The current topic: Syntax and semantics

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
✓ Logic programming: Prolog
  ✓ Introduction
  ✓ Rules, unification, resolution, backtracking, lists.
  ✓ More lists, math, structures.
  ✓ More structures, trees, cut.
  ✓ Negation.
- Syntax and semantics
- Exceptions

Announcements

- Reminder: The deadline for submitting a re-mark request for Term Test 2 is ***the end of class* today.**
- Reminder: Lab 3 is due on Monday at 10:30 am.
- Aids allowed for the final exam:
  - One double-sided aid sheet, produced however you like, on standard letter-sized (8.5" x 11") paper.
- Exam period office hours:
  - Monday Dec. 8th, 12:30-1:30, SF3207
  - Wednesday Dec. 10th, 12:30-1:30, SF3207
  - Friday Dec. 12th, 12:30-1:30, SF3207
  - Monday Dec. 15th, 12:30-1:30, SF3207
  - Tuesday Dec. 16th, 11:00-12:00, SF3207

Summary: what the parser does

- The parser must understand the syntax of the program:
  - produce a representation of the syntax
    - either explicitly, as a data structure
    - or implicitly, in the sequence of function calls
  - do it efficiently
  - do it without having to look ahead at too much of the source code
    - preferably, just one symbol ahead

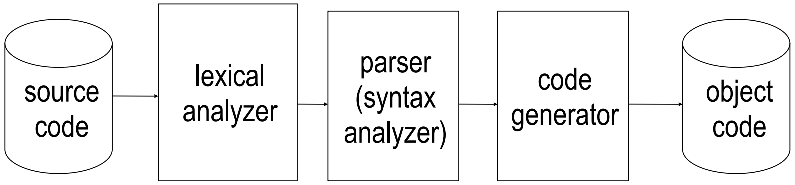
Translation

- A translator can be either
  - A *compiler*
    - translates an entire program from source code to object code
    - output is what the programmer wants to hand to the user
  - An *interpreter*
    - translates one statement at a time
    - executes the statement
    - output is what the user wants to see

Where do we get the tokens for the parser?

- To decide what to do next, the parser needs to know what token comes from the source code next.
- A "token" is not just a single character, but a chunk of the input such as a number, a name, an operator.
  - may be more than just a sequence of characters
  - could have name, type, value
- Conceptually, breaking the input into tokens is a separate step, *lexical analysis*.
  - The parser is complicated enough, without dealing with breaking the input into tokens.
- The lexical analyzer breaks the source code into tokens, discarding comments and labeling the tokens according to what kind of thing they are (name, number, operator, etc.).
- In a C compiler, the preprocessor would do its work before the lexical analyzer.

The structure of a compiler



- Omitted: optimization, various stages of code generation

The current topic: Exceptions

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
✓ Logic programming: Prolog
✓ Syntax and semantics
- Exceptions

Exceptions

- Origins
- Exceptions in Java
- Classifying exceptions
- Program design with exceptions

- Reference:
  - Sebesta, chapter 14

Why exceptions?

- Exceptions report exceptional conditions: unusual, strange, disturbing.
- These conditions deserve exceptional treatment: not the usual go-to-the-next-step, continue-onwards approach.
- Therefore, understanding exceptions requires thinking about a different model of program execution.

Since PL/I

- PL/I was designed about 1964.
- It includes "ON" conditions:

```
ON SUBSCRIPTRANGE
  BEGIN;
  ...
  END;
```

  - This specifies how to handle a particular error (the error `SUBSCRIPTRANGE`).

- There are built-in and user-defined exceptions.
- Handling is controlled dynamically:
  - ON blocks (exception handlers) are executable and appear as part of the program.

Problems with PL/I's approach

- Hard to figure out how a particular exceptional condition is handled in a particular block of code.
  - May depend on execution path of the program.
  - So need to trace code to determine which exception handler is active at a particular point.

More recent approaches

- We've already seen exception handling in Python.
- Java uses a stricter and more structured approach.
  - Syntax is similar to C++.

Exceptions in Java

- Two statements:

  `throw` <Throwable>`;`

  `try {` <statements> `}`
  `catch (`<Throwable parameter>`) {` <statements> `}`

  - A `catch` clause must come after a `try` clause; it "belongs" to the `try` clause.
  - Notice that in Java, exceptions must be descendants of class `Throwable`.
  - In C++, anything can be `thrown`.

- Analogy:
  - `throw` = "I'm in trouble, so I throw a rock through a window, with a message tied to it."
  - `try` = "Someone in the following block of code might throw rocks of various kinds. All you catchers line up ready for them."
  - `catch` = "If a rock of my kind comes by, I'll catch it and deal with it."

