Mechanics of Procedure Stack (In no particular language)

Review of how Procedure Call Stack and Heap Work

No particular language but typical of

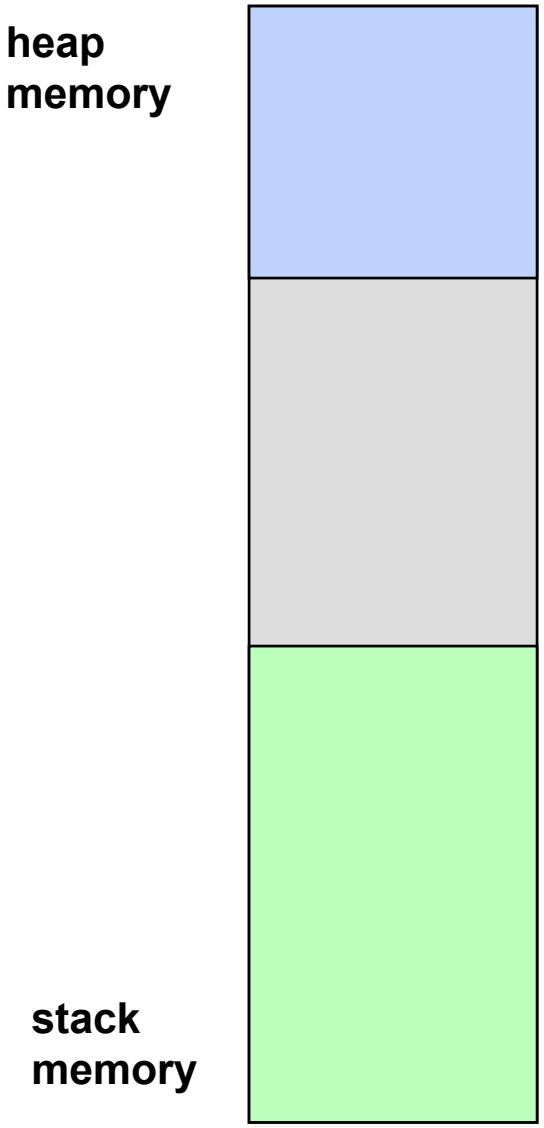
- Java, C, C++ (languages taught in 1st year)

- Here we review what is probably your understanding of how function and procedure calls work (probably as was taught to you in first year programming courses.)

- We also consider "objects" that are typically allocated on the heap, as opposed to procedure stack.

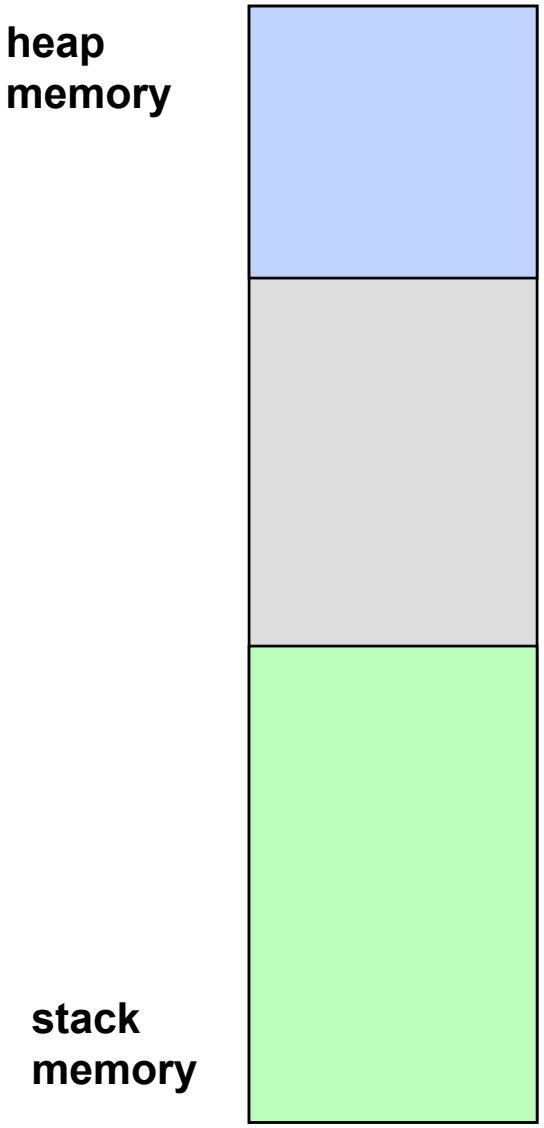
- Our reason for doing this is to see if this model will still hold up for Javascript. In particular Javascript's notion of Closures

