Lecture 17

Exceptions
Errors

Writing programs is not trivial.
Most (large) programs that are written contain errors:
in some way, the program doesn’t do what it’s meant to do.

There are three main kinds of error:

- syntactic errors
- semantic errors
- input/resource errors
Programming languages are **formal languages**.
This means that there is a formal definition of what is an acceptable *text* for a program in a given language.
For example, Java requires all commands to end with a semicolon (`;`).
The compiler will flag any syntactic errors, and will not produce any executable code unless the program is syntactically correct.
Syntactic errors are not serious, as they can be automatically detected.
Programs do exactly what the source code says they do: unfortunately, this might not be what they’re meant to do. This is a semantic error.

For example,

- a program might not provide some required functionality (due to poor design or requirements elicitation).
- a program might provide incorrect functionality (e.g., errors in rounding up fractions).

The issue here is the correctness of the program.
Input/Resource Errors

A program might function adequately, but be dependent on factors outside the control of its designers and programmers. For example, it might require user input in a specific format, or it might need to access servers across a network, or it might require a minimum amount of RAM memory on the machine it is running on.

If some requirement is not met, the program may fail (not in a disastrous way, one would hope). The issue here is the robustness of the program.
Errors, Failures and Exceptions

A semantic coding error (or ‘bug’) might go unnoticed until the error causes some input/resource error (such as accessing an array outside its bounds). In such cases, the Java interpreter will raise an exception.

To review exceptions, we’ll go through an example of a program that contains a bug, a fatal error that causes the Java interpreter to halt and report the error.

Here’s the program:
A Buggy Program

in class Prop

```java
public static void main(String[] args) {
    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());
}
```
Let’s go through the execution of this `main()`-method.

As we’ve seen, some methods call other methods, which may in turn call other methods, and so on.

The Java interpreter keeps track of where it is by maintaining a method-call stack, which stores all the nested method calls. The first method on this stack is always `main()`.

The first line declares two variables of type `Prop`:

```
Prop a, t;
```
Let’s go through the execution of this `main()`-method. As we’ve seen, some methods call other methods, which may in turn call other methods, and so on.

The Java interpreter keeps track of where it is by maintaining a method-call stack, which stores all the nested method calls.

The first method on this stack is always `main()`

The first line declares two variables of type `Prop`:

```
Prop a, t;
```
Let’s go through the execution of this `main()`-method.
As we’ve seen, some methods call other methods, which may in turn call other methods, and so on.
The Java interpreter keeps track of where it is by maintaining a *method-call stack*, which stores all the nested method calls.
The first method on this stack is always `main()`

The first line declares two variables of type `Prop`:
```java
Prop a, t;
```
Let’s go through the execution of this `main()`-method. As we’ve seen, some methods call other methods, which may in turn call other methods, and so on. The Java interpreter keeps track of where it is by maintaining a method-call stack, which stores all the nested method calls. The first method on this stack is always `main()`

The first line declares two variables of type `Prop`:

```java
Prop a, t;
```
The next command creates an instance of `Prop`, corresponding to a propositional variable 'a, and assigns this instance to the Java variable a.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of Prop, corresponding to a propositional variable ‘a, and assigns this instance to the Java variable a.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of `Prop`, corresponding to a propositional variable `a`, and assigns this instance to the Java variable `a`.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of Prop, corresponding to a propositional variable 'a, and assigns this instance to the Java variable a.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of Prop, corresponding to a propositional variable 'a, and assigns this instance to the Java variable a.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of `Prop`, corresponding to a propositional variable `a`, and assigns this instance to the Java variable `a`.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.
The next command creates an instance of `Prop`, corresponding to a propositional variable 'a, and assigns this instance to the Java variable `a`.

Note that parameters to a method are always evaluated before that method is actually called.

```java
a = new Prop(
    Operators.makeVar("a"),
    new Prop[0]);
```

Once a method is fully evaluated, it is removed from the call stack.

