CSCI 2120:
Software Design & Development II

Defensive Programming

Overview

- Defensive Programming
  - Why program defensively?
  - Encapsulation
  - Access Restrictions
  - Defining & Identifying DP
- Asserts
- Exceptions
  - Unchecked Exceptions
  - Checked Exceptions

Defensive Programming - Motivation

- Normally, your classes will form part of a larger system
- So other programmers will be using and relying upon your classes
- Obviously, your classes should be *correct*, but equally importantly, your classes should be *robust* – that is, resistant to accidental misuse by other programmers
- You should aim to ensure that no errors in the final system can be attributed to the behaviour of your classes
- We use the terminology "*client code*" for the code written by other programmers that is using your classes

Defensive Programming - Encapsulation

- One of the most important features of OOP is that it facilitates *encapsulation* – a class encapsulates both the data it uses, and the methods to manipulate the data
- The external user *only* sees the public methods of the class, and interacts with the objects of that class purely by calling those methods
- This has several benefits
  - Users are insulated from needing to learn details outside their scope of competence
  - Programmers can alter or improve the implementation without affecting any client code

Defensive Programming - Access Restriction

- Encapsulation is enforced by the correct use of the access modifiers, public, private, <default>, and protected
- If you omit the access modifier, then you get the default, sometimes known as “package”
- These latter two modifiers are only really relevant for multi-package programs that use inheritance, so we need only consider public and private at the moment

Defensive Programming

Defensive programming is a companion to Design by Contract.

- Design by Contract establishes the Preconditions & Postconditions between the Implementer & the client developer. If the client violates the preconditions, then the implementer guarantees nothing.

- Defensive programming checks the client's request and alerts them if they violated the preconditions as a courtesy.

Defensive Programming

Design by Contract is complementary to defensive programming because

- With preconditions, it makes clear which inputs (to methods) are unexpected.
- With postconditions, it makes it clear when an internal bug has occurred.
- But it does not prescribe predictable behaviour in the face or unexpected inputs and internal errors.

Defensive Programming

- Defensive programming is a loosely defined collection of techniques to reduce the risk of failure at run time.
- One technique is “Making the software behave in a predictable manner despite unexpected inputs or user actions.”
- Related: Making the software behave in a predictable manner despite internal errors (bugs).

Defensive Programming

- Consider a search routine

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a )
```

Defensive Programming

- Bob implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    return k ;
}
```

Defensive Programming

- Chris implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
   assert a != null ;
   int k = 0 ;
   while( k < a.length && a[k] != x ) ++k ;
   assert k == a.length || a[k] == x ;
   return k ;
}
```

Throws an exception if condition is false and assertion checking is enabled

Defensive Programming

- Dan implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    Assert.check( a != null , "'search' precondition failed");
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    Assert.check( k == a.length || a[k] == x , "'search' postcondition failed") ;
    return k ;
}
```

Defensive Programming

- Dan's Assert class looks like this

```
class Assert {
    static void check( boolean cond, String message) {
      if( ! cond ) throw new AssertionError( message) ; }
}
```

Defensive Programming

- Eve implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    if( a == null ) return 0 ;
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    return k ;
}
```

Defensive Programming

- Bob, Chris, Dan and Eve all wrote code that meets the contract.
- Bob was not practicing defensive programming
- Chris and Dan were practicing defensive programming.
- Eve was practicing poor programming! If you take the time to check a precondition, it is better to make someone aware of the failures.

Defensive Programming - Fail Fast Programming

- Defensive checks (such as assertions) are analogous to fuses in a power circuit.
- They cause erroneous systems to “fail fast”. I.e. to fail before further damage is done.
- They also help pinpoint the root cause of a fault.
- A safety critical system should also “fail safe”. The combination of fail fast, fail safe, fault tolerance (recovery from failure), and failure reporting is the best.
- Eve’s solution masks the earlier error and is a “garbage in – garbage out” solution.

Defensive Programming - Partial checks vs Full checks

- Note that Chris and Dan did not check the postcondition, rather they checked an implication of the postcondition. (A “partial check”.)
- Whether it is worth the computational and design costs to check the full pre- or postcondition is a function of many inputs
  - The confidence in the code.
  - The cost of error.
  - The cost of a partial check vs. a full check
  - The sufficiency of a partial check vs. a full check.

Defensive Programming - DP & Contracts

- Defensive programming is complementary to the use of contracts.
- A contract obviously guides the writing of run-time defensive checks.
- A defensive check helps ensure that the contract is being respected.
- Systems such as JML, Spec#, and .NET Contracts can automatically turn contracts into run-time defensive checks.

Defensive Programming - Valid values vs Legal values

We use defensive programming as API implementers to ensure that the state of our data models remains valid.

Consider the following problem, whereby we design a class Date to model a calendar date that consists of a day, month, year.

If we use integers to hold the day, month, year values. Then the valid bounds for a day value is more restrictive than the bounds of a legal int value.

For instance, it makes no sense to have a day -17 or day 70081 despite that those are both legal integer values.

Defensive Programming - Valid values vs Legal values