Stack and Heap



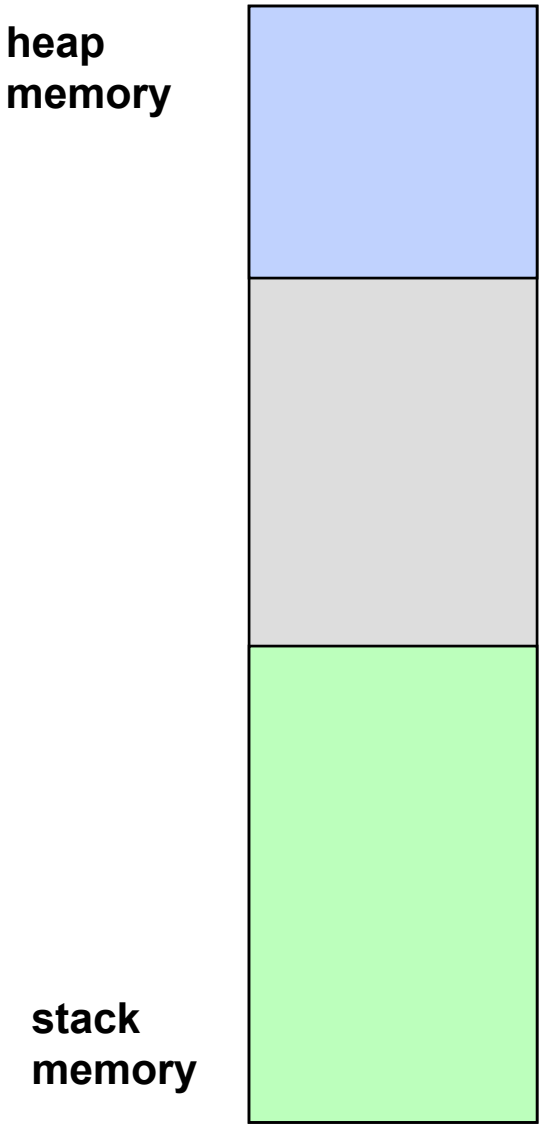
- When a program runs the OS grants it some memory to work within.
- The memory is used in two distinct regions: the stack and the heap.
- A typical way to implement the stack and the heap is to use the contiguous allocated memory and grow the stack from one end and the heap from the other.
- If the two ever collide an “out of memory” or “stack overflow” run-time error results.
- Memory can overflow because of a bug, or because a program simply needs more memory than is available.
- Some programing techniques, like recursion, can easily get out of hand and use up a lot of stack space to represent sub-problems.

Review of how Procedure Call Stack and Heap Work



- The heap is used for items, most often created with `new`, that have a lifetime independent of the function call that created them.
- Items on the heap remain there until they are explicitly deleted.
- In non-garbage collected languages, like C++, C, heap items are removed by the programmer explicitly.
- In garbage collected languages like Java, heap items are removed by a garbage collector when they are no longer accessible.
- Garbage collectors are often viewed as slower but safer.
- There is no specific order to where things are placed in the heap. (In fact a "heap manager" might run in the background and continuously re-arrange and defragment the heap.)

Review of how Procedure Call Stack and Heap Work



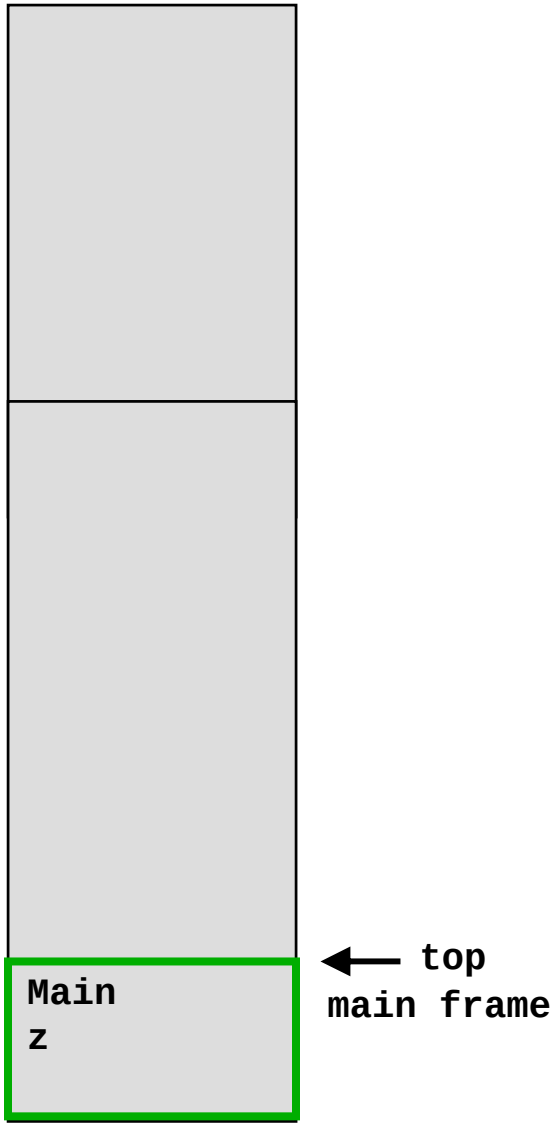
- The stack is used to implement procedure, or function, calls.
- When a procedure is called a "stack frame" representing that procedure is placed on top of the stack; when the procedure returns its stack frame is popped off the stack.
- The stack grows and shrinks in a "first in – last out" sequence.
- So the lifetime of stack frames matches exactly the calls and returns of the procedures they represent. (By contrast, the lifetime of items on the heap is independent of the function calls that placed them there.)
- Since procedures can be called at any time and from anywhere, they cannot make assumptions about what is "below them" on the stack.

