**Scoping Rules Affect Program Meaning**

We define several different *scoping rules*, rules that define which variable is visible where.

We show

* the same program can have different meaning under different scoping rules
* scoping rules can affect which programs are considered valid

**Standard Static Scoping**

Consider the following example:

class World {

int sum;

int value;

void add() {

sum = sum + value;

value = 0;

}

void main() {

sum = 0;

value = 10;

add();

if (sum % 3 == 1) {

int value;

value = 1;

add();

print("inner value = ", value);

print("sum = ", sum);

}

print("outer value = ", value);

}

}

**Question:** What is the result of invoking ‘main’ according to standard Java rules *(put ‘static’ in front of class methods and fields, replace ‘print’ with ‘System.out.println’)*?

Answer:

inner value = 1

sum = 10

outer value = 0

With static scoping (also called ‘lexical scoping’) rules, simple rules determine which identifier refers to which symbol:

* identifier refers to the symbol that was declared closes to the place **in program text**

**We will assume static scoping** (unless otherwise specified).

Below we see some alternatives, to understand what those could be in principle.

**Dynamic Scoping**

In **dynamic scoping**, symbol refers to the variable that was most recently declared within program execution.

Dynamic scoping views variable declaration as a statement that establishes which symbol is considered to be the ‘current one’.

**Question:** What is the result of the same example?

Answer:

inner value = 0

sum = 11

outer value = 0

Dynamic scoping used to be popular in languages like LISP, but it can make programs difficult to understand. Sometimes it can lead to shorter code.

Dynamic scoping can be **simulated** in languages with static scoping by explicitly referring to **maps** in our program.

class World {

[Map](http://www.google.com/search?hl=en&q=allinurl%3AMap+java.sun.com&bntI=I%27m%20Feeling%20Lucky) env; // implemented like e.g. Hashtable<String, Stack<Int>>

// env.declareNew(id) --> env.get(id).push(0)

// env.get(id) --> env.get(id).top()

// env.pop(id) --> env.get(id).pop()

// env.update(id, v) --> {env.get(id).pop(); env.get(id).push(v);}

void add() {

env.update("sum", env.get("sum") + env.get("value"));

env.update("value", 0);

}

void main() {

env.update("sum", 0);

env.update("value", 0);

add();

if (env.get("sum") % 3 == 1) {

env.declareNew("value");

env.update("value", 1);

add();

print("inner value = ", value);

print("sum = ", env.get("sum"));

env.pop("value");

}

print("outer value = ", env.get("value"));

}

{

env = new ...

env.declareNew("sum");

env.declareNew("value");

}

}

**Giving Priorities to Local vs Global Variables**

We could imagine that, if there is both a global and a local variable, we give the priority to the global variable.

Consider the following program.

class World {

int sum;

int value;

void main() {

sum = 0;

value = 10;

if (value > 5) {

int value;

value = 1;

print("value = ", value);

sum = sum + value;

}

print("sum = ", sum);

}

}

What would the result be with standard Java scoping rules?

Answer:

value = 1

sum = 1

What would the result be if global variables take precedence?

Answer:

value = 1

sum = 1

We prefer to use the closest declaration (unless otherwise specified)

**Multiple Independent Scopes**

Some languages (including ML, Java, Scala) use independent scopes for different purposes.

In principle, one could use different scopes for:

* scope for variables
* scope for method names
* scope for user-defined types (e.g. classes)

In such a language, we can have:

class World {

int sum;

int value;

void sum() {

sum = sum + value;

value = 0;

}

void main() {

sum();

}

}

The compiler knows whether ‘sum’ refers to method or variable according to the place where the identifier occurrs in the syntax tree (in the case above: whether it has parentheses () afterwards or not).

We will avoid such multiple scopes (unless otherwise specified), and expect the compiler to reject such declarations.