Comp-303 : Programming Techniques
Lecture 7

Alexandre Denault
Computer Science
McGill University
Winter 2004

Announcements

- How was the GUI tutorial?
- Both midterm will be held in Leacock 26.
- Promise to update the website tonight or tomorrow.
  - Add the Tutorial slides
  - Update the course slides
  - Add the T.A. email
  - Put the grading scheme for the project

Last lecture . . .

- Data Abstraction allows us to seperate the external interface of an object from its inner working.
- When successful, Data Abstraction allows us to modify the implementation of an object without modifying the other objects using it.
- Differences between mutable and immutable objects.
- Examples with IntSet and Poly.
- Some methods from object may need to be overrided:
  - equals()
  - similar()
  - hashCode()
  - clone()
  - toString()

Public/Protected/Private (last lecture)

Java	Interfaces, Classes	Variables, Methods, Inner-Classes
Public	visible everywhere	visible everywhere, inherited by all subclasses
Protected	-	visible to any class in the package, inherited by all subclasses in or outside the package
blank	visible only in current package	visible to any class in the package, inherited by all subclasses in the package but not outside package
Private	-	not visible to any class, not inherited by any class

Violating the REQUIRES clause

- What happens if the REQUIRES clause is not met ?

```
public static int gcd (int n, int d) {
  // REQUIRES: n, d > 0
  // EFFECTS: Returns the greatest common divisor of n and d

  while (n != d)
    if (n > d)
      n = n  - d;
    else
      d = d - n;

  return n;
}
```

Solution #1: do nothing

- *Partial implementation*: "it's up to the user to call gcd correctly".
- What happens if the user does call gcd with d or n $= 0$? less than 0 ?
- Partial procedures lead to programs that are not robust.
- A *robust* program continues to behave reasonably in the presence of errors:
  - At best: provide an approximation of error-free behavior. a.k.a. Graceful degradation
  - At worst: halt with a meaningful error message without causing damage to permanent data.

Solution #2: return special result

- Return 0: 0 cannot be the result of a correctly called gcd, so can be used as a special result.
- What happens if you return 0 as the result of gcd ?
  - Now the caller must check for the special result. This is inconvenient.
- Sometimes the whole range of return values is legal, so there is no *special* result to return.
- For example, the get method in Vector returns the value of the vector's ith element (either null or Object).
  - There are no value to convey that the index out of bounds.

Solution #3: use an exception

- Exceptions signifies that something unusual occurs.
- They are separated from normal control flow:
  - A() calls B() calls C() calls D()
  - D throws an exception
  - A catches the exception
  - All the remaining code in D,C and B is skipped
- The use of exceptions is supported by Java Exception types.

Specifications

- Procedure that can terminate exceptionally has a throws clause in the header:

```
public static int fact (int n) throws NonPositiveException
```

- More than one exception can be thrown:

```
public static int search (int [ ] a, int x)
  throws NullPointerException, NotFoundException
  // EFFECTS: if a is null throws NullPointerException
  //  else if  x is not in a throws NotFoundException
  //  else returns i such that a[i] == x
```

- Header lists exceptions that are part of specified behavior.
- The Effects clause lists the exceptions and the action in each case.
- Termination of a function by throwing an exception is ok when you have an error.

Exception types

- Exceptions are objects.
- As such, they exist in the object hierarchy:

```
Object
   Throwable
      Error
         Exception
            ...
            (new checked exceptions)
            ...
            RuntimeException
               ...
               (new unchecked exceptions)
               ...
```

Constructing Exception Types

- A simple exception, only type as information:

```
Exception e1 = new myException();
```

- An exception with some extra info:

```
Exception e2 = new myException(
 "here is where and why this exception occurred");
```

- Or even passing an object:

```
Exception e3 = new myException(
 "here is where and why this exception occurred",
 someObjectWhichCausedTheException);
```

Defining Exceptions

- Usually, the following is enough:

```
public class myException extends Exception {}
```

- This is equivalent to:

```
public class myException extends Exception {
   myException() { super();}
   myException(String s) {super(s);}
}
```

- Exception be me much more elaborate:

```
public class myException extends Exception {
    Object offensiveObject;
    myException(String s; Object o)
       {super(s);offensiveObject = o;}
  Object getOffensiveObject() {return offensiveObject;}
}
```

- Java does not enforce ???Exception naming, but is good practice.

Where to define exception types ?

- Some Exceptions occur in many packages
  (ex: NotFoundException).
- It makes sense to avoid naming conflicts and define a separate Exception package.
- However, if special Exceptions are thrown by your package, you can define them in your package.

Throwing Exceptions

```
public static int fact (int n) throws NonPositiveException {
   ...
   if (n <= 0) throw
      new NonPositiveException(
         "Num.Fact: Used a negative number for n");
   ...
}
```

- What to put in the string ?
- Something that conveys information about what went wrong: minimally the class and method that threw the Exception.
- Note that many methods may throw the same exception.

Catching Exceptions

```
try { x = Num.fact(y); }
catch (NonPositiveException e) {
  // in here, use e
  System.out.println(e);
}
```

- This code handles the exception explicitly.
- If a NonPositiveException is thrown anywhere within the try block, execution proceeds at the start of the catch block, after the thrown exception is assigned to e.

