Variables and Scopes

CS 430 :: Spring '19

Prof. Bowers

What is a variable?

```
int x = 5;
```

What do we need to know about the variable above to use it / make this code work?

Variables

A **variable** is an abstraction of a memory cell.

1. name
2. address(**r-value**)
3. value (**l-value**)
4. type
5. lifetime
6. scope

How many variables are involved in the following?

```
Object o = new Object()?
```

Variables

A **variable** is an abstraction of a memory cell.

1. name
2. address (**r-value**)
3. value (**l-value**)
4. type
5. lifetime
6. scope

Think about the following assignment:

```
x = y = 5;
```

Why is the '=' right-associative? Why doesn't `5 = y` work? Why are the address and value called the r-value and l-value?

Binding

Binding refers to making an association between an attribute and an entity. Binding can happen at language design time, language implementation time, compile time, load time, link time, or run time.

```
int count = getUserInput();
count = count + 1;
```

- Type of `count` is bound at compile time.
- Value of `count` is bound at run-time.
- Meaning of `+` is bound at compile-time.
- Representation of 1 is bound at compiler design-time.
- Possible values of `count` are bound at compiler design-time.

Binding Times

Bindings can be either **static** meaning they occur before runtime and cannot be changed during program execution or **dynamic** meaning they occur during runtime and are changeable during execution.

Examples of type bindings.

Java (static type binding)

```
int x = 5; // x is now an int forever and always
x = "hello"; // Compiler error, x is an int
```

JavaScript (dynamic type binding)

```
var x = 5; // x is an int.
x = "hello"; // Now it's a String.
```

Java 10 (Implicit static type binding)

```
var x = 5;
x = "hello"; // Compiler error, x is an int
```

Kotlin (Same as Java 10 but no ;-))

```
var x = 5
x = "hello" // Compiler error, x is an int
```

Evaluation of Dynamic type binding.

- Causes less reliability in programs.
- Type checking occurs at run-time leading to
  - Slower programs
  - More memory intense programs.

Variable allocation

Variables are **allocated** when the variable is bound to memory and **deallocated** when the binding is released.

The **lifetime** of a variable is the time between allocation and deallocation.

- `static` variables are allocated before program execution and remain bound until program termination.

```
class Foo { static int bar = 5; }
```

- **Stack-dynamic variables** are allocated as part of the stack frame when the frame is pushed to the stack on a function/method call

```
int foo() {
  int x = 5;
  System.out.println(7);
  int y = 8;
  System.out.println(x + y);
}
```

- **Heap-dynamic variables** are allocated on the heap.
  - Explicitly with `new`:

```
x = SomeClass.new # Ruby heap allocation
```

  - Implicitly with literals.

```
x = [1, 2, 3, 4] # Python list allocation
```

Scope

- Variables are **local** in the program unit or block in which they are declared.
- A variable is **visible** if it can be referenced.

```
public class X {
  public int x = 5;
  public void foo() {
    int y = 7;
    // What variables are visible?
    // What variables are local?
  }
}
```

Scope

- Variables are **local** in the program unit or block in which they are declared.
- A variable is **visible** if it can be referenced.

```
public class X {
  public int x = 5;
  public void foo() {
    if ("".equals("")) {
      int y = 5;
      // What variables are visible?
      // What variables are local?
    }
    int y = 7;
  }
}
```

Scope: Shadowing

- Variables are **shadowed** if their name is the same as another visible variable. How do we resolve these issues?

```
public class X {
  public int x = 5;
  public void foo() {
    int x = 7;
    System.out.println("What is this? " + x);
  }
}
```

Referencing environment

The **referencing environment** at a point in the program is the list of variables visible at that point (non-shadowed).

```
public class X {
  public int x = 5, int y = 7;
  public void foo() {
    int x = 7;
    // Point A
  }
}
```

Referencing environment

```
public class X {
  public int x = 5, int y = 7;
  public void foo() {
    int x = 7;
    // Point A
  }
}
```

Referencing environment at `Point A` is `{X.foo.x, X.y}`.