Array constructor

`Prop` constructor

**Line 2**
The next command also creates an instance of `Prop`, and assigns that instance to the variable `t`:

```java
t = new Prop(Operators.AND_OP, new Prop[1] {a});
```

This *does not* correspond to a well-formed term of propositional logic.
The next command also creates an instance of Prop, and assigns that instance to the variable t:

```java
t = new Prop(Operators.AND_OP,
             new Prop[1]{a});
```

This does not correspond to a well-formed term of propositional logic.
The next command also creates an instance of Prop, and assigns that instance to the variable t:

```java
t = new Prop(Operators.AND_OP, 
               new Prop[1]{a});
```

This does not correspond to a well-formed term of propositional logic.
The Bug

We’re representing terms as tree structures. As a tree, this value has a top operator, ‘and’, and has just one subtree, the variable \(a\).

Of course, the operator ‘and’ requires \(two\) arguments. This might worry us (with good cause), but it doesn’t raise any compiler-errors, and the interpreter executes the code just as it should do.

Specifically, the Prop constructor is evaluated:
An Ill-Formed Term

Prop constructor

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

So the instance stored in `t` has

- field `t.op` storing `AND_OP`, and
- field `t.operands` storing an array with one element, the `Prop` corresponding to the variable `a`. 
The remaining line in the `main()`-method should print a string to standard output:

```
System.out.println(t.toString());
```

Before the call to `println()` can do anything, its argument `toString()` must be evaluated:

```
t.toString()
```
The remaining line in the `main()`-method should print a string to standard output:

```
System.out.println(t.toString());
```

Before the call to `println()` can do anything, its `argument` must be evaluated:

```
t.toString()
```

method-call stack

```
t.toString()
  toString()
    main
```
public String toString()
{
    return toString(Operators.MAXPREC);
}

The argument is evaluated (to 48), and then toString(int) is called. The result of this call is the value that will be returned (to println()).
public String toString()
{
    return toString(Operators.MAX_PREC);
}

The argument is evaluated (to 48), and then \texttt{toString(int)} is called. The result of this call is the value that will be returned (to \texttt{println()}).
public String toString()
{
    return toString(Operators.MAX_PREC);
}

The argument is evaluated (to 48), and then `toString(int)` is called.

The result of this call is the value that will be returned (to `println()`).
public String toString()
{
    return toString(Operators.MAX_PREC);
}

The argument is evaluated (to 48), and then toString(int) is called. The result of this call is the value that will be returned (to println()).
Recall:
- prec is 48;
- t.op is AND_OP;
- t.operands is an array with one element.

```java
public String toString(int prec) {
    return op.toString(operands, prec);
}
```
Recall:
- prec is 48;
- t.op is AND_OP;
- t.operands is an array with one element.

```java
public String toString(int prec)
{
    return op.toString(operands,
                      prec);
}
```
The value returned from the call

```java
op.toString(operands, prec);
```

is the value which will be returned to `toString()`, which will return it to `println()`.

Since `t.op` is `AND_OP`, the method that is called here is the method in the anonymous class in the declaration of `AND_OP`:
No problem with arg[0].toString()
Recall that args is t.operands, which has only one element....
public String toString(Prop[] args, int prec) {
    String s = args[0].toString(getPrecedence())
        + " and "
        + args[1].toString(getPrecedence());
    if (prec < AND_OP_PREC)
        s = "(" + s + ")";
    return s;
}
public String toString(Prop[] args, int prec)
{
    String s = args[0].toString(getPrecedence())
            + " and "
            + args[1].toString(getPrecedence());
    if (prec < AND_OP_PREC)
        s = "(" + s + ")";
    return s;
}

No problem with args[0].toString()

Recall that args is t.operands, which has only one element. . . .
The Resource Error

When the interpreter attempts to evaluate

```java
args[1].toString(AND_OPPREC)
```

it attempts to get a reference to the second `Prop` in the array `args`, i.e., `t.operands[1]`.

But this array only has one element, so `args[1]` is `out of bounds`.

In such cases, Java throws an Exception:
The Resource Error

When the interpreter attempts to evaluate

```java
args[1].toString(AND_OP_PREC)
```

it attempts to get a reference to the second Prop in the array `args`, i.e., `t.operands[1]`.

But this array only has one element, so `args[1]` is out of bounds.