```
int month;
```

month vs integer

- integer: -2147483648 to 2147483647.
- month: 1 to 12

Defensive Programming - Assert or Exception?

What is the difference between Java exception handling and using assert conditions?

When should we use Exceptions and when should we use the assert keyword?

Defensive Programming - Assert or Exception?

In Java, which is more highly recommended, and why?

```
public void doStuff(Object obj) {
    assert obj != null;
    ...
}
```

vs.

```
public void doStuff(Object obj) {
    if (obj == null) {
        throw new IllegalArgumentException("object was null");
    }
    ...
}
```

Defensive Programming - Assert or Exception?

Short Answer:

"**Unchecked Exceptions** are designed to detect programming errors of the users of your library, while **assertions** are designed to detect errors in your own logic."

Defensive Programming - Assert or Exception?

Long Answer:

Assertions are removed at runtime unless you explicitly specify to "enable assertions" when compiling your code. Java Assertions are not to be used on production code and should be restricted to private methods, since private methods are expected to be known and used only by the developers. Also assert will throw AssertionError which extends Error not Exception, and which normally indicates you have a very abnormal error (like "OutOfMemoryError" which is hard to recover from, isn't it?) you are not expected to be able to treat.

Assert

Java assert statement provides some support for defensive programming.

```
assert i > 0 ;
```

means

```
{if( !(i>0) ) throw new AssertionError() ; }
```

if the program is run with assertions enabled.

The JVM parameter “–ea” will enable assertions.

Assert

However when a Java program is run without assertions enabled, assert statements have no effect.

Whereas, Exceptions take effect at all times

Exceptions

- Even if your classes are well-protected, errors still occur
  - Client code attempts to use your methods incorrectly, by passing incorrect or invalid parameter values
  - Your code cannot perform the services it is meant to due to circumstances outside your control (such as an Internet site being unavailable)
  - Your own code behaves incorrectly and/or your objects become corrupted
- Java provides **exceptions** (checked and unchecked) to handle these situations

Exceptions - Motivation

- **Idea: exceptions can represent unusual events that client could handle (as well as errors)**
  - **Finite data structure is full; can't add new element**
  - **Attempt to open a file failed**
  - **Network connection dropped in the middle of a transfer**
- **Problem: the object that detects the error doesn't (and probably shouldn't) know how to handle it**
- **Problem: the client code could handle the error, but isn't in a position to detect it**
- **Solution: object detecting an error throws an exception; client code catches the exception and handles it**

Exceptions - The method "throws" an Exception

- If a parameter is invalid, then the method cannot do anything sensible with the request and so it creates an object from an Exception class and “throws” it
- If an Exception is thrown, then the runtime environment immediately tries to deal with it
  - If it is an *unchecked* exception, it simply causes the runtime to halt with an error message
  - If it is a *checked* exception, then the runtime tries to find some object willing to deal with it
- The method `charAt` throws a `StringIndexOutOfBoundsException`

  which is unchecked and hence causes the program to cease execution (crash!)

Exceptions - Throw your own Exceptions

- Your own methods and/or constructors can throw exceptions if clients attempt to call them incorrectly
- This is how your code can enforce rules about how methods should be used
- For example, we can insist that the deposit and withdraw methods from the BankAccount class are called with positive values for the amount
- The general mechanism is to check the parameters and if they are invalid in some way to then
  - *Create* an object from class IllegalArgumentException
  - *Throw* that object

Exceptions - Throw your own Exceptions