Example

```
int i = 0;
int sum = 0;
try {
  while (true) {
    sum += i++;
    if (i >= 10)
      throw new Exception("i at limit");
  }
}
catch (Exception e) {
  System.out.println("sum to 10 = " + sum);
}
```

- Output:

```
sum to 10 = 45
```

Why is that example not a good one?

- The situation the exception reports is not exceptional.
  - The point at which the exception occurs is completely predictable.
  - But this doesn't mean that exceptions are only used for "bad" situations.
    - For example, a program might keep reading from a file until an end-of-file exception occurs. In this case, the exception is expected, but not predictable (since different files have different lengths).
- There is likely to be inefficiency.
  - Extra work in creating and handling an exception.
- It's uncharacteristic.
  - Real uses of exceptions aren't usually local. Either the exception isn't raised locally, or it isn't caught locally.
  - That is, throw and catch aren't generally in the same block of code.
    - Why?

Java exception syntax

- 'throw' <throwable>
  - where <throwable> is a reference to an instance of `Throwable` or a subclass
- 'try' '{' <statements> '}' <catchclause> { <catchclause> } [<finallyclause>]
- <catchclause> ::= 'catch' '(' <parameter> ')' '{' <statements> '}'
- <parameter> ::= <throwable class> <name>
- <throwable class> ::= 'Throwable' | <descendant of Throwable>
- <finallyclause> ::= 'finally' '{' <statements> '}'
  - C++ doesn't have allow a `finally` clause, but recall that Python does.
- The <statements> in a <catchclause> can be anything you like.
  - including other `throw` statements
  - referring to the <parameter> as needed

Example

```
try { ... }
catch (OutOfMemoryError e) {
  System.out.println("Good grief! I give up.");
  throw e;
}
catch (ImportantButLocalProblem e) {
  System.err.println("Oh, dear: " + e.getMessage());
}
catch (UnrecoverableProcessingError foo) {
  System.err.println("We're stuck: " + foo.getMessage());
  throw new MyDisasterReportingException();
}
```

Another example

- Suppose `ExSup` is the parent of `ExSubA` and `ExSubB`.

```
try { ... }
catch (ExSubA e) {
  // We do this if an ExSubA is thrown.
}
catch (ExSup e) {
  // We do this if any ExSup that's not an ExSubA is thrown.
}
catch (ExSubB e) {
  // We never do this, even if an ExSubB is thrown.
}
finally {
  // We always do this, even if no exception is thrown.
}
```

The methods of Throwable

- Constructors:
  `Throwable( )`, `Throwable(String message)`

- Other useful methods:
  `getMessage( )`
  `printStackTrace( )`

An example

```
void m1 () { ...
  if (...) throw ExA("expl"); ...
}
void m2 () { ...
  m1(); ...
}
void m3 () {
  try { ...
    m2(); ...
  }
  catch (ExA e) {
    // corrective action and/or error message.
    // Error message example:
    e.printStackTrace();
    System.err.println(e.getMessage());
  }
}
```

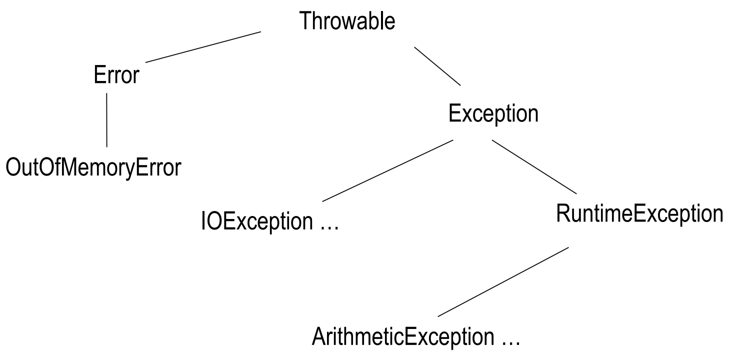
What do we gain with exceptions?

- Less programmer time spent on handling errors.

- Cleaner program structure:
  - No need to communicate errors by returning (and repeatedly checking for!) error codes.
  - This means that your algorithms aren't cluttered with error checks and error-handling code.
  - So there's a much clearer distinction between algorithm code and error-handling code.

The programmer's view

- Some of Java's built-in exceptions:



Throwable

Error

OutOfMemoryError

Exception

IOException …

RuntimeException

ArithmeticException …

Which exceptions should you catch?

- Don't say this:

```
catch (Throwable thr) { ... }
```

  - It's not specific enough.
  - Only catch exceptions that you're actually prepared (and able) to handle.
  - The equivalent code in C++ (to catch all exceptions) is:

    ```
    catch(...) { ... }
    ```

    where the first "..." is actual code, not just a placeholder.

