Input Validation Throwing Exceptions

Target Course

CS2 (object oriented)

Learning Goals

A student shall be able to:

1. Apply principles of secure design and defensive programming techniques when developing software.

IAS Outcomes

The CS2013 Information Assurance and Security outcomes addressed by this module are:

IAS Knowledge Topic	Outcome
Defensive Programming	3. Explain the risks with misusing interfaces with third-party code and how to correctly use third-party code. [Familiarity] 5. Demonstrate the identification and graceful handling of error conditions. [Usage]

Dependencies

- Assumes knowledge of exceptions, using try/catch blocks.
- Assumes knowledge of checked and unchecked exceptions (if using Java).
- Assumes familiarity with building a data structure.

Summary

Discuss best practices in handling errors by throwing exceptions.

Estimated Time

[Provide the estimated amount of lecture time to cover this module, using the notion of time as defined in CS2013.]

Materials

When is exception throwing used?

Throwing an exception is an alternative way to handle errors (to printing out error messages and stopping execution). When an error condition is detected we might choose to create a new exception and *throw* it. The exception is propagated down the stack of calling methods and if it is not caught it causes the program to crash and terminate.

What is an example of throwing an exception?

Suppose you are creating a data structure that stores key-value pairs (e.g. a hash table, tree, etc.), where each key is unique. In the following example the `add` method throws a `DuplicateKeyException` if a duplicate key already exists in the data-structure. Note that `DuplicateKeyException` is defined as a checked exception in a separate class (i.e. it is not part of the Java standard library). Since `DuplicateKeyException` is checked, the `add` method must declare that it throws the exception in its header. Additionally any method that calls `add` will be required by the compiler to *handle* the `DuplicateKeyException`, either by catching it or by declaring in its header that it too throws a `DuplicateKeyException`, as the method in the Client class does in the example below.

```
public class DemoDataStructure<K, V>
{
  public void add(K key, V value)throws DuplicateKeyException
  {
      if(this.contains(key))
            throw new DuplicateKeyException(“Key already exists”);
  }
  ... //assume more code here
```

```
public class DuplicateKeyException extends Exception
{
  public DupicateKeyException(String msg)
  {
    super(msg);
  }
}
```

```
public class Client
{
  public void useStrcuture throws DuplicateKeyException
  {
    DemoStrucuture d = new DemoStructure<String, String>();
    d.add(“A”, “Value1”);
    d.add(“A”, “Value2”);
    ... //assume more code here
  }
  ... //assume more code here
}
```

When should an exception be thrown versus simply reporting an error in some other manner?

You may have noticed that the example above did not have to throw an exception to deal with a duplicate key. The situation might have been handled using a conditional to detect the duplicate key, report the error and reject the addition of the pair. This is a design choice..

Are there any differences between throwing a checked or unchecked exception?

If the exception thrown is an unchecked exception, the throwing method (`add` in the above example) would not need to declare the exception in its header. The client method `useStructure` would also not be required to handle it by the compiler. Thus unchecked exceptions require less specification.

On the other hand, checked exceptions provide a more formal way to let calling methods know about the exception as a possible output value.

When should an unchecked exception be thrown versus a checked exception?

According to the oracle java tutorial,

> “*If a client (or calling method) can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.* “

Note that Java is one of the few programming languages that provide checked exception. In most other languages all exceptions are *unchecked.* Thus it is important for the method throwing the exception to document this as a possible output and for client methods to read the documentation.

What are some best practices for error handling with using 3rd-party interfaces?

We might think of `DemoStructure` as a 3rd party interface. Using 3rd party code responsibly includes taking the time to understand all the different outputs including error codes or exceptions which may be thrown by the 3-rd party code and handle them appropriately. Typically this will involve reading the interface's documentation. Failure to handle notices of errors from 3rd party code can lead to unstable environments and security vulnerabilities.

One type of attack takes advantage of file system API by feeding in a string that contains leading ghost characters [3]. One example is a triple-dot vulnerability where the string `.../../../winnt` is provided to the API that ultimately gives access to a protected area on the hard drive. In this type of attack the triple dot causes the API code to not recognize an attempt at doing a directory traversal attack.

Assessment Methods

[List the assessment methods that have been used to assess student learning for this module. The format of these methods is fairly flexible, but should be applied consistently within the module.]

References

[1] The Java Tutorials https://docs.oracle.com/javase/tutorial/essential/exceptions. Aug 2015.

[2] Best Practices for Exception Handling. Oreilly, On Java.com. http://archive.oreilly.com/pub/a/onjava/2003/11/19/exceptions.html Accessed Aug 2015.

[3] Hoglund G. and McGraw G. (2004). Exploiting Software: How to Break Code. Addison Wesley.