```
Public BankAccount(int amount) {
  if (amount >= 0) {
   balance = amount;
  } else {
   throw new IllegalArgumentException(
       "Account opening balance " +
        amount + " must be >0”);
  }
}
```

- If the amount is negative then *declare* the variable ie, *create* the object and then *throw* it
- The constructor for IllegalArgumentException takes a String argument which is used for an error message that is returned to the user
- Throwing an exception is often used by **constructors** to prohibit the construction of invalid objects

Exceptions - Predictable Errors

- **Unchecked** exceptions terminate program execution and are used when the client code must be seriously wrong
- However, there are error situations that do not necessarily mean that the client code is incorrect, but reflect either a transient, predictable or easily-correctable mistake – this is *particularly* common when handling end-user input, or dealing with the operating system
- For example, printers may be out of paper, disks may be full, Web sites may be inaccessible, filenames might be mistyped and so on.

Exceptions - Checked Exceptions

- Methods prone to such errors may elect to throw **checked** exceptions, rather than unchecked exceptions
- Using checked exceptions is more complicated than using unchecked exceptions in two ways:
  - The programmer must *declare* that the method might throw a checked exception, and
  - The client code using that method is *required* to provide code that will be run if the method *does* throw an exception

Exceptions - Client Perspective

- Many of the Java library classes declare that they *might* throw a checked exception

public **FileReader**(File file) throws FileNotFoundException

Creates a new FileReader, given the File to read from.

Parameters:

file - the File to read from

Throws:

FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

Exceptions - try and catch

- **Basic syntax**

```
try {
    somethingThatMightBlowUp( );
} catch (Exception e) {
    recovery code – here e, the exception object, is a "parameter"
}
```

- **Semantics**
  - **Execute try block**
  - **If an exception is thrown, terminate throwing method and all methods that called it, until reaching a method that catches the exception (has a catch with a matching parameter type)**
  - **Catch block can either process the exception, re-throw it, or throw another exception**

Exceptions - try and catch

- **Can have several catch blocks**

```
try {
    attemptToReadFile( );
} catch (FileNotFoundException e) {
    …
} catch (IOException e) {
    …
} catch (Exception e) {
    …
}
```

  - **Semantics: actual exception type compared to exception parameter types in order until a compatible match is found**
  - **No match – exception propagates to calling method**

Exceptions - try and catch

- If code *uses* a method that might throw a checked exception, then it *must* enclose it in a try/catch block

```
try {
  FileReader fr = new FileReader("lect.ppt");
   // code for when everything is OK
  } catch (java.io.FileNotFoundException e) {
   // code for when things go wrong
}
```

- *Try* to open and process this file, but *be prepared* to *catch* an exception if necessary

Exceptions - try and catch

- If everything goes smoothly, the code in the try block is executed, the code in the catch block is skipped
- Otherwise, if one of the statements in the try block causes an exception to be thrown, then execution immediately jumps to the catch block, which tries to recover from the problem
- What can the catch block do?
  - For human users: report the error and ask the user to change their request, or retype their password, or …
  - In all cases: Provide some feedback as to the likely cause of the error and how it may be overcome, even if it ultimately it just causes execution to cease

Exceptions - try and catch and finally

- **One last wrinkle: finally**

```
try {
    …
} catch (SomeException e) {
    …
} catch (SomeOtherException e) {
    …
} finally {
    …
}
```

  - **Semantics: code in the finally block is *always* executed, regardless of whether we catch an exception or not**
  - **Useful to guarantee execution of cleanup code no matter what**

Exceptions - try and catch and finally

```
// demonstrate try...catch...finally
public static void throwException() throws Exception {
   try { // throw an exception and immediately catch it
      System.out.println("Method throwException");
      throw new Exception(); // generate exception
   }
   catch (Exception exception) { // catch exception thrown in try
      System.err.println(
         "Exception handled in method throwException");
      throw exception; // rethrow for further processing

      // code here would not be reached; would cause compilation errors

   }
   finally { // executes regardless of what occurs in try...catch
      System.err.println("Finally executed in throwException");
   }

   // code here would not be reached; would cause compilation errors
}
```

Exceptions - Using & Testing Exceptions

