Object – Oriented Design with UML and Java
Part VI: Exception Handling in Java
Exception Handling

- A false sense of security?
- Care must be taken if you intend for the program to continue running.
- Both C++ and Java support exceptions.
- Normal control flow (looping, etc.) and recursion are relatively easy to get right; Multi-threaded control flow is very difficult; Exception Handling looks easy, but it takes planning and effort to get right.
- Design you Exception Handling as part of a larger error logging, monitoring, diagnostics, and recovery infrastructure.
- Try to avoid throwing exceptions yourself, in most cases.
- Consider every plausible failure scenario.
- Is your system required to return meaningful error messages?
The Basic E.H. Model

- Java and C++ have subtle differences in their exception handling, but here, in the abstract, is what the two approaches have in common...

```java
try {
    throw new Exception( "Error!" );
    // Skip over this code
}

catch( Exception e ) {
    // Do something about the Exception...
}
```

The `throw` statement is like a `goto`; it jumps to the nearest matching `catch` without executing any intervening code, with the following exceptions:

- In Java, any code within an intervening `finally` block will get executed.
- C++ objects in stack memory will get destructed when they go out of scope. Note: do not throw an exception from a C++ destructor!
C++ Example

- Suppose a class `DatabaseException` is a subclass of `Exception`...

```cpp
try
{
    throw new DatabaseException( "Deadlock detected!" );
    cout << "The impossible has happened!" << endl;
}

catch( Exception* e ) // Nearest matching catch.
{
    cout << "Exception caught. " << endl;
}
```

- Outputs: Exception caught.
Java’s Checked Exceptions

The C++ language allows anything to be thrown.
Java only allows throwing subclasses of `java.lang.Throwable`.

- Java provides Compile-Time Support for “checked” exceptions:

- **Checked Exceptions** are subclasses of `java.lang.Exception`, excluding subclasses of `RuntimeException`.

- If there is a chance that a checked exception might get thrown from a given method, that method **must** either catch it, or declare it with a throws clause:

```java
public void myMethod() throws FooException {
    throw new FooException();
}
```
Java’s Exception Hierarchy

*Unchecked exceptions* include subclasses of Error and RuntimeException.

- RuntimeExceptions (such as NullPointerException and ClassCastException) are thrown by the Java runtime to indicate programming errors.

- Errors include: OutOfMemoryError, StackOverflowError, and ThreadDeath, ...

- Errors are difficult or impossible to recover from.
Catching Everything

```java
try {
    throw new DatabaseException( "Deadlock detected!" );
}
catch( DatabaseException dbe ) {
    retryAfterDeadlock();
}
catch( Throwable t ) // Beware of Errors.
{
    log.Error( "Unexpected Problem!" + t ); // see Log4j
}

● C++ also provides a way to catch all possible exceptions:
catch( ... ) {}
A More Realistic Example

try {
    dbConnection.beginTransaction();
    boolean result = doTransaction( dbConnection, info );
    dbConnection.commitTransaction();
    return result;
}

catch( Throwable t ) {
    // Better: catch expected exceptions
    dbConnection.rollbackTransaction();
    Log.log( "Transaction rolled back!", Severity.ERROR, t );
    return false;
}

finally {
    // This code gets executed no matter what.
    dbConnection.close(); // Frees system resources.
}
Exception Handling

It is not easy to write robust code in the presence of exceptions.

- Ensure that every function leaves its object in such a state that its destructor (C++) may be called, whenever any exception gets thrown.
- It is desirable, but sometimes very difficult, to ensure that if an exception gets thrown, the object is in the same state it was in before the function ever got called.
- At a minimum, make sure the object remains in a consistent state!

Note that in C++ when using templates, an exception can emanate from any operation on the template class. Be very careful!

```cpp
template < class X >
X TemplateClass< X >::copy( ) {
    return X; // Might throw!! Invokes X’s copy constructor!!
}
```
An alternative to using exception handling is to have every method that might fail return a boolean. (This technique cannot be used with a constructor).

```java
Vector< int > intVector;
boolean success = populateVector( intVector );
if( success ) { ...; } // Use the now-filled intVector.
boolean populateVector( Vector< int >& intVector ) {
    try {
        intVector.add( 1 );
    }
    catch( ... ) { return false; }
    return true;
}
```
A False Sense of Security

- Find the bug in the following Java code:

```java
class Foo {
    private int numFoos = 0;
    private int maxFoos = 10;
    private static FooList theFoos = new FooList( 10 );

    public void addFoo( Foo f ) throws FailedFooException {
        numFoos++;
        if( maxFoos < numFoos ) throw new FailedFooException( "Foo!" );
        theFoos.add( f );
    }
}
```
Null Pointers & Production Code

- Common programming error, even with experienced developers:
  ```java
  foo.bar();
  ```

- What if `foo` is `null`?

- In C++ the program will likely crash. In Java, it will throw a `NullPointerException`. Production code often looks more like this:

  ```java
  try {
    if( foo != null ) foo.bar();
  }
  catch( Exception ex ) { // Log and forget?
    Log.error( "Unexpected Error Caught!", ex );
    return false;
  }
  return true;
  ```
Exception Handling Policy

- In Java, throw `RuntimeExceptions` to indicate programming errors.
- Throw “checked exceptions” when the caller might be able to recover.
- Don’t catch an exception if you cannot recover, unless you are at the highest level of the call stack, such as in `main()` or `run()`. In these cases, catch all exceptions to ensure they are logged, for debugging.
- If you do catch an exception, log it. Ensure that there is a process for automatically utilizing the logged information. Log file phishing to notice application defects is not acceptable for production systems.
- Exceptions should be logged exactly once, by design. Consider Log4J.
- Include as much information as you can, to aid debugging.
- Consider designing an application-specific subclass of `RuntimeException` for containing additional information about what caused the exception.
- A common example of a recoverable exception is in transactional database code, where the DBMS, upon detecting deadlock, will kill one of the two deadlocked processes at random. The failed transaction can be retried
- If using an application framework such as `Spring`, then study their E.H. policy in detail.
Design By Contract

An interface may be thought of as a contract that the implementing class is making with all of the interface’s clients. “Design by Contract” is a theory that views an OO software system as collaborating components, whose interactions should strictly adhere to the terms of such contracts.

- Interfaces do not do a good job of defining semantics; and semantics should be well documented. Think like a lawyer and document as much as you can :^) Under what conditions might the software possibly fail?

Central to the theory of Design by Contract are pre-conditions & post-conditions for methods, and class invariants. It is easy to write code to check such conditions, and good quality production code is full of such checks.

- In C++, use assert statements liberally. assert( int expression ) is a macro (that gets “compiled out” in production code if NDEBUG is defined), which will print an “nice” error and abort the program if the expression evaluates to 0 (logically also: null & false). Use this (or similar Java code) to check your code’s assumptions. For example, assert( p ); will fail if p is a null pointer.