throwing the exception later without requiring changes to existing code
- Exceptions that are checked and enforced at compile time are called *checked exceptions*

Catching any exception

- It is possible to create a handler that catches any type of exception
  - Catch the base-class exception type ***Exception***

```
catch(Exception e) {
  System.out.println("Caught an exception");
}
```

  - This will catch any exception, so put it at the end of your list of handlers to avoid preempting any exception handlers that might otherwise follow it

- To get specific information about the exception, you can call the methods that come from ***Exception***'s base type ***Throwable***

Catching any exception (Cont.)

- Gets the detail message, or a message adjusted for this particular locale
  - *String getMessage( )*
  - *String getLocalizedMessage( )*
- Returns a short description of the *Throwable*, including the detail message if there is one
  - *String toString( )*
- Prints the *Throwable* and the *Throwable*'s call stack trace
  - *void printStackTrace( )*
  - *void printStackTrace(PrintStream)*
  - *void printStackTrace(java.io.PrintWriter)*
- Records information within this *Throwable* object about the current state of the stack frames
  - *Throwable fillInStackTrace( )*

Catching any exception (Cont.)

- Here's an example that shows the use of the basic *Exception* methods
  - The methods provide successively more information—each is effectively a superset of the previous one

```
1  import static net.mindview.util.Print.*;
2
3  public class ExceptionMethods {
4    public static void main(String[] args) {
5      try {
6        throw new Exception("My Exception");
7      } catch(Exception e) {
8        print("Caught Exception");
9        print("getMessage():" + e.getMessage());
10       print("getLocalizedMessage():" +
11         e.getLocalizedMessage());
12       print("toString():" + e);
13       print("printStackTrace():");
14       e.printStackTrace(System.out);
15     }
16   }
17 }
```

```
/* Output:
Caught Exception
getMessage():My Exception
getLocalizedMessage():My Exception
toString():java.lang.Exception: My Exception
printStackTrace():
java.lang.Exception: My Exception at
ExceptionMethods.main(ExceptionMethods.ja
va:6)
*///:~
```

Rethrowing an exception

- **Rethrow the exception that you just caught**

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

- **Rethrowing an exception causes it to go to the exception handlers in the *next higher context***
- **Any further catch clauses for the same try block are still ignored**
- ***printStackTrace( )* will pertain to the exception's origin, not the place where you rethrow it**
- ***fillInStackTrace( )* installs new stack trace information**

Rethrowing an exception (Cont.)

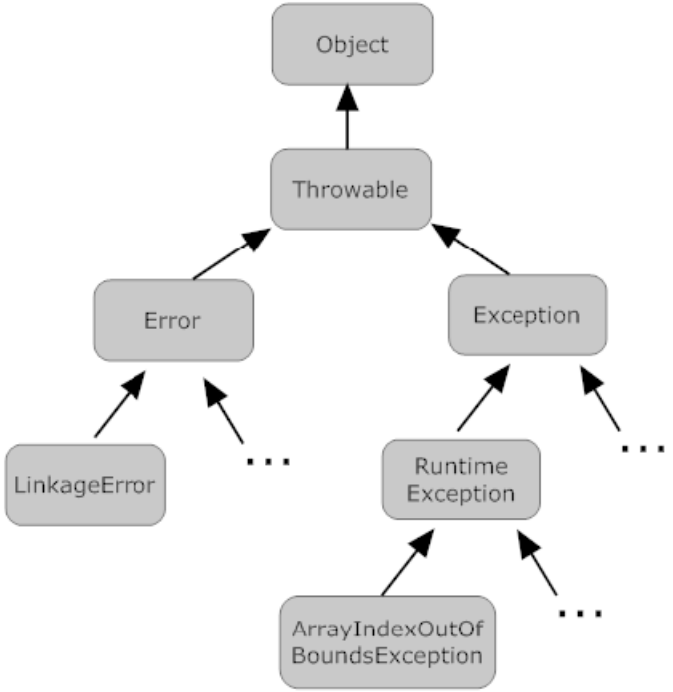
```java
//: exceptions/Rethrowing.java
// Demonstrating fillInStackTrace()

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 Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside g(),e.printStackTrace()");
      e.printStackTrace(System.out);
      throw e;
    }
  }
  public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside h(),e.printStackTrace()");
      e.printStackTrace(System.out);
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try {
      g();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
}
```

```
/* Output:
originating the exception in f()
Inside g(),e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)
originating the exception in f()
Inside h(),e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.h(Rethrowing.java:20)
        at Rethrowing.main(Rethrowing.java:35)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.h(Rethrowing.java:24)
        at Rethrowing.main(Rethrowing.java:35)
```