Review of how Procedure Call Stack and Heap Work

- **Example: Function calls with only parameter passing, return values and primitive local variables**

Mechanics of Procedure Stack (no particular language)

heap memory

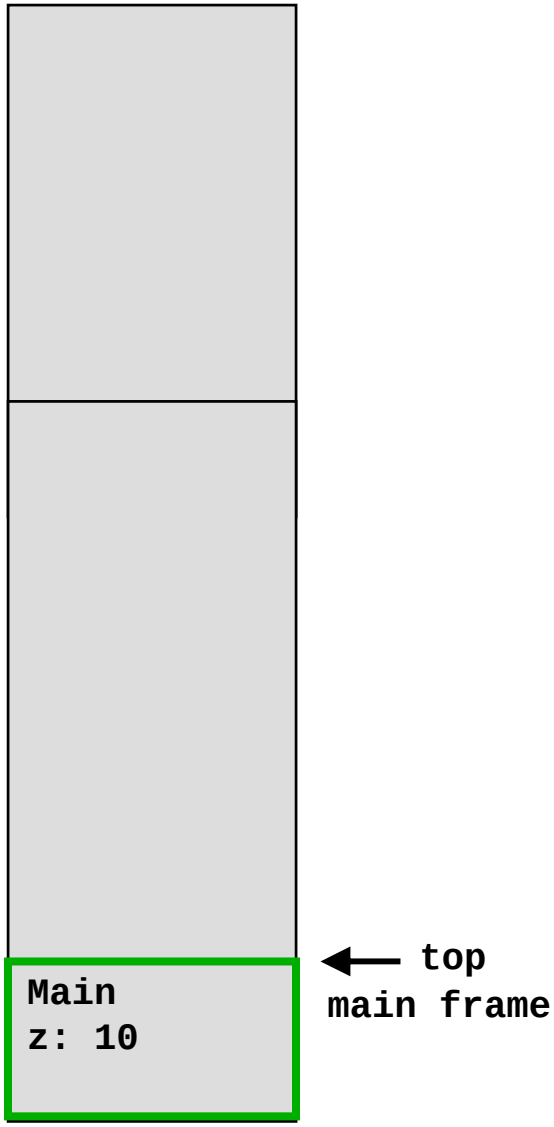


stack memory

```
int g(int i) {
     int j = i*i;
     return j;
  }
 int f(int x) {
    int y = g(x + 2);
    return x + y;
 }
 main(){
   int z;
   z = 10;
   z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)

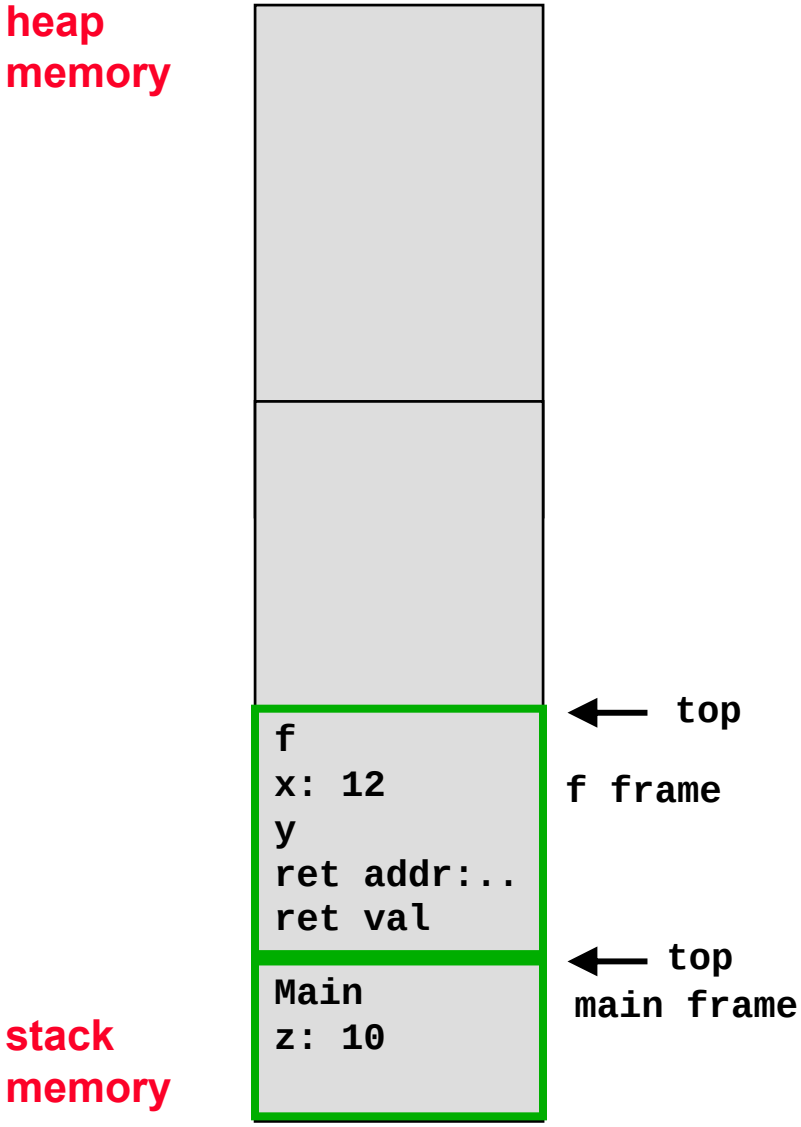
heap memory



stack memory

