Advanced Object-oriented Programming

Lecture 18

Exceptions
Exceptions

We’ve seen that exceptions generally arise from input/resource failures.

Semantic coding errors (bugs) can go unnoticed until some input is of the wrong form (ArithmeticException), or until some ‘resource’ isn’t available (NullPointerException, ArrayIndexOutOfBoundsException).

When this happens, a RuntimeException is thrown.

These are generally unforeseeable (because they arise from bugs).
Runtime Exceptions

The best way of recovering from these is to put a try-catch block at the top level (e.g., in the main() method) with a general error message:

```java
public static void main(String[] args) {
    try {
        // all the code
    } catch (RuntimeException re) {
        // report the error
        // exit gracefully
    }
}
```
Other Exceptions

There are other subclasses of Exception (but not of RuntimeException) that are useful for other specific kinds of input/resource failure.

For example, the IOException class ‘signals that an I/O exception of some sort has occurred.’

This class has even more specific subclasses that include:

- FileNotFoundException
- MalformedURLException
- RemoteException
Checked Exceptions

These exceptions are not expected in the ‘normal running of the Java interpreter’.

They signal resource errors (e.g., disk errors, unexpected user-input) that the programmer cannot ignore.

They are treated very differently from the unchecked exceptions:

whenever it is possible that a method might throw one of these exceptions, the programmer must either catch them (in a try-catch block), or advertise the fact that the method might throw one of these exceptions.

These are called checked exceptions

...and the compiler will check that programmers either catch or advertise.
Checked Exceptions

These exceptions are not expected in the ‘normal running of the Java interpreter’.

They signal resource errors (e.g., disk errors, unexpected user-input) that the programmer cannot ignore.

They are treated very differently from the unchecked exceptions:

whenever it is possible that a method might throw one of these exceptions, the programmer must either catch them (in a try-catch block), or advertise the fact that the method might throw one of these exceptions.

These are called checked exceptions...

...and the compiler will check that programmers either catch or advertise.
Checked Exceptions

These exceptions are not expected in the ‘normal running of the Java interpreter’.

They signal resource errors (e.g., disk errors, unexpected user-input) that the programmer cannot ignore.

They are treated very differently from the unchecked exceptions:
whenever it is possible that a method might throw one of these exceptions, the programmer must either catch them (in a try-catch block), or advertise the fact that the method might throw one of these exceptions.

These are called checked exceptions

... and the compiler will check that programmers either catch or advertise.
Checked vs Unchecked

NB: unchecked = subclass of RuntimeException

checked = *not* a subclass of RuntimeException
Advertising Exceptions

Checked exceptions must either be caught (‘handled’) or advertised.

For example, the class `java.io.BufferedInputStream` has the following method:

```java
public int read() throws IOException { ... }
```

(which reads Input values (from keyboard, filestore, network connection, etc.)

Any method that calls this method must either catch any IOExceptions,

or advertise that it may throw such exceptions

(in the same way that `read()` itself advertises this).
Advertising Exceptions

Checked exceptions must either be caught (‘handled’) or advertised.

For example, the class `java.io.BufferedInputStream` has the following method:

```java
public int read() throws IOException {
    ... }
```

(which reads Input values (from keyboard, filestore, network connection, etc.)

Any method that calls this method must either catch any `IOException`,
or advertise that it may throw such exceptions
(in the same way that `read()` itself advertises this).

Advertising Exceptions

Checked exceptions must either be caught (‘handled’) or advertised.

For example, the class `java.io.BufferedInputStream` has the following method:

```java
public int read() throws IOException {
    ... }
```

(which reads Input values (from keyboard, filestore, network connection, etc.)