Standard Java exceptions

- The Java class *Throwable* describes anything that can be thrown as an exception
- There are two general types of *Throwable* objects
  - *Error* represents compile-time and system errors
  - *Exception* is the basic type that can be thrown from any of the standard Java library class methods and from your methods and runtime accidents



Special case: *RuntimeException*

- ***RuntimeException*** **are always thrown automatically by Java and you don't need to include them in your exception specifications**
  - **E.g., NullPointerException**
- **You never need to write an exception specification saying that a method might throw a *RuntimeException***
  - **They are *unchecked exceptions* and *dealt with automatically***
- **A *RuntimeException* (or anything inherited from it) is a special case**
  - **The compiler doesn't require an *exception specification* for these types**
  - **The output is reported to *System.err***

Special case: *RuntimeException* (Cont.)

- Keep in mind that only exceptions of type *RuntimeException* (and subclasses) can be ignored in your coding

- A *RuntimeException* represents a programming error
  - 1. An error you cannot anticipate. For example, a null reference that is outside of your control
  - 2. An error that you, as a programmer, should have checked for in your code (such as *ArrayIndexOutOfBoundsException* where you should have paid attention to the size of the array)
    - An exception that happens from point #1 often becomes an issue for point #2

- You can see what a tremendous benefit it is to have exceptions in this case, since they help in the debugging process

Performing cleanup with *finally*

- There's often some piece of code that you want to execute whether or not an exception is thrown within a *try* block
- To achieve this effect, you use a *finally* clause at the end of all the exception handlers

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

Performing cleanup with finally (Cont.)

- This program also gives a hint for how you can deal with the fact that exceptions in Java do not allow you to resume back to where the exception was thrown

```
1  class ThreeException extends Exception {}
2
3  public class FinallyWorks {
4    static int count = 0;
5    public static void main(String[] args) {
6      while(true) {
7        try {
8          // Post-increment is zero first time:
9          if(count++ == 0)
10           throw new ThreeException();
11         System.out.println("No exception");
12       } catch(ThreeException e) {
13         System.out.println("ThreeException");
14       } finally {
15         System.out.println("In finally clause");
16         if(count == 2) break; // out of "while"
17       }
18     }
19   }
20 }
```

```
/* Output:
ThreeException
In finally clause
No exception
In finally clause
*///:~
```

What's *finally* for?

- In a language without garbage collection and without automatic destructor calls, ***finally*** is important because it allows the programmer to guarantee the release of memory regardless of what happens in the ***try*** block
  - But Java has garbage collection, so releasing memory is virtually never a problem
- The ***finally*** clause is necessary when you need to set something other than memory back to its original state
  - This is some kind of cleanup like an open file or network connection, something you've drawn on the screen, or even a switch in the outside world

Using *finally* during *return*

- **Because a *finally* clause is always executed, it's possible to return from multiple points within a method and still guarantee that important cleanup will be performed**

```
1  import static net.mindview.util.Print.*;
2
3  public class MultipleReturns {
4    public static void f(int i) {
5      print("Initialization that requires cleanup");
6      try {
7        print("Point 1");
8        if(i == 1) return;
9        print("Point 2");
10       if(i == 2) return;
11       print("Point 3");
12       if(i == 3) return;
13       print("End");
14       return;
15     } finally {
16       print("Performing cleanup");
17     }
18   }
19   public static void main(String[] args) {
20     for(int i = 1; i <= 4; i++)
21       f(i);
22   }
23 }
```

```
Initialization that requires cleanup
Point 1
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
```