```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

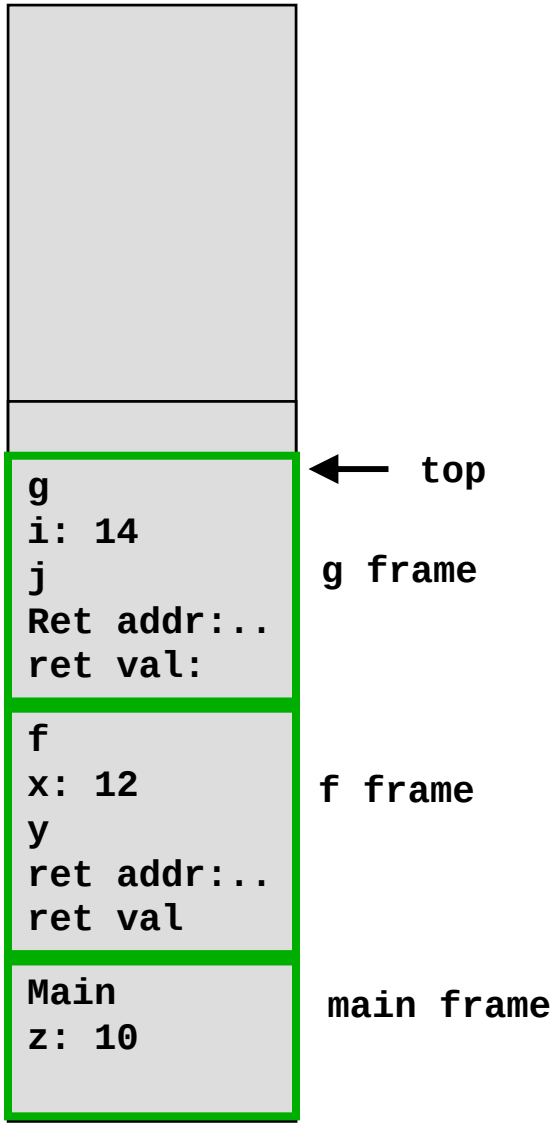
Mechanics of Procedure Stack (no particular language)




```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)

heap memory

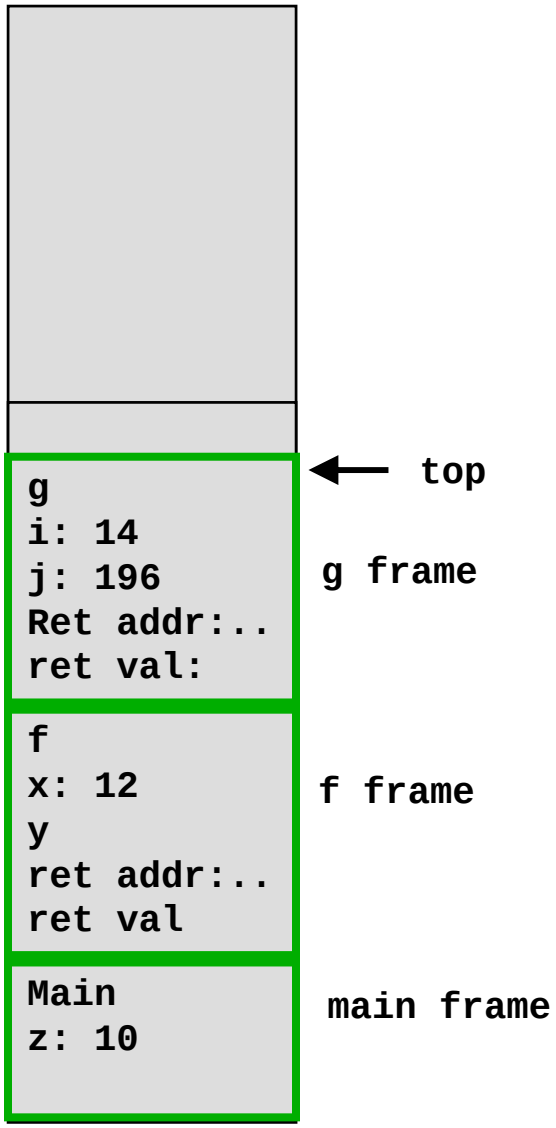


stack memory

```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
  print(z);
}
```

Mechanics of Procedure Stack (no particular language)

heap memory

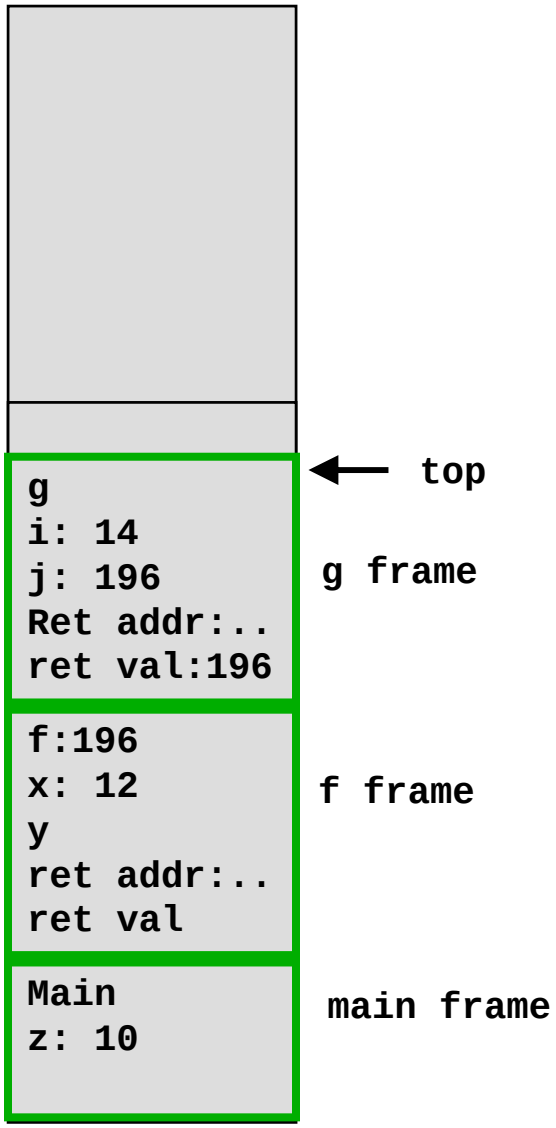


stack memory