Example: Shadowed global variable in C++

```cpp
#include <iostream>

int x = 5;

int main() {
        int x = 6;
        std::cout << x << std::endl;
        std::cout << ::x << std::endl;
        return 0;
}
```

Example: Python::Function-level Scope

```
def foo():
  x = 5
  if x == 5:
    y = 3
  print x + y # y is in scope

foo() # prints out 8
print x + y # Error: x and y are stack-dynamic and in foo
```

Example: Python::Global Variables::establishing a local variable

```
x = 5
def bar():
  print(x)

def baz():
  x = 7
  print(x)

bar()
baz()
print(x)
```

What get's printed?

Example: Python::Global Variables::establishing a local variable

```
x = 5
def bar():
  print(x)

def baz():
  x = 7
  print(x)

bar()
baz()
print(x)
```

Output:

```
5
7
5
```

Example Python::Global Variables::global keyword

```
x = 5
def bam():
  global x # Global keyword tells python to use the globa
  x = 7
bam()
print(x)
```

Output:

```
7
```

Example Python::Global Variables::can't mix and match

```
x = 5
def hipster():
        print(x)
        x = 4
        print(x)
hipster() # Error hipster() tried to use
          # x before it was cool
```

Block scopes vs. Function scopes

```
int foo() {
  int x;
  if (someTest()) { x = 5; }
  else     { x = 7; }
  return x;
}
```

```
def foo()
  if someTest()
    x = 5
  else
    x = 7
  end
  return x
end
```

Block scopes vs. Function scopes

Bad Java:

```
int foo() {
  if (someTest()) { int x = 5; }
  else     { int x = 7; }
  return x; // Compiler error
}
```

If Ruby had block level scoping:

```
def foo()
  x = 0 # Dummy declaration.
  if someTest()
    x = 5
  else
    x = 7
  end
  return x
end
```

Nested procedures::Python Example 1

```
def foo():
  x = 5
  def bar():
    x = 7
  bar()
  print(x)

foo() # prints 5
```

Nested procedures::Python Example 2

```
def foo():
  a = 5
  b = 7
  def bar():
    b = 8
    print(a, b) # prints 5 8
  bar()
  print(a, b) # prints 5 7
```

Nested procedures::Python Example 2

```
def foo():
  a = 5
  b = 7
  def bar():
    print(a, b) # error variable 'b' referenced before as
    b = 8
  bar()
  print(a, b)
```

Nested procedures::Javascript 2 (Explicit Declaration)

```
function foo() {
        var x = 5;
        function bar() {
                var x = 7;
        }
        bar();
        alert(x);
}
foo(); // Alerts 5
alert(x); // Error
```

Nested procedures::Javascript 2 (Explicit Declaration)

```
function foo() {
        var x = 5;
        function bar() {
                x = 7;
        }
        bar();
        alert(x);
}
foo(); // Alerts 7
alert(x); // Error
```

Nested procedures::Javascript 3 (Implicit Declaration)

```
function foo() {
        x = 5;
        function bar() {
                x = 7;
        }
        bar();
        alert(x);
}
foo(); // Alerts 7
alert(x); // Alerts 7 (variable x is global)
```

Cons of this approach?

Static vs. Dynamic Scoping

- **Static scoping** is determined by the structure of the program and can be determined by looking at the code.
- **Dynamic scoping** is determined by the call-stack and can only be determined at run-time. (You have to work backwards through the call stack to find first reference to a variable of that name.)

Static vs. Dynamic Scoping Examples

```
 1 func main() {
 2   var x = 5
 3       y = 2
 4   // location A
 5   func g() {
 6     var x = 12
 7         z = 8
 8     f() // Location B
 9   }
10   func f() {
11     println(x) // Location C
12   }
13   g()
14 }
```

What is the output with static scope? What is the output with dynamic scope? What are the referencing environments at A, B, and C in both cases?