Any method that calls this method must either catch any IOExceptions,
or advertise that it may throw such exceptions (in the same way that `read()` itself advertises this).
For Example: Catching

Here's a method that calls `read()` and catches any `IOException` that might be thrown.

```java
public int readInt(BufferedInputStream b) {
    try {
        return b.read();
    } catch (IOException ioe) {
        return -1;
    }
}
```

(For example, the program that uses this method might take a value of -1 as an indication of an error.)
For Example: Passing the Buck

Here’s a method that doesn’t catch IOExceptions, but passes them on.

```java
public int readInt(BufferedInputStream b) throws IOException {
    return b.read();
}
```

Now any method that calls this method will either have to catch any IOExceptions, or advertise that they may throw IOExceptions.
For Example: Passing the Buck

Here’s a method that doesn’t catch `IOException`, but passes them on.

```java
public int readInt(BufferedInputStream b) throws IOException {
    return b.read();
}
```

Now any method that calls this method will either have to catch any `IOExceptions`, or advertise that they may throw `IOExceptions`. 
Decisions, Decisions, . . .

Choosing which of these options (catching or advertising) to follow is a design decision.

One should try to identify the most appropriate place to handle exceptions.

There should be enough information to:

- formulate a meaningful error-message for the user; and
- make a reasonable decision as to whether, and how, to carry on.
Decisions, Decisions, . . .

Choosing which of these options (catching or advertising) to follow is a design decision.

One should try to identify the most appropriate place to handle exceptions.

There should be enough information to:

- formulate a meaningful error-message for the user; and
- make a reasonable decision as to whether, and how, to carry on.
Decisions, Decisions, . . .

As a rule of thumb:

- If an exception is to be caught, it should be caught as deep down in the method-call stack as possible (i.e., as close to the top-level, or `main()` method as possible).

We’ll look at an example from the Parser.java file from the practical.
As a rule of thumb:

- If an exception is to be caught, it should be caught as deep down in the method-call stack as possible (i.e., as close to the top-level, or \texttt{main()} method as possible).

We’ll look at an example from the \texttt{Parser.java} file from the practical.
The Parser

Parser.java contains one class, `Parser`, which contains just one public method:

```java
public Prop parse(String s) { ... }
```

This method reads through the given string and, if the string is a well-formed term of propositional logic, constructs an instance of `Prop` that represents that term.

But what if the string is not well-formed?
Decisions, Decisions, . . .

If the string that is passed to the `parse()` method is not well-formed, there are two — fairly general — options:

1. return `null`
2. throw an exception
Returning special values (such as null or -1) for special cases is a standard practice:

```java
public int find(int elt, int[] vals) {
    for (int i=0; i < vals.length; i++) {
        if (vals[i] == elt) return i;
    }
    // if we're here, elt not found
    return -1;
}
```

-1 can’t be an index of an array
Returning special values (such as null or -1) for special cases is a standard practice:

```java
public int find(int elt, int[] vals) {
    for (int i=0; i < vals.length; i++) {
        if (vals[i] == elt) return i;
    }
    // if we're here, elt not found
    return -1;
}
```

-1 can’t be an index of an array
Returning special values (such as null or -1) for special cases is a standard practice:

```java
public int find(int elt, int[] vals) {
    for (int i=0; i < vals.length; i++) {
        if (vals[i] == elt) return i;
    }
    // if we're here, elt not found
    return -1;
}
```

-1 can’t be an index of an array
This method searches an array for a given element; if the element is found in the array, it returns the index at which the element was found; otherwise, it returns -1.

Thus, a result of -1 indicates that the element doesn’t occur in the array.

The danger here is that this is an *implicit* convention, one that any user of the `find()` method should be aware of (so it should be clearly documented).
Option 2

For this option (the one we shall follow), we need to decide:

- what kind of exception to throw;
- where to throw such an exception;
- where (if at all) to catch such exceptions

As to what kind, we’ll create our own kind....
For this option (the one we shall follow), we need to decide:

- what kind of exception to throw;
- where to throw such an exception;
- where (if at all) to catch such exceptions

As to what kind, we’ll create our own kind....
For this option (the one we shall follow), we need to decide:

- what kind of exception to throw;
- where to throw such an exception;
- where (if at all) to catch such exceptions