```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)

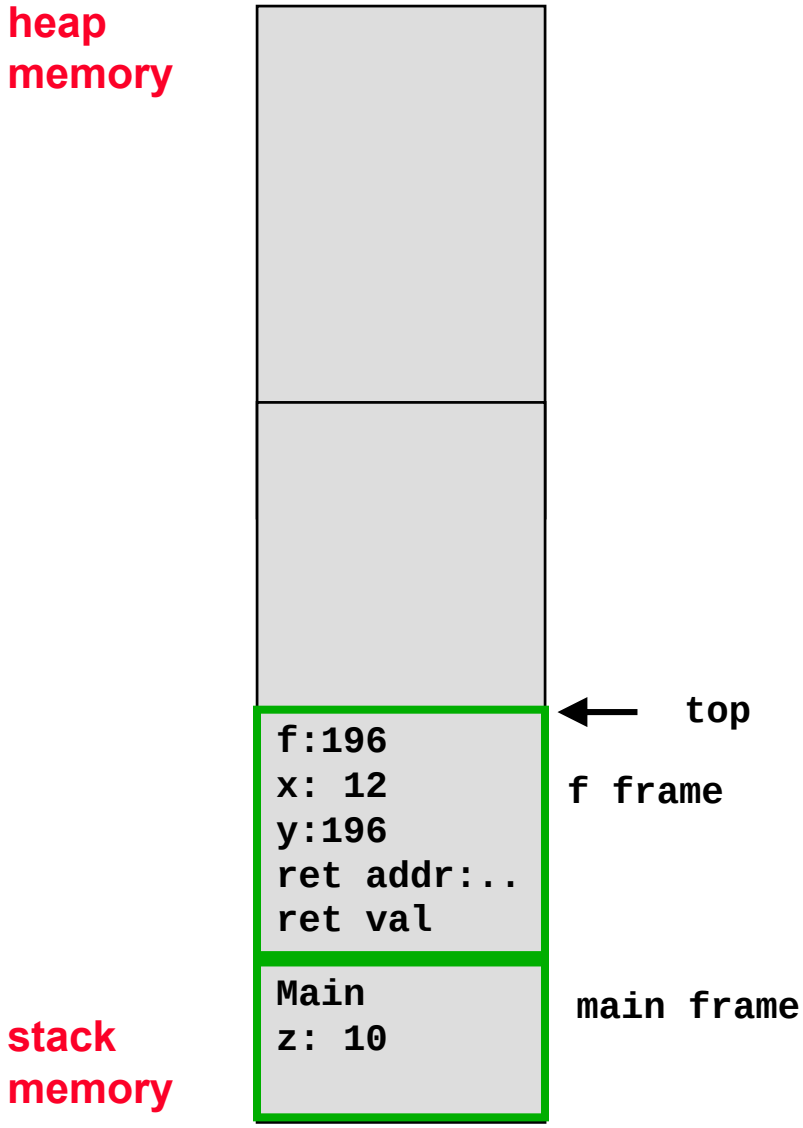
heap memory



stack memory

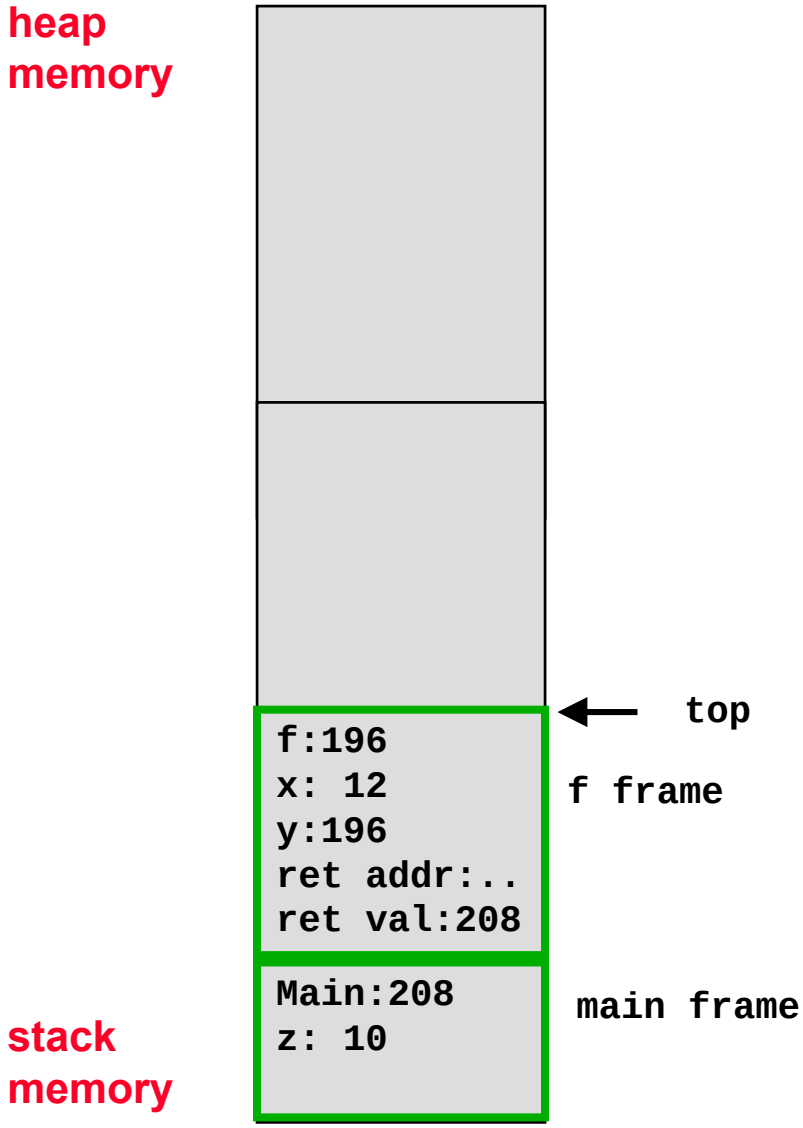
```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)