In such cases, Java throws an Exception:
ArrayIndexOutOfBoundsException

The following appears on standard error output (usually standard output):

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException
    at Operators$3.toString(Operators.java:164)
    at Prop.toString(Prop.java:66)
    at Prop.toString(Prop.java:55)
    at Prop.main(Prop.java:90)
```

This information is called the ‘stack trace’, and gives the state of the method-call stack at the point when the Exception was thrown (and gives useful line numbers in the source files).
ArrayIndexOutOfBoundsException

The following appears on standard error output (usually standard output):

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException
    at Operators$3.toString(Operators.java:164)
    at Prop.toString(Prop.java:66)
    at Prop.toString(Prop.java:55)
    at Prop.main(Prop.java:90)
```

This information is called the ‘stack trace’, and gives the state of the method-call stack at the point when the Exception was thrown

(and gives useful line numbers in the source files).
Error-Recovery

Note that the exception that is thrown can be caught at any point in the stack trace: this is called error-recovery.

In Java, methods are caught in try-catch clauses:

```java
try {
    // code that can raise an exception
} catch (Exception e) {
    // recovery code
}
```

A program is robust if it recovers from unexpected errors (‘fails gracefully’).

NB - ‘unexpected errors’ means resource errors beyond the programmer’s control — and this doesn’t apply in this example (it’s just a programmer’s bug)
Error-Recovery

Note that the exception that is thrown can be caught at any point in the stack trace: this is called error-recovery.

In Java, methods are caught in try-catch clauses:

```java
try {
    // code that can raise an exception
} catch (Exception e) {
    // recovery code
}
```

A program is robust if it recovers from unexpected errors (‘fails gracefully’).

NB - ‘unexpected errors’ means resource errors beyond the programmer’s control — and this doesn’t apply in this example (it’s just a programmer’s bug)
Error-Recovery

Note that the exception that is thrown can be caught at any point in the stack trace: this is called error-recovery.

In Java, methods are caught in try-catch clauses:

```java
try {
    // code that can raise an exception
} catch (Exception e) {
    // recovery code
}
```

A program is robust if it recovers from unexpected errors (‘fails gracefully’).

NB - ‘unexpected errors’ means resource errors beyond the programmer’s control — and this doesn’t apply in this example (it’s just a programmer’s bug)
Catching Exceptions

For example, if the `toString()` method was:

```java
public String toString()
{
    String s;
    try {
        s = toString(MAX_PREC);
    }
    catch (Exception e)
    {
        s = "the term is not well-formed";
    }
    return s;
}
```

Then the output of the `main()`-method would be:

```
the term is not well-formed
```
Catching Exceptions

For example, if the `toString()` method was:

```java
public String toString()
{
    String s;
    try { s = toString(MAX_PREC); } catch (Exception e) { s = "the term is not well-formed"; } return s;
}
```

Then the output of the `main()`-method would be:

```
the term is not well-formed
```
Generally, a robust (= good) program will, when things go wrong, catch any exceptions and:

- inform the user (if necessary), and
- carry on (if possible).

Note that this is what the java interpreter does: when an exception is thrown (and not caught), it informs the user by printing to stderr the type of exception (e.g., ArrayIndexOutOfBoundsException), and the stack trace.

But note that ‘the term is not well-formed’ is a much better error message for most users!
A Rule of Thumb

Generally, a robust (= good) program will, when things go wrong, catch any exceptions and:

- inform the user (if necessary), and
- carry on (if possible).

Note that this is what the java interpreter does: when an exception is thrown (and not caught), it informs the user by printing to stderr the type of exception (e.g., ArrayIndexOutOfBoundsException), and the stack trace.

But note that ‘the term is not well-formed’ is a much better error message for most users!
A Rule of Thumb

Generally, a robust (= good) program will, when things go wrong, catch any exceptions and:

- inform the user (if necessary), and
- carry on (if possible).

Note that this is what the Java interpreter does:
when an exception is thrown (and not caught), it informs the user by printing to stderr the type of exception (e.g., ArrayIndexOutOfBoundsException), and the stack trace.

