Scope ≡ the set of program locations where a given identifier is visible.
Thus scope allows us to "hide" identifier information.

In small applications we manage scope via nesting, but this doesn't extend well.
The object oriented method of handling scope is to label each name as to how public it is going to be.
This is the approach taken by Java classes.

- public — visible to all
- protected — visible to subclasses and other classes in the same package
- default — visible to other classes in the same package
- private — visible to only self

This approach tends to be too restrictive, thus we would like a more general option.
We instead decide to have a separate section of a program with package API's.
This is the approach taken by Java interfaces.
The usage of this method is growing more popular over tine.
OCaml uses this as well; it has structures and signatures:

- structures are analogous to classes and define implementation.
- signatures are analogous to interfaces and define API, for example:

```
(* signature: *)
module type Q = sig type 'a queue;; (* gives the API for a queue *)
(* structure: *)
type 'a queue = struct
  Empty | node of int * 'a * 'a queue
  (* code for queue *)
end;;
```

But OCaml goes beyond Java with something called functors.
These compile the functions from functions to functions.
These let you "edit" the source code to conform to signatures.
Thus you can choose your visible interface!
This is more flexible, but also more complicated.

The elephant in the room in terms of scope is separate compilation.
We want to be able to compile parts of our program separately.
Scope rules are often determined by that goal.

We also want to be able to prevent propagation of errors.
Programming has a hierarchy of errors (from least to most severe):

1. implementation restriction
   The program isn't at fault; it is following the spec.
   The spec, however, limits the implementation.
2. unspecified behavior
   The spec defines a range of acceptable behaviors. We must be portable & robust (able to handle all cases).

```
(eq? '(x) '(x))
; option a: two singleton lists may be created -> #f
; option b: the same object is used -> #t
```

3. signaled error
   an exception is required by the spec

```
(open-input-file "foo") ; with inexistent "foo"
```

4. undefined behavior
   Should NEVER happen.
   Behavior is undefined (implementations are not required to even detect the error)

```
(car 0)
(car '())
```

Programmers often think of exceptions as the standard way of dealing with errors,
BUT we have many methods of dealing with errors:

1. fire programmers
   This does not tend to work, since errors ALWAYS appear.

2. Have the compiler check for mistakes
   This occurs in C:

```
char *p;
...
p = nullptr;
...
*p...; // undefined behavior -- often causes a crash
```

   as well as in OCaml:

```
let p = None or ... in
...
match p with
| None -> xxxxs
| Some x -> ...x...;;
```

   But even this static checking doesn't always work; consider an index exception in C!
   We could define a class including a subset of integers that prevents this if we wanted.

3. List the assumptions made about the arguments to any bit of code
   The callee assumes the preconditions are met, and the caller is responsible for that.

```
char index(char *p, int i) {return p[i];}
// p!= nullptr, 0<= i <= sizeof(array p)
// We can imagine a  language which uses
// preconditions as part of its syntax as follows
char index(char *p, int i)
precondition(p!= NULL \&\& 0 <= i \&\& i <= isizeof(p))
{return p[i];}
```

   This can be implemented via a runtime check such as an assertion.
   Static checking partially does this!

4. Define behavior for all bad inputs, returning a special value.
   The caller is responsible for checking for the value, but it can be passed along indefinitely.

```
char *p = NULL; *p; // returns -128 as a special marker "bad char"
```

   This is not often implemented in this way specifically because it restricts the domain. Most machines use it as in the method of 'floats':

```
double a = 127;
double b = 0;
double c = a/b; // returns infinite (in Java)
double d = c + 7; // sets d to infinity
```

   This is great because it is safe and thus very popular!

5. Do not specifically define behavior on errors.

```
int[3] a;
a[3] = 0; // modifies some random unassuming variable
```

   This is often used in languages where performance is paramount, like C, C++, FORTRAN.

6. Stop the program before it can do more damage.
   I/O — crash in C/ C++ on x86-64/SEASNet (related to the abort() call)

7. Stop the program, but allow the programmer to deal with it.
   This is called **exception handling**.

```
try {
  f(x, y);
  g(w, y);
} except(Exception e) {
  // executed iff f/g throws an Exception
} finally {
  // ALWAYS executed, NO MATTER WHAT
  cleanup();
}
```

This method requires us to change the source code and handle in a structured way

In Java (and many other languages), exceptions form a hierarchy:

Throwable

Error — (not the fault of the programmer, can happen anywhere; ie virtual memory exhaustion)

Exception

IOException

FileNotFoundException

...

Java uses static checking to ensure the user is prepared for any contingency.
Each method must declare any exception it can throw in its signature.

Exception handling is a complication:

```
int x = f(a, b);
// acquire a lock
int y = g(x, b);
// release a lock
// These operations may be performed out of order,
// or we may not return, instead performing a nonlocal jump
//
// For delicate code, exception handling may not be
// our best option; instead we may do:
int x = f(a, b);
// acquire lock
if (x != FAILURE) {
  int y = g(x, b);
  f (y != FAILURE) // release lock
}
```

We would ideally want code that lists all normal cases and follows with the unusual ones.

How are exceptions implemented?

We want to look dynamically through the stack for a stack (down to main if need be).

Thus the catcher pushes a marker, and the thrower walks down the stack looking for the first.

This cannot be done statically.