```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

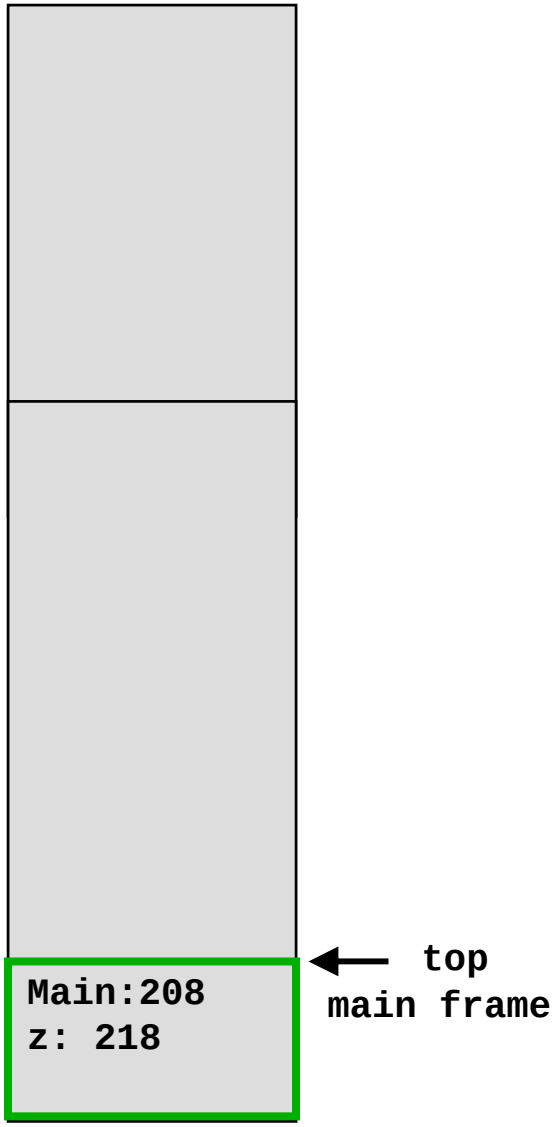
Mechanics of Procedure Stack (no particular language)




```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)

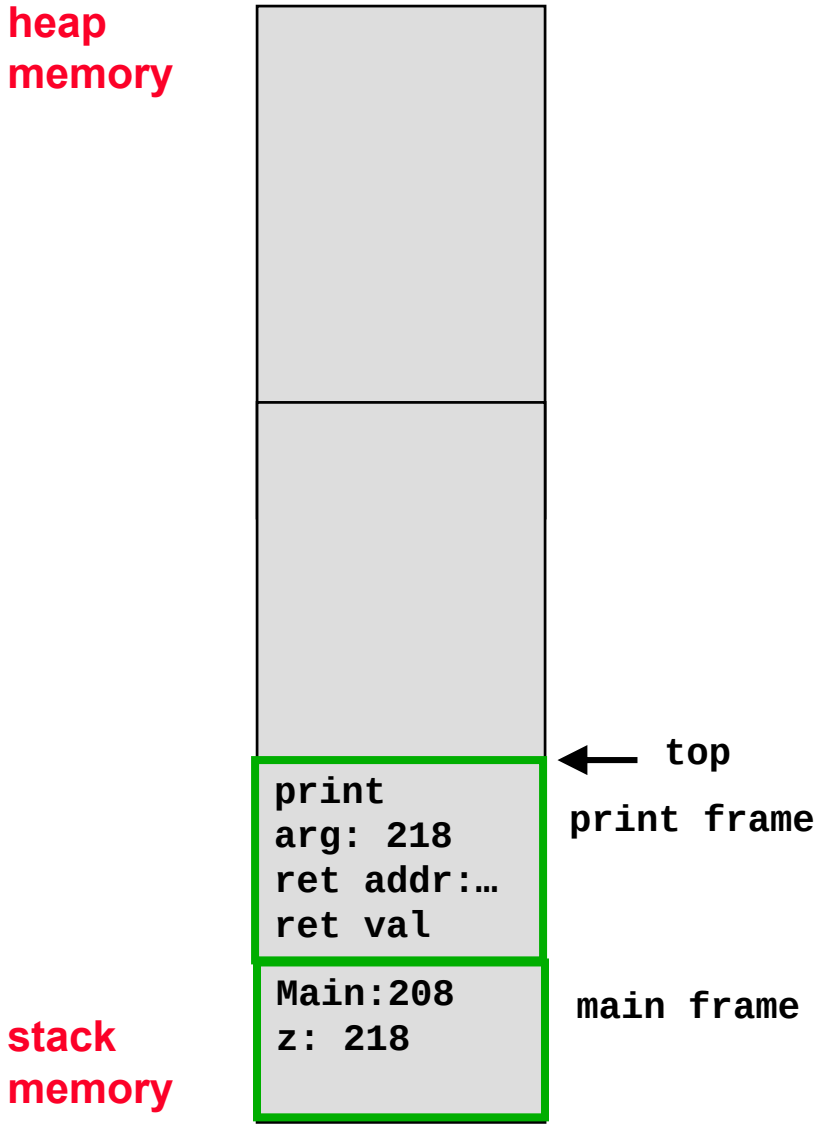
heap memory



stack memory