But note that ‘the term is not well-formed’ is a much better error message for most users!
The Exception Zoo

java.lang.Exception
  |
  +-java.lang.RuntimeException
    |
    +-java.lang.IndexOutOfBoundsException
    |
    | +-java.lang.ArrayIndexOutOfBoundsException
    |
    +-java.lang.NullPointerException
    |
    ...

The try-catch construct can be used to catch specific subclasses of Exception. For example, the method

```java
public static int parseInt(String s)
```

in class java.lang.Integer attempts to read a string as an integer.

```java
int i = Integer.parseInt("27");
```

will set `i` to 27.

If the String parameter cannot be parsed as an integer, then the method throws a NumberFormatException.
Catching NumberFormatException

String s = "3w";
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println("not a valid integer: ",
                       + nfe.getMessage);
}

Catching the specific subclass of RuntimeException
The Exception class has some useful methods
Catching NumberFormatException

String s = "3w";
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println("not a valid integer: "+ nfe.getMessage());
}

Catching the specific subclass of RuntimeException
The Exception class has some useful methods
Catching NumberFormatException

```java
String s = "3w";
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println("not a valid integer: "+ nfe.getMessage());
}
```

Catching the specific subclass of RuntimeException

The Exception class has some useful methods
Different types of Exception can be caught separately:

```java
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println(...);
} catch (NullPointerException npe) {
    System.err.println("s not instantiated");
    nfe.printStackTrace();
}
```

If `s` is, e.g., "doh", the first catch-block will be executed; if `s` is null, the second catch-block will be executed.
Different types of `Exception` can be caught separately:

```java
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println(...);
} catch (NullPointerException npe) {
    System.err.println("s not instantiated");
    nfe.printStackTrace();
}
```

If `s` is, e.g., "doh", the first catch-block will be executed; if `s` is null, the second catch-block will be executed.
Different types of Exception can be caught separately:

```java
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println(...);
} catch (NullPointerException npe) {
    System.err.println("s not instantiated");
    nfe.printStackTrace();
}
```

If `s` is, e.g., "doh", the first catch-block will be executed; if `s` is `null`, the second catch-block will be executed.
When an exception is thrown, the Java interpreter looks through the list of catch-blocks, and executes the first one that applies to the particular exception.

Superclasses should therefore be below subclasses:

```java
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
    System.err.println(...);
} catch (NullPointerException npe) {
    System.err.println("s not instantiated");
} catch (RunTimeException re) {
    System.err.println("something else...");
}
```
Default Actions

After some number (including zero!) of catch-blocks, you can include a `finally-block`, which will be executed whether or not an exception is thrown.

```java
// some code to open a file
try {
    // to write to the file
} catch (IOException ioe) {
    // oops
} finally {
    // close the file
}
```
The `RuntimeException` class is for exceptions that can occur ‘in the normal running of the Java Virtual Machine.’

(That quote is from the API!)

They generally arise from semantic coding errors (programmers’ bugs).

Exceptions can be ‘thrown’ and ’caught’.

As we’ll see, uncaught errors generally need to be advertised through `throws`-clauses in method heads.

This has the advantage that users of methods that might throw exceptions know this, and therefore know that they should either pass those exceptions on (and advertise this), or catch those exceptions.
The `RuntimeException` class is for exceptions that can occur ‘in the normal running of the Java Virtual Machine.’
(That quote is from the API!)
They generally arise from semantic coding errors (programmers’ bugs).

Exceptions can be ‘thrown’ and ’caught’.

As we’ll see, uncaught errors generally need to be advertised through `throws`-clauses in method heads.

This has the advantage that users of methods that might throw exceptions know this, and therefore know that they should either pass those exceptions on (and advertise this), or catch those exceptions
RuntimeException

However, subclasses of RuntimeException do not need to be advertised.

Advertising them would be of no benefit, as they almost always arise through oversight or carelessness.

For this reason, RuntimeException and its subclasses are called unchecked exceptions.
And Finally...

Bugs are a fact of life, but can be contained:

```java
public static void main(String[] args) {
    try {
        // code here
    }
    catch (Exception e) {
        System.out.println("An error occurred. " +
                           "Please submit a report ...");
    }
}
```
Types of errors
Exceptions
method-call stack
stack trace
try-catch

Next:
Checked Exceptions