Names

❖ also called *identifiers*

❖ ad hoc design choices

❖ case sensitivity / insensivity

❖ length + lexical rules (start with letter, alphanumeric, etc)

❖ clashes with *reserved words*, *keywords*
Variables

❖ Originally references to memory locations

❖ now typically placeholders for values of various **types**

❖ for example in functional languages variables can store closures of arbitrary **higher-type**, eg. \( (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \)

❖ Some languages allow you to obtain the memory location where the variable contents are stored

❖ e.g. \&var in C - the \& syntax takes *variables* to *pointers* - virtual memory address of the variable

❖ **Aliasing**: when two variables point to the same memory location. What could go wrong/be unexpected for a programmer?
Binding

- an association between an entity and an attribute
- e.g. between a variable and its type
  - or a variable and its scope
- why does the runtime system need to know the type of a variable?
- the time when things are bound is important and differs from language to language
Static vs dynamic binding

- **static binding** - occurs before execution and remains unchanged during.
- **dynamic binding** - first occurs during execution or changes during.
Static type binding

- sometimes done through explicit type declaration, or through type inference
  - type inference
    - primitive: e.g. in Fortran I, J, K, L, M and N are Integer types, otherwise Real assumed. In Perl $p is a number or a string, @p an array, %p a hash
    - sophisticated: in modern functional languages, explicit type declarations are rare, the compiler can do most of the work (Hindley-Milner type inference, using unification) and gives the most general type, using polymorphism (this is related to Java generics)
Dynamic type binding

- variable bound to a type when assignment is performed at runtime
- a variable’s type can change during execution
- common in scripting languages, e.g. Perl and JavaScript
- type checking done at runtime, which can be costly (both in time and memory)

```plaintext
x = 3.5;
x = [“hello”, “world”];
```
Kinds of variables

- **Static variables**
  - bound to a memory location at initialisation time
  - class variables are static variables in the context of OO

- **Stack variables**
  - memory allocated from run-time stack and bound when declaration is executed, statically typed
  - deallocated when no longer in scope
  - e.g. a variable declared inside a Java method

- **Explicit heap variables**
  - bound with explicitly allocated memory
  - e.g. using `malloc` in C or `new` in Java

- **Implicit heap variables**
  - dynamically assigned during assignment, in languages with dynamic type binding
The scope of a variable is the part of the code in which it can be referenced.

- **Local** variables are declared within a program block, the block is the scope of the variable.

- **Global** variables have global scope, unless they are temporarily hidden by a locally scoped variable with the same name.

- **Static (or lexical) scoping** - the scope of a variable can be determined at compile time.
// a globally-scoped variable
var a = 1;

// global scope
test function one(){
    alert(a);
}

// local scope
function two(a){
    alert(a);
}

// local scope again
test function three(){
    var a = 3;
    alert(a);
}

// Intermediate: no such thing as block scope in javascript
function four(){
    if(true){
        var a = 4;
    }
    alert(a); // alerts '4', not the global value of '1'
}

// Intermediate: object properties
function Five(){
    this.a = 5;
}

// Advanced: closure
var six = function(){
    var foo = 6;
    return function(){
        // javascript "closure" means I have access to foo in here,
        // because it is defined in the function
        // in which I was defined.
        alert(foo);
    }()
}

// Advanced: prototype-based scope resolution
function Seven(){
    this.a = 7;
}

// [object].prototype.property loses to [object].property in the lookup
Seven.prototype.a = -1;
// won't get reached, because 'a' is set in the constructor above.
Seven.prototype.b = 8;
// Will get reached, even though 'b' is NOT set in the constructor.

// These will print 1-8
one();
two(2);
three();
four();

alert( new Five().a);
six();
alert( new Seven().a);
alert( new Seven().b);
Example - OCaml

```ocaml
let add_polynom p1 p2 =
  let n1 = Array.length p1
  and n2 = Array.length p2 in
  let result = Array.create (max n1 n2) 0 in
  for i = 0 to n1 - 1 do result.(i) <- p1.(i) done;
  for i = 0 to n2 - 1 do result.(i) <- result.(i) + p2.(i) done;
  result;;
```

function arguments, scope is the whole `let` body

local variables with scope the expression following the first `in`
Dynamic scope

- determined at runtime, scope depends on control flow
- someone actually thought that this is a good idea... it exists in Perl!

```perl
first();

sub first {
    local $x = 1;
    my    $y = 1;
    second();
}

sub second {
    print "x=", $x, "\n";
    print "y=", $y, "\n";
}
```

The variable $y is a true local variable. It's available only from the place that it's declared up to the end of the enclosing block. In particular, it's unavailable inside of `second()`, which prints "y=", not "y=1". This is called *lexical scope*.

`local`, in contrast, does not actually make a local variable. It creates a new 'local' value for a *global* variable, which persists until the end of the enclosing block. When control exits the block, the old *value* is restored. But the variable, and its new 'local' value, are still global, and hence accessible to other subroutines that are called before the old value is restored. `second()` above prints "x=1", because $x is a *global* variable that temporarily happens to have the value 1. Once `first()` returns, the old value will be restored. This is called *dynamic scope*, which is a misnomer, because it's not really scope at all.

http://perl.plover.com/local.html#5_Dynamic_Scope