```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);
}
```

Mechanics of Procedure Stack (no particular language)

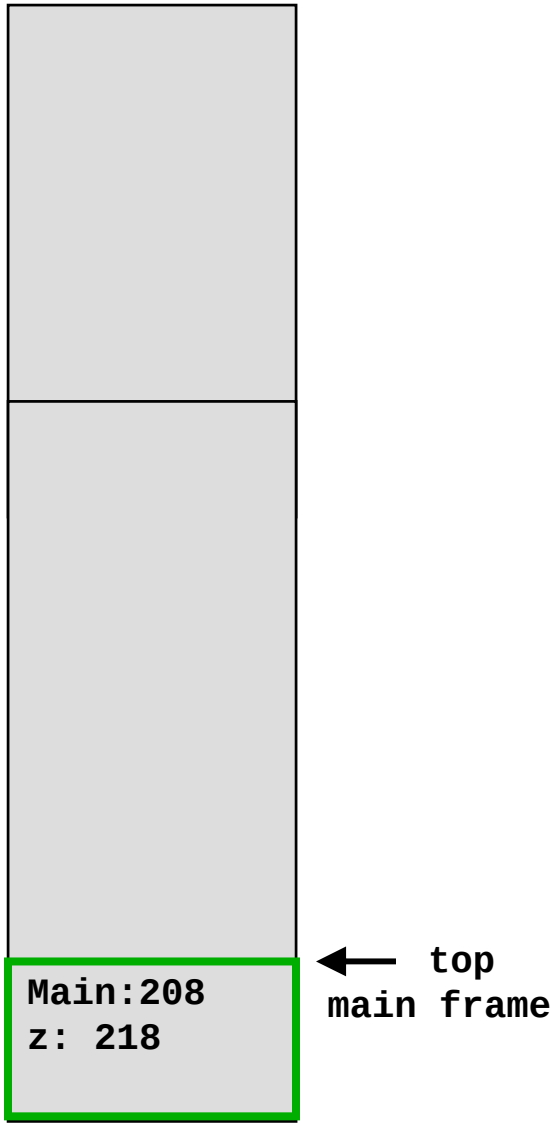



```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
   print(z);

}
```

Mechanics of Procedure Stack (no particular language)

heap memory



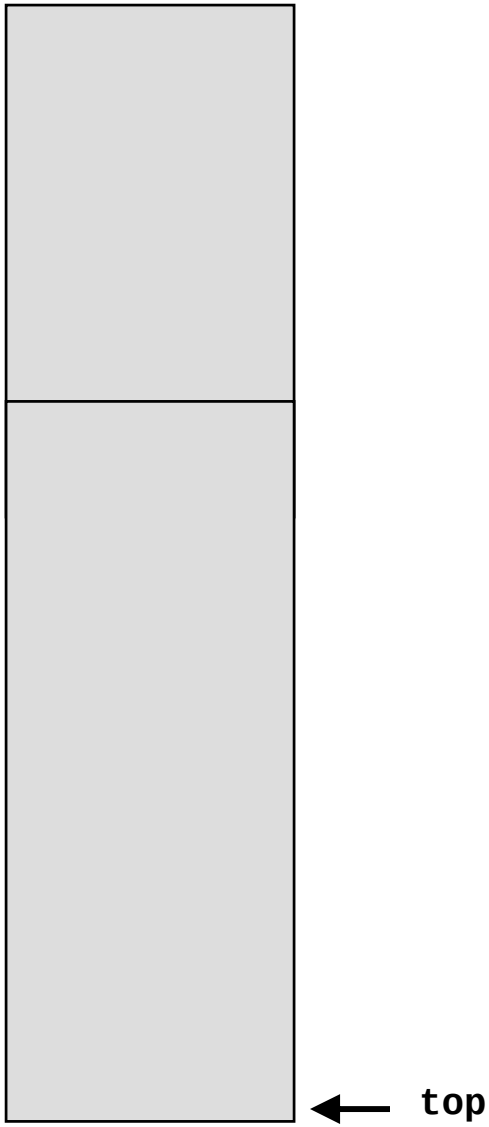
stack memory

```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
  print(z);