- Don't even say this, for the same reason:

```
catch (Exception e) { ... }
```

Things not to catch, continued

- Think hard before saying this:

```
catch (Error e) { ... }
```

  - The Error class consists of things that you probably can't handle effectively.
    - e.g. OutOfMemoryError, StackOverflowError.

- You can certainly write programs that handle Error or Throwable when you're experimenting with exceptions.
  - Just don't do it in real code.

Which classes should you throw?

- Suppose you're writing your own exception class. What should the parent class be?

- It is legal:
  - to throw an instance of `Throwable` or any of its descendants.
  - to extend `Throwable` or any of its descendants.
- So your parent class can be `Throwable` or any of its descendants.

- But `Throwable` itself, and `Error` and its descendants are probably not suitable for throwing in an ordinary program.
  - `Throwable` isn't specific enough.
  - `Error` and its descendants describe serious, unrecoverable conditions.
    - e.g. `OutOfMemoryError` (actually a child of `VirtualMachineError`)

Your exceptions should extend Exception

- Example: a method `m( )` that throws your own exception `ExSub`, a subclass of `Exception`:

```
class MyClass {
  public static class ExSub extends Exception {...}
  public void m() { ...
    if (...) throw new ExSub("oops!"); ...
  }
}
```

- But the compiler complains!
  - Why?

You have to announce what you might throw

- Exceptions are, in general, "checked" by the compiler. It wants you to tell it that your method might throw an `ExSub`:

```
class MyClass {
  public static class ExSub extends Exception {...}
  public void m() throws ExSub { ...
    if (...) throw new ExSub ("oops!"); ...
  }
}
```

- You are not *guaranteeing* that `m()` will throw an `ExSub`; you're only *reserving the right* to throw an `ExSub`.
  - Why should this be explicitly stated?

- In C++, the "`throws`" clause is optional. Functions that don't include this clause are allowed to throw anything they like.

The caller's responsibility

```
public void m(int mlimit) throws ExSub { ...
  if (...) throw new ExSub ("oops!"); ...
}

public void mcaller(int x) throws ExSub {
  try {
    int y = m(x); ...
  }
  catch (ExSub e) {
     ...
  }
}
```

- `mcaller` must either
  - announce "`throws ExSub`" – in this case, the call to `m()` does not need to be enclosed in a `try...catch` block.

    or
  - `catch ExSub.`

No "throws" for Errors and RuntimeExceptions

- `Errors` and `RuntimeExceptions` (e.g. `NullPointerException`, `IndexOutOfBoundsException`) are not checked:

```
class MyClass {
  public static class ExSub extends RuntimeException {...}
  public void m() /* no "throws" */ { ...
    if (...) throw new ExSub("oops!"); ...
  }
}
```

- The compiler accepts this. A method is allowed to throw an `Error` or a `RuntimeException` even if its announced interface doesn't mention it.
  - Why?

- Question: Why would you choose to extend `Exception` instead of `RuntimeException`? Or is the compiler's exception checking such a benefit that you should *always* choose `Exception`?

Deciding if it's "Runtime"

The Java API documentation says:

- (under `Exception`): "The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch."

- (under `RuntimeException`): "`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine."

Deciding if it's "Runtime"

The Java Language Specification (section 11, 3rd edition) says:

- "The runtime exception classes (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs. Many of the operations and constructs of the Java programming language can result in runtime exceptions. The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irritation to programmers."

- "For example, certain code might implement a circular data structure that, by construction, can never involve null references; the programmer can then be certain that a `NullPointerException` cannot occur, but it would be difficult for a compiler to prove it. The theorem-proving technology that is needed to establish such global properties of data structures is beyond the scope of this specification."

Deciding if it's "Runtime"

- non-`RuntimeException` examples:
  - `IOException`, `SQLException`

- `RuntimeException` examples:
  - `ArithmeticException`, `IndexOutOfBoundsException`, `NullPointerException`

More documentation: `IOException` says:

- "Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations."
  - Is it clear that this shouldn't be a `RuntimeException?`
  - The distinction between `Runtime` and non-`Runtime` exceptions is not clear-cut.

Deciding if it's "Runtime"

- You need to decide whether the benefits of checked exceptions outweigh the additional work that will be required as a result of using them.
  - Make this decision carefully. Don't automatically decide to use runtime exceptions just to avoid additional work.

- Some things to consider:
  - When a method is declared as throwing a checked exception, callers have to make an explicit decision about how they want to deal with such an exception: either throw it or catch it; doing neither is not an option.
  - Is there anything that can be done when the exception occurs, or is terminating the program the only option? If something can be done, this is a good reason to use a checked exception.
  - Is the exception similar to `NullPointerException` in the sense that it only occurs as a result of poorly-written code? This is a good reason to use a runtime exception.