```java
@Test
void testSeconds() {
    IllegalArgumentException thrown = assertThrows(IllegalArgumentException.class, () -> {
        //Code under test
        Time time = new Time( hours: 1, minutes: 1, seconds: 100);
    });
    assertEquals( expected: "seconds must be 0-59", thrown.getMessage());
}
```

- Java provides a many exception classes that cover most common possibilities
- Exceptions are simply objects in a Java program, so you can write your own classes of exceptions if desired

Exceptions - Some Useful Exceptions

- IllegalArgumentException
- IndexOutOfBoundsException
- NullPointerException
- ArithmeticException
- IOException, FileNotFoundException

Exceptions - Some Useful Exceptions

- **When we discussed programming by contract, we described how to throw an exception to indicate an error (precondition not met or other reason)**

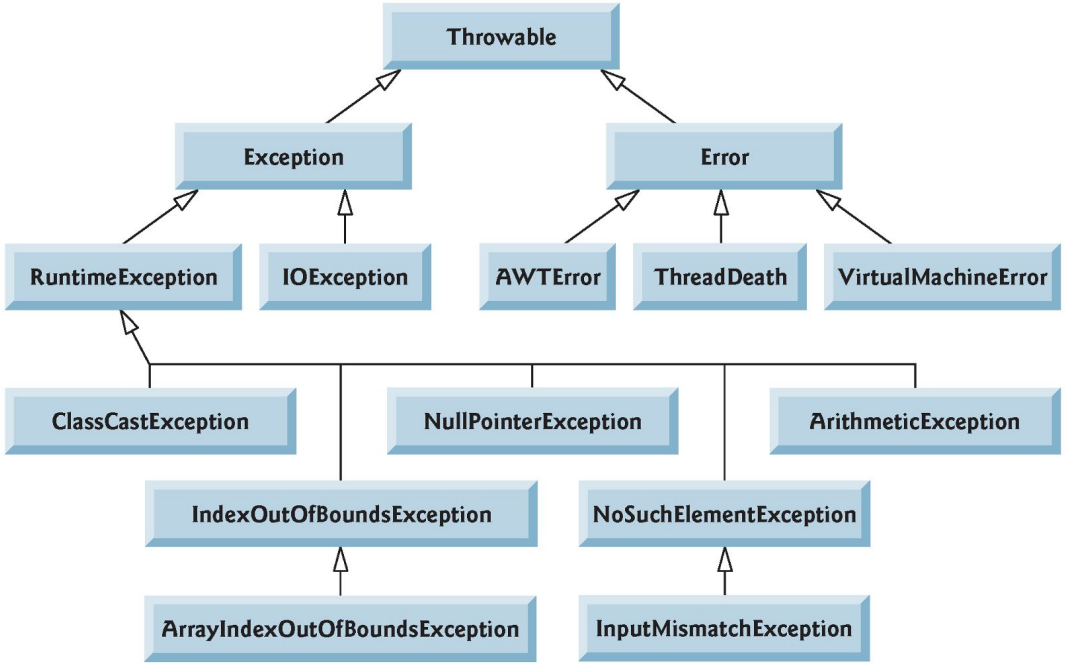
```
if (argument == null) {
    throw new NullPointerException( );
}