}
```

Mechanics of Procedure Stack (no particular language)

heap memory



stack memory

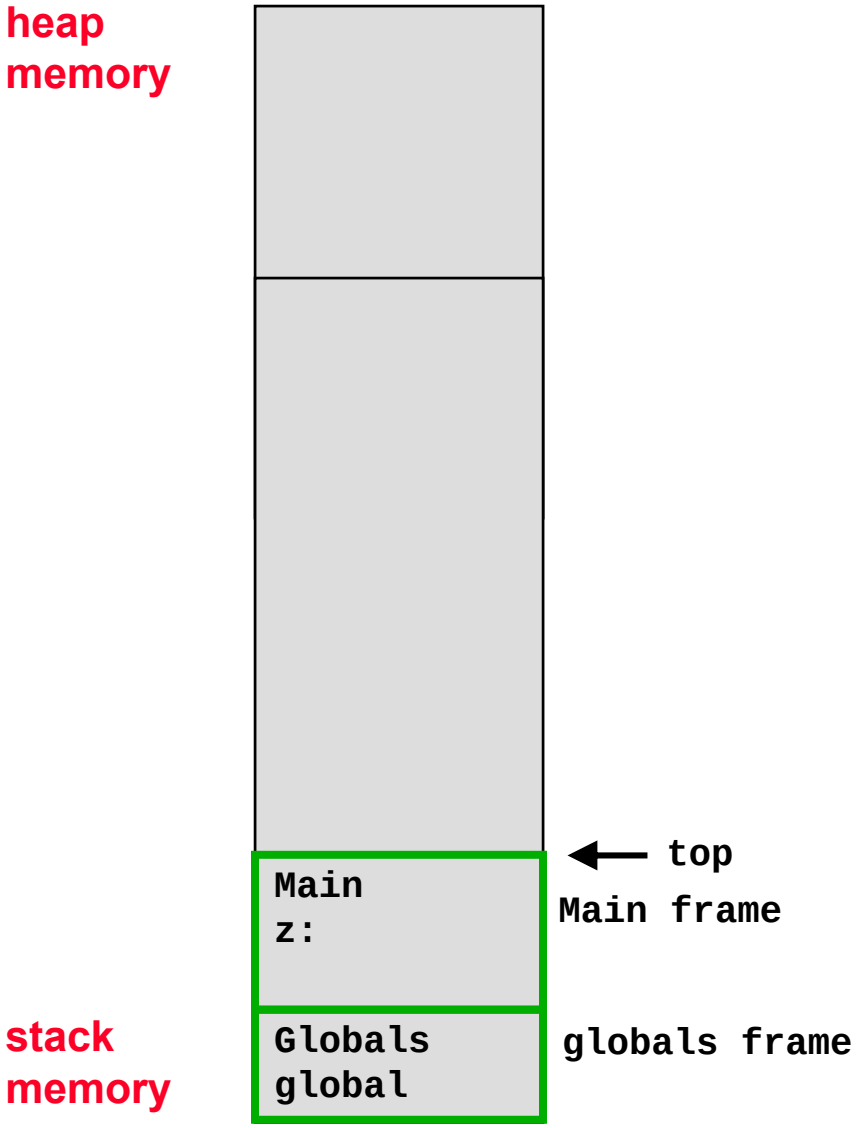
```
int g(int i) {
    int j = i*i;
    return j;
 }
int f(int x) {
   int y = g(x + 2);
   return x + y;
}
main(){
  int z;
  z = 10;
  z = f(z+2);
  print(z);

}
```

Review of how Procedure Call Stack and Heap Work

- **Accounting for globally scoped variables**

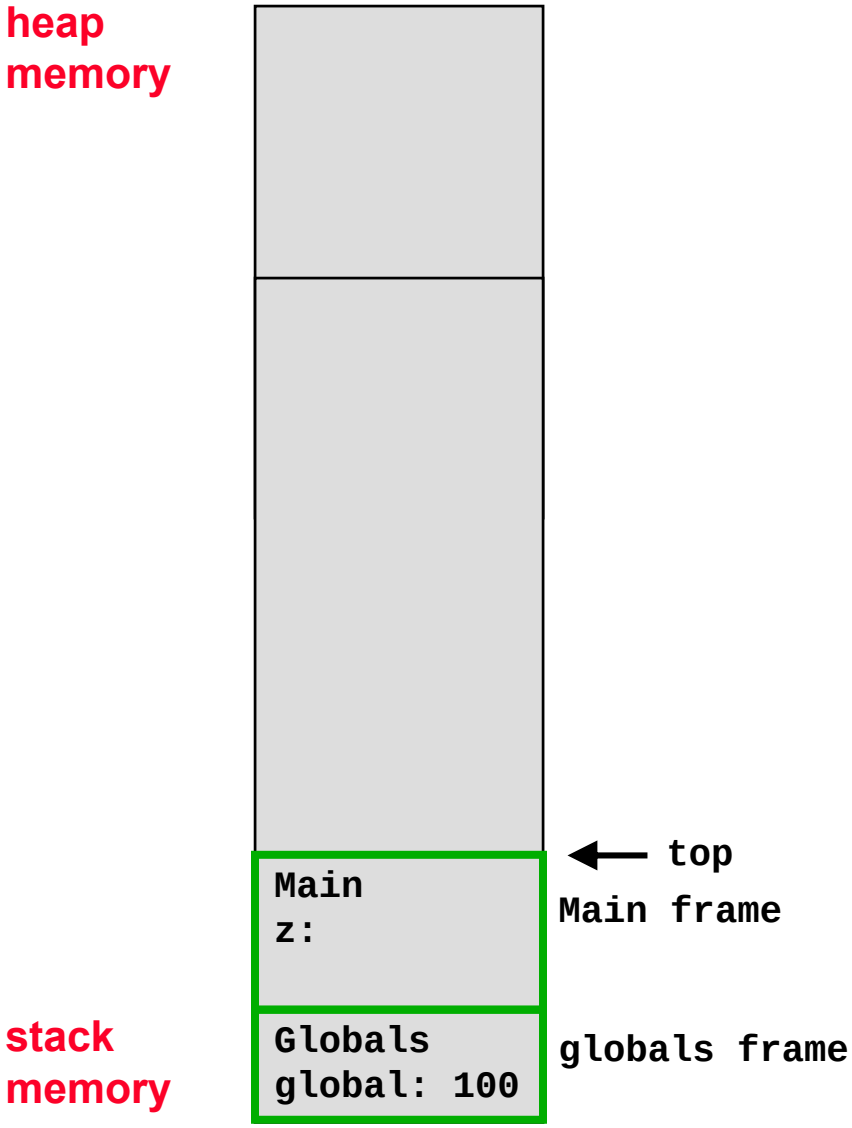
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