As to what kind, we’ll create our own kind....
Class ParseException

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

    public String getMessage() {
        return "Parse error: " + super.getMessage();
    }
}
```

Inheriting Exception functionality
Overriding the method in class Exception
This is a way of calling the method being overridden
public class ParseException extends Exception {

    public ParseException(String s) {
        super(s);
    }

    public String getMessage() {
        return "Parse error: " + super.getMessage();
    }
}

Inheriting Exception functionality
Overriding the method in class Exception
This is a way of calling the method being overridden
public class ParseException extends Exception {

    public ParseException(String s) {
        super(s);
    }

    public String getMessage() {
        return "Parse error: " + super.getMessage();
    }

}

Inheriting Exception functionality
Overriding the method in class Exception
This is a way of calling the method being overridden
Inheriting Exception functionality
Overriding the method in class Exception
This is a way of calling the method being overridden
The superclass, Exception, provides most of what we need (the functionality of throwing, catching, etc.)

We simply override the Exception.getMessage() method by adding ‘Parse error:’ to the start of the error message.

(the getMessage() method returns the string that is given as a parameter to the constructor)

Because this class is a subclass of Exception, we effectively inherit all the Java interpreter’s mechanisms for throwing this kind of exception.
Where to Throw ParseExceptions?

Parser has only one public method, and 14 private methods that perform tasks such as

- reading variable names
- reading operator names
- reading white space, etc.

Most of these functions could (and do) throw an exception if the input is not of the expected form.

For example the following is from a method to read an operator name:
Reading Operator Names

```java
if (s.equals("and")) {
    return Operators.AND;
}
if (s.equals("or")) {
    return Operators.OR;
}
if (s.equals("implies")) {
    return Operators.IMPLIES;
}
// else
throw new ParseException(
    "Expected one of ‘and’, ‘or’, or ‘implies’");
```

If the input is not what is expected, a `ParseException` is thrown.
if (s.equals("and")) {
    return Operators.AND;
}
if (s.equals("or")) {
    return Operators.OR;
}
if (s.equals("implies")) {
    return Operators.IMPLIES;
}
// else
throw new ParseException("Expected one of ‘and’, ‘or’, or ‘implies’");

If the input is not what is expected, a ParseException is thrown.
Where to Catch ParseExceptions?

The only public method in Parser is parse(); all of the private methods that throw exceptions must, ultimately have been called by this method.

The user of this method might be entering terms to be parsed from the command line, or from a GUI. We don’t know where to print out the error message, so it doesn’t make sense to catch exceptions here.

Instead, we pass the buck:

```java
public Prop parse(String s) throws ParseException { ... }
```
Where to Catch ParseExceptions?

The only public method in `Parser` is `parse()`, all of the private methods that throw exceptions must, ultimately have been called by this method.

The user of this method might be entering terms to be parsed from the command line, or from a GUI. We don’t know where to print out the error message, so it doesn’t make sense to catch exceptions here.

Instead, we pass the buck:

```java
public Prop parse(String s) throws ParseException {
    ...}
```
Where to Catch ParseExceptions?

The only public method in Parser is parse(); all of the private methods that throw exceptions must, ultimately have been called by this method.

The user of this method might be entering terms to be parsed from the command line, or from a GUI. We don’t know where to print out the error message, so it doesn’t make sense to catch exceptions here.

Instead, we pass the buck:

```java
public Prop parse(String s) throws ParseException { ... }
```
Catching at the Top Level

Now `ParseException` can be caught at (or closer to) the top level of whatever application uses the parser.

For example, in a main method, having set up a `BufferedReader br:`
in a main method

Parser p = new Parser();
System.out.println(
    "Enter a term, ‘q’ to quit");
String str = "";
while (! str.equals("q")) {
    str = br.readLine();
    try {
        System.out.println(p.parse(str).toString());
    } catch (ParseException pe) {
        System.out.println(pe.getMessage());
    }
}
Encapsulating Propositions?

The main problem with our implementation of propositions is that we can construct instances of $\text{Prop}$ that do not correspond to any well-formed term (see Lecture 17).

This problem concerns the adequacy of the representation.
### Adequacy of Representation

Any implementation of an abstract data type *represents* the data elements in some way.

For example, stacks may be represented by arrays and pointers; Boolean terms by tree structures.

In Java, a data element will be represented by an instance of a class (e.g., `Stack`, `Prop`).

The representation is said to be **adequate** if:

- every data element can be represented by some instance
- every instance represents a data element.
Adequacy of Representation

Any implementation of an abstract data type represents the data elements in some way.

For example, stacks may be represented by arrays and pointers; Boolean terms by tree structures.

In Java, a data element will be represented by an instance of a class (e.g., Stack, Prop).

The representation is said to be adequate if:

- every data element can be represented by some instance
- every instance represents a data element.
An Inadequate Constructor

The constructor for the class `Prop` is:

```java
public Prop(Operator o, Prop[] ps) {
    op = o;
    operands = ps;
}
```

This is clearly ‘too generous’: it allows the creation of instances that do not correspond to well-formed terms.

We could test that the given array of `Props` contains the correct number of operands for the given operator (we omit the details... )
public Prop(Operator o, Prop[] ps) throws BadTermException {
    if (ps is the right size for o) {
        op = o;
        operands = ps;
    } else {
        throw new BadTermException();
    }
}
public class BadTermException
    extends Exception
{
    public String getMessage() {
        return "incorrect number of arguments";
    }
}
If we run the following program:

```java
public static void main(String[] args)
    throws BadTermException
{
    Prop a, t;
    a = new Prop(Operators.makeVar("a"),
                 new Prop[0]);
    t = new Prop(Operators.AND_OP,
                 new Prop[1]{a});
    System.out.println(t.toString());
}
```
We get the following on standard error:

```
Exception in thread "main" BadTermException: 
  incorrect number of arguments 
  at Prop.<init>(Prop.java:31)
  at Prop.main(Prop.java:92)
```

Note: this is the Prop constructor
We get the following on standard error:

```
Exception in thread "main" BadTermException: incorrect number of arguments
  at Prop.<init>(Prop.java:31)
  at Prop.main(Prop.java:92)
```

Note: this is the Prop constructor
If we run *this* program:

```java
public static void main(String[] args)
    throws BadTermException {
    try {
        Prop a, t;
        a = new Prop(Operators.makeVar("a"),
                      new Prop[0]);
        t = new Prop(Operators.AND_OP,
                      new Prop[1]{a});
        System.out.println(t.toString());
    } catch (BadTermException e) {
        System.out.println(e.getMessage());
    }
}
```
Then we get the following on standard output:

```
terminal output

incorrect number of arguments
```

...which is much more user-friendly.
However, That’s not the best Solution

This solves the problem of adequacy of representation: now, the only instances of Prop that can be created are those corresponding to well-formed terms.

However, although it allowed us to see that main methods and constructors can both advertise exceptions, there is a simpler solution, which lets us see that we can make constructors private.
Another solution is:

- make the `Prop` constructor private (`BadTermException` will no longer be needed);
- add package-visible methods to construct terms:
  ```java
  Prop makeAndTerm(Prop p1, Prop p2)
  ```
  for each operator and constant;
private Prop(Operator o, Prop[] ps) {
    op = o;
    operands = ps;
}

Prop makeAndTerm(Prop p1, Prop p2) {
    return new Prop(Operators.AND_OP,
                    new Prop[]{p1, p2});
}

Prop makeNotTerm(Prop p) {
    ...
}

...
One common situation where resource errors are common is in data structures, such as Stacks, Lists, Cardhands, etc., where we have operations to access elements of the data structure (or to add to data structures that have fixed capacity)

For example, consider taking the top (or popping) an empty stack, taking the top card in an empty hand, etc.

In these situations, Exceptions are the best solution
One common situation where resource errors are common is in data structures, such as Stacks, Lists, Cardhands, etc., where we have operations to access elements of the data structure (or to add to data structures that have fixed capacity)

For example, consider taking the top (or popping) an empty stack, taking the top card in an empty hand, etc.

In these situations, Exceptions are the best solution
One common situation where resource errors are common is in data structures, such as Stacks, Lists, Cardhands, etc., where we have operations to access elements of the data structure (or to add to data structures that have fixed capacity)

For example, consider taking the top (or popping) an empty stack, taking the top card in an empty hand, etc.

In these situations, Exceptions are the best solution
public class Stack {

    private int[] values;
    private int pointer;

    public int top() throws EmptyStackException {
        if (pointer < 0) {
            throw new EmptyStackException();
        }
        return values[pointer];
    }

    ...
}
Summary

- Checked Exceptions
- Advertising Exceptions
- private constructors

Next:

Concurrency