Programming Language Concepts
Names, Scopes, and Bindings

Janyl Jumadinova
26 January, 2017
A name is exactly what you think it is
- Most names are identifiers operators
- symbols (like ‘+’) can also be names
A name is exactly what you think it is
– Most names are identifiers operators
– symbols (like ‘+’) can also be names

A binding is an association between two things, such as a name and the thing it names.
A name is exactly what you think it is
- Most names are identifiers operators
- Symbols (like ‘+’) can also be names

A **binding** is an association between two things, such as a name and the thing it names.

The **scope** of a binding is the part of the program (textually) in which the binding is active.
**Binding Time**

is the point at which a binding is created or, more generally, the point at which any implementation decision is made.
Binding

### Binding Time

is the point at which a binding is created or, more generally, the point at which any implementation decision is made.

- language design time program structure, possible type
- language implementation time
  - I/O, arithmetic overflow, type equality (if unspecified in manual)
Other Implementation Decisions

- program writing time
  - algorithms, names
Other Implementation Decisions

- program writing time
  - algorithms, names
- compile time
  - plan for data layout
Other Implementation Decisions

- program writing time
  - algorithms, names
- compile time
  - plan for data layout
- link time
  - layout of whole program in memory
Other Implementation Decisions

- program writing time
  - algorithms, names
- compile time
  - plan for data layout
- link time
  - layout of whole program in memory
- load time
More Implementation Decisions

- run time
  - value/variable bindings, sizes of strings
  - NOTE: run time includes
    - program start-up time
    - module entry time
    - elaboration time (point at which a declaration is first “seen”)
    - procedure entry time
The terms **STATIC** and **DYNAMIC** are generally used to refer to things bound before run time and at run time, respectively.
A Few Links about Bytecode

- Not just used for Java:
  https://en.wikipedia.org/wiki/List_of_JVM_languages
- JVM instruction list: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html
- Understanding Bytecode makes you a better programmer:
- Java Bytecode Fundamentals Blog:
In general, early binding times are associated with greater efficiency.

Later binding times are associated with greater flexibility.
In general, early binding times are associated with greater efficiency.
Later binding times are associated with greater flexibility.
Compiled languages tend to have early binding times.
Interpreted languages tend to have later binding times.
Fundamental to all programming languages is the ability to **name data**
- i.e., to refer to data using symbolic identifiers rather than addresses

```c
double *d = (double *)malloc(8);
*d = 3.14; /* No name is bound to the value 3.14 */
/* The name ''d'' is bound to the ADDRESS containing 3.14 */
```
Fundamental to all programming languages is the ability to **name data** - i.e., to refer to data using symbolic identifiers rather than addresses.

Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names.
Scope Rules - control bindings

- Fundamental to all programming languages is the ability to name data - i.e., to refer to data using symbolic identifiers rather than addresses

- Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names

```c
double *d = (double *)malloc(8);
*d = 3.14; /* No name is bound to the value 3.14 */
/* The name ‘d’ is bound to the ADDRESS containing 3.14 */
```
The period of time from creation to destruction is called the **LIFETIME** of a binding.
Lifetime and Storage Management

- The period of time from creation to destruction is called the **LIFETIME** of a binding.
- If object outlives binding it’s **garbage**.
- If binding outlives object it’s a **dangling reference**.
Lifetime and Storage Management

- The period of time from creation to destruction is called the **LIFETIME** of a binding.
- If object outlives binding it’s **garbage**.
- If binding outlives object it’s a **dangling reference**.
- The textual region of the program in which the binding is active is its **scope**.
Lifetime and Storage Management

Storage Allocation mechanisms

- Static
- Stack
- Heap
Lifetime and Storage Management

Storage Allocation mechanisms
- Static
- Stack
- Heap

Static allocation for
- code
- globals
- static or own variables
In C, variables can be global (visible to any function)

```c
int i; /* i is global */
int f(int x) {
    return i+x; /* i is visible inside function f */
}
main() {
    int j;  /* j is visible only within main */
i = 10;  /* i is visible inside function main */
j = 99;
j = f(3); /* sets j to 13 */
}
Static Example

When we compile this, \( i \) is stored in a fixed location, while \( j \) is allocated on the stack.

(C-to-ARM assembly from [http://assembly.ynh.io/](http://assembly.ynh.io/))
Two Types of Scoping

**Static scoping** (also called “lexical scoping”)
- most familiar (Java, C)
- scope of variables known at compile time
Two Types of Scoping

**Static scoping** (also called “lexical scoping”)
- most familiar (Java, C)
- scope of variables known at compile time

**Dynamic scoping**
- scope depends on order of function calls at execution time
- pretty rare nowadays
Static Scope Example (Java)

```java
public static int x, y;
public static void main(String[] args) {
    x = 10; y = 20;
test();
    System.out.println("x = " + x + ", y = " + y);
}

System.out.println("x = " + x + ", y = " + y);
}

public static void test() {
    int x = 50, y = 60;
    System.out.println("x = " + x + ", y = " + y);
}
```

Output:
- x = 50, y = 60
- x = 70, y = 80
- x = 10, y = 20

Scope of x, y
Scope of x, y ("hole" in the scope of x, y)
Scope of x, y ("hole" in the scope of x, y)
public static int x,y;
public static void main(String[] args) {
   x = 10; y = 20;
   test1();
}
public static void test1() {
   int x = 50, y = 60;
   test2();
}
public static void test2() {
   System.out.println("x = " + x + ", y = " + y);
}
What Happens in Dynamic Scoping?

```java
int x, y;
start() {
    x = 10; y = 20;
test1();
test2();
}
test1() {
    int x = 50, y = 60;
test2();
}
test2() {
    System.out.println("x = " + x + ", y = " + y);
}
```

Outputs "x = 50, y = 60"

Outputs "x = 10, y = 20"
In-Class Exercise

JavaScript Case Study - See Handout in the repository