Once caught, what next?

- Here's what the user sees when you divide 1 by 0:

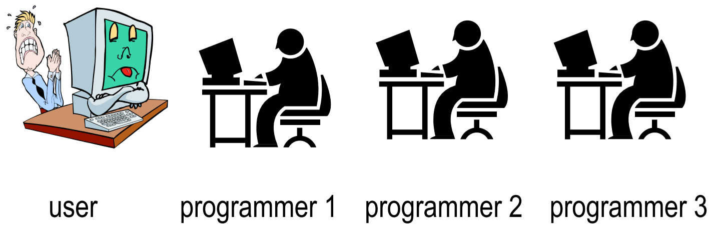
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Thrower.main(Thrower.java:7)
```

- How is this reasonable?
  - How is the user supposed to use this information?

- Those error messages are for programmers, not users.
  - Even programmer-users aren't helped.

What's the context?

- It's a programmer who decides to throw exceptions, and a programmer who catches them:



Who does what? Who knows what?

- The user misuses the program.
  - needs to see a helpful error message

- Programmer 1 writes the program that the user misused.
  - needs to provide a helpful error message on the basis of a caught exception
  - knows the purpose of the application and understands the user interface

- Programmer 3 writes the code that detects trouble.
  - knows the kind of error that occurred and throws the appropriate exception
  - but cannot know the context: how serious the error is, what was going on when it happened, how it should be presented to the user

Who does what? Who knows what?

- Programmer 2 catches the exception thrown by Programmer 3.
  - knows enough context to prepare a description of the error
    - The type of the exception itself, and the message it carries, tell Programmer 2 what to say in the context of what Programmer 2's code was doing at the time when the trouble arose.
  - but Programmer 2 does not know how to present the description to the user
- Can Programmer 1 tell Programmer 2 and Programmer 3 how to handle the problem? No:
  - They may no longer work in the same jobs.
  - Their code may be used in more than one application anyway.

They can all work together

Programmer 1's role:

- Make an error-handling object with these tasks:
  - decide which exceptions are fatal, which need to be reported as errors or warnings, etc.
  - carry out the actual reporting task when it's appropriate.
- Programmer 1's code passes the error-handler (a "filter") to Programmer 2's code, which passes it to Programmer 3's code.

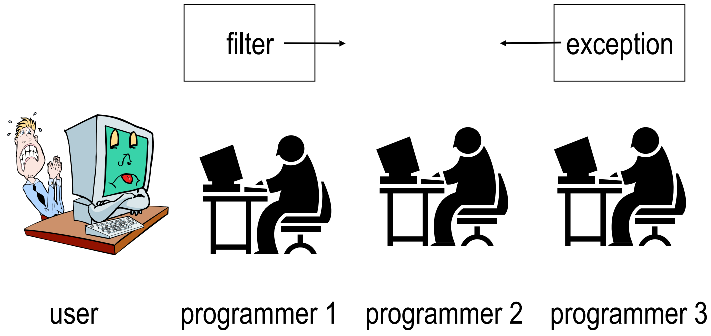
Programmer 3's role:

- When an error condition is detected, create an exception.
- If Programmer 3 is sure the code can't continue, then they simply throw the exception.
- If Programmer 3 isn't sure, they pass the exception to a method of the filter, which may: display a message; do nothing and return; re-throw this exception; or throw another exception that describes the original exception with some kind of classification data.

Programmer 2's role

- Programmer 2's catch clause catches the exception thrown by Programmer 3. The exception is then handled with the help of Programmer 1's filter:
  - The filter can be used to prepare an appropriate error message, using the exception itself and information provided by Programmer 2.
    - e.g., the exception is a divide-by-zero error, and Programmer 2 was assigning task priorities, so perhaps "inappropriate zero priority during task assignment"?
  - The filter can also be used to display the error message.
    - in a text-based error window? flashed on a plane's instrument panel?
- If the filter decides the exception is important enough, it may finish by throwing another exception that tells the top-level code to end program execution.
  - This should happen in a user-friendly way.
- In more complex applications, there may be a sequence of levels between 3 and 2 where partially-described exceptions have further details added. Eventually every exception must reach a level where a complete description can be given.

Passing objects back and forth



Are exceptions plus filters better?

- This approach is better than:
  - not knowing how important an exceptional condition is.
  - not knowing how to report an exceptional condition to the user.
  - reporting raw exception traces to the user.
- But is it better than where we started, with PL/I's dynamic exception-handling?