Variations on catching

- Multiple catch clauses can be used to handle different type of exceptions.

```
try { x.foobar() }
catch (OneException e) { ... }
catch (AnotherException e) { ... }
catch (YetAnotherException e) { ... }
```

- You can also use nested try clauses.
- Here, the inner try block throws anotherException, It is handled by the outer catch clause.

```
try { ...
  try { ... throw new anotherException(); ...}
  catch (SomeException e) { .. throw new anotherException(); ...}
... }
catch (anotherException e) { ... }
```

Exceptions & subtypes

```
try { ...
 throw new oneException();
 ...
 throw new anotherException();
 ...
} catch (Exception e) { ... }
```

- This catch clause will catch all exceptions occurring within the try block.
- Pretty weird control flow are possible, especially when using the Finally clause.
- Don't abuse exceptions for fancy control flow:
  catch clauses are usually implemented inefficiently because they are *exceptional*

Catching Exceptions: try syntax

```
try {
   statements
} catch (exception_type1 identifier1) {
   statements
} catch (exception_type2 identifier2) {
   statements
} finally {
   statements
}
```

- First, the code within try block is executed.
- If an exception is thrown, the execution of the statements in try block is stopped.
- The catch statements will be executed if the exception type matches.
- Statements in the finally block is always executed, even if the exception is not caught.

Exception types

- Checked exceptions (extends exception)
  - must be listed in the throws clause of the called procedure that throws the exception.
  - must be handled by the caller code either by
    - propagation
    - catching
  - otherwise, we get a compile-time error.
  - Example: IOException
- Unchecked exceptions (extends RuntimeException)
  - don't have to be listed
  - don't have to be handled
  - Example: NullPointerException, IndexOutofBoundsException

Handling Exceptions Implicitly

- If the caller calls a procedure without a try clause then the thrown exception is propagated to the caller of the caller.
  (A calls B calls C calls D throws e, A catches e)
- To propagate, the caller must list the thrown exception
  (checked by compiler).
- Unchecked exceptions are automatically propagated until they reach an appropriate catch clause
  (not checked by compiler)
- But you can still list them in the header and it is good practice to do so
  (so the user is aware of the unchecked exception and can catch it if desired)

Programming with exceptions

- How to handle thrown exceptions ?
- Possibilities:
  - *handle* specifically: separate catch clauses deal with each situation in a different way
  - *handle* generically: one catch clause for supertype exception takes generic action like println and halt or restart program from earlier state
  - *reflect* the exception: the caller also terminates by throwing an exception by propagation or by throwing a different exception (usually better)
  - *mask* the exception: the caller handles the situation and continues with normal flow

Reflection

```
public static min (int [ ] a)
   throws NullPointerException, EmptyException {
   // EFFECTS: if a is null throws NullPointerException else
   // if a is empty throws EmptyException
   // else returns minimum value of a

   int m;

   try {
      m = a[0];
   } catch (IndexOutOfBoundsException e) {
      throw new EmptyException(Arrays.min);
   }

   for (int i = 1; i < a.length; i++)
      if (a[i] < m) m = a[i];

   return m;
}
```

Masking

```
public static boolean sorted (int [ ] a)
   throws NullPointerException {
   // EFFECTS: if a is null throws NullPointerException else
   // if a is sorted in ascending order returns true else
   // returns false

   int prev;

   try { prev = a[0]; }
   catch (IndexOutOfBoundsException e) {
      return true; }

   for (int i = 1; i < a.length; i++)
      if (prev <= a[i])
         prev = a[i];
      else
         return false;

   return true;
}
```

Design issues

- When to use exceptions ?
  - To replace REQUIRES: clauses and make a procedure total instead of partial.
  - To avoid encoding information in ordinary results.
  - To signal special, usual erroneous situations in non-local use of procedures.
- When not to use exceptions ?
  - When the context of use is local
  - When the REQUIRES: clause would be too expensive or impossible to check

Exceptions are exceptional

- Exceptions should not be thrown as a result of normal use of a program.
  - Catching exceptions is not efficient.
  - You should be able to ignore exceptions when you read the code to understand regular behavior.
  - There are other ways to discontinue execution in non-exceptional cases:
    - break: terminates switch, for, while, do
    - continue: skips to end of loop body
    - return: terminates method

Checked versus Unchecked

- Advantages of checked exceptions:
  - Provide protection:
    - compiler warns about exception that are not caught
    - forces the programmer to catch them / deal with them
  - Prevents wrongly captured exceptions
- Disadvantages of checked exceptions:
  - Forces the user to deal with them even if he is 100% sure that they cannot occur.
- Use unchecked exceptions if you expect they will not occur because they can be conveniently and inexpensively avoided.
- Otherwise, use checked exceptions.

Defensive programming

- Defensive programming is checking for errors that are not supposed to occur (introduced by other procedures, hardware, user). These situations are not described by in the procedure specifications.
- Exceptions are good for this, because they provide a means of conveying information about errors and handle errors without cluttering the main flow.
- Typically, you could use one generic unchecked exception. e.g. FailureException
- In the following code fragment, the user is sure that NotFoundException cannot occur. However, it is a checked exception, so it must be caught:

```
try {Search(a,x); }
catch (NotFoundException e) {
    throw new FailureException(
       "thisClass.thisProcedure" + e.toString()); }
```

Summary

- Exceptions are thrown under exceptional conditions. The should not be used for regular control flow.
- Exceptions move conditions from REQUIRES clause to EFFECTS clause.
- Checked exceptions are declared in method header and must be caught or propagated by caller.
- Unchecked exceptions do not have to be declared and are propagated automatically.
- A caught exception can be reflected or masked.
- When using Defensive Programming, all sources of errors must be checked, even the unlikely and the impossible.

Tool of the day: Nokia 5100

- The Nokia 5100 is a Java-enabled cell-phone.
- It uses the Java MIDP (Mobile Information Device Profile) virtual machine.
- Developers can use the J2ME Wireless Toolkit to create application that will work on this cell phone.
- Developing for specialized hardware (such as a cellphone) is very different from developing for a personal computer.