if (index < 0 || index > size) {
    throw new IndexOutOfBoundsException("No such item");
}
```

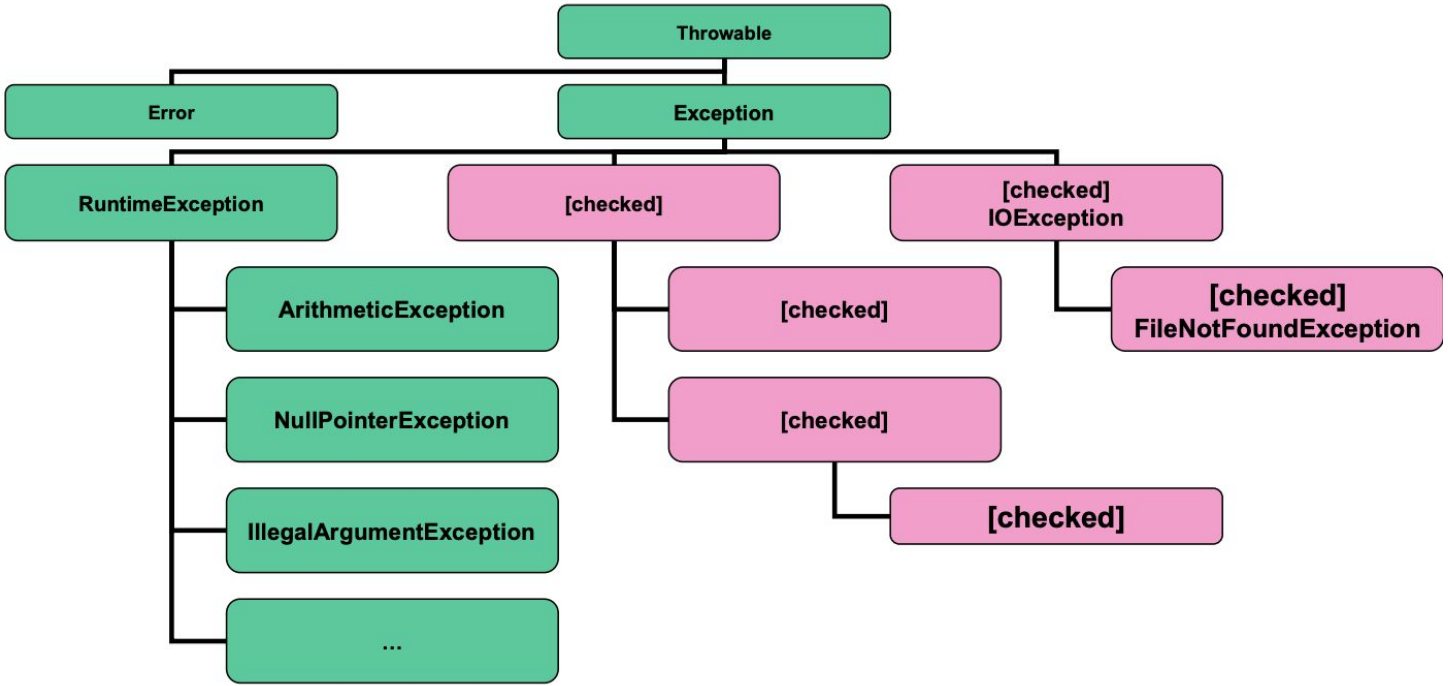
Exceptions - Exception vs RuntimeException

- Checked exceptions in Java extend the java.lang.Exception class
- Unchecked exceptions extend the java.lang.RuntimeException class

Exceptions - Exception vs Error



Exceptions - RuntimeException vs checked



Exceptions - Programmer Perspective

- If you choose to write a method that throws a checked exception, then this must be *declared* in the source code, where you must specify the *type* of exception that might be thrown

```
public void printFile(String fileName) throws
  java.io.FileNotFoundException {
  // Code that attempts to print the file
}
```

- If you declare that your method might throw a checked exception, then the compiler will *force* any client code that uses your method to use a try/catch block
- This explicitly makes the client code responsible for these situations

Exceptions - Checked or Unchecked ?

- Unchecked Exceptions
  - Any method can throw them without declaring the possibility
  - No need for client code to use try/catch
  - Causes execution to cease
  - Used for fatal errors that are unexpected and unlikely to be recoverable
- Checked Exceptions
  - Methods must declare that they might throw them
  - Client code must use try/catch
  - Causes control flow to move to the catch block
  - Used for situations that are not entirely unexpected and from which clients may be able to recover
  - Use only if you think the client code might be able to do something about the problem

Exceptions - Summary

- Programming defensively means making your code **robust** to unexpected use.
- Use the **need to know** principle: Only expose the parts of your class that your client classes need to know
- Java exceptions provide a uniform way of handling errors
- Exceptions may be Unchecked or Checked

Exceptions - Summary

- **Intended for unusual or unanticipated conditions**
  - **Relatively expensive if thrown (free if not used)**
  - **Can lead to obfuscated code if used too much**
- **Guideline: Use in situations where you are in a position to detect an error, but only client code would know how to react**
- **Guideline: Often appropriate in cases where a method's preconditions are met but the method isn't able to successfully establish postconditions (i.e., method can't do what is requested through no fault of the caller)**