Pitfall: the lost exception

- It's possible for an exception to simply be lost
  - This happens with a particular configuration using a *finally* clause

```
1  class VeryImportantException extends Exception {
2    public String toString() {
3      return "A very important exception!";
4    }
5  }
6
7  class HoHumException extends Exception {
8    public String toString() {
9      return "A trivial exception";
10   }
11 }
12
13 public class LostMessage {
14   void f() throws VeryImportantException {
15     throw new VeryImportantException();
16   }
17   void dispose() throws HoHumException {
18     throw new HoHumException();
19   }
20   public static void main(String[] args) {
21     try {
22       LostMessage lm = new LostMessage();
23       try {
24         lm.f();
25       } finally {
26         lm.dispose();
27       }
28     } catch(Exception e) {
29       System.out.println(e);
30     }
31   }
32 }
```

```
/* Output:
A trivial exception
*///:~
```

Constructors

- "If an exception occurs, will everything be properly cleaned up?“
  - Most of the time you’re fairly safe, but with constructors there’s a problem
- The constructor puts the object into a safe starting state
  - But it might perform some operation—such as opening a file
  - If you throw an exception from inside a constructor, these cleanup behaviors might not occur properly
- *finally* is not the solution
  - If a constructor fails partway through its execution, it might not have successfully created some part of the object that will be cleaned up in the *finally* clause

Constructors (Cont.)

```
1  import java.io.*;
2
3  public class InputFile {
4    private BufferedReader in;
5    public InputFile(String fname) throws Exception {
6      try {
7        in = new BufferedReader(new FileReader(fname));
8        // Other code that might throw exceptions
9      } catch(FileNotFoundException e) {
10       System.out.println("Could not open " + fname);
11       // Wasn't open, so don't close it
12       throw e;
13     } catch(Exception e) {
14       // All other exceptions must close it
15       try {
16         in.close();
17       } catch(IOException e2) {
18         System.out.println("in.close() unsuccessful");
19       }
20       throw e; // Rethrow
21     } finally {
22       // Don't close it here!!!
23     }
24   }
```

```
25   public String getLine() {
26     String s;
27     try {
28       s = in.readLine();
29     } catch(IOException e) {
30       throw new RuntimeException("readLine() failed");
31     }
32     return s;
33   }
34   public void dispose() {
35     try {
36       in.close();
37       System.out.println("dispose() successful");
38     } catch(IOException e2) {
39       throw new RuntimeException("in.close() failed");
40     }
41   }
42 } ///:~
```

Constructors (Cont.)

- The safest way to use a class which might throw an exception during construction and which requires cleanup is to use ***nested try blocks***
- The basic rule is: Right after you create an object that requires cleanup, begin a ***try-finally***

```
1  public class Cleanup {
2    public static void main(String[] args) {
3      try {
4        InputFile in = new InputFile("Cleanup.java");
5        try {
6          String s;
7          int i = 1;
8          while((s = in.getLine()) != null)
9            ; // Perform line-by-line processing here...
10       } catch(Exception e) {
11         System.out.println("Caught Exception in main");
12         e.printStackTrace(System.out);
13       } finally {
14         in.dispose();
15       }
16     } catch(Exception e) {
17       System.out.println("InputFile construction failed");
18     }
19   }
20 }
```

Exception matching

- When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they are written
  - When it finds a match, the exception is considered handled, and no further searching occurs
- A derived-class object will match a handler for the base class

```
1  class Annoyance extends Exception {}
2  class Sneeze extends Annoyance {}
3
4  public class Human {
5    public static void main(String[] args) {
6      // Catch the exact type:
7      try {
8        throw new Sneeze();
9      } catch(Sneeze s) {
10       System.out.println("Caught Sneeze");
11     } catch(Annoyance a) {
12       System.out.println("Caught Annoyance");
13     }
14     // Catch the base type:
15     try {
16       throw new Sneeze();
17     } catch(Annoyance a) {
18       System.out.println("Caught Annoyance");
19     }
20   }
21 }
```

The compiler will give you an error message

```
/* Output:
Caught Sneeze
Caught Annoyance
*///:~
```

Thank you

zhenling@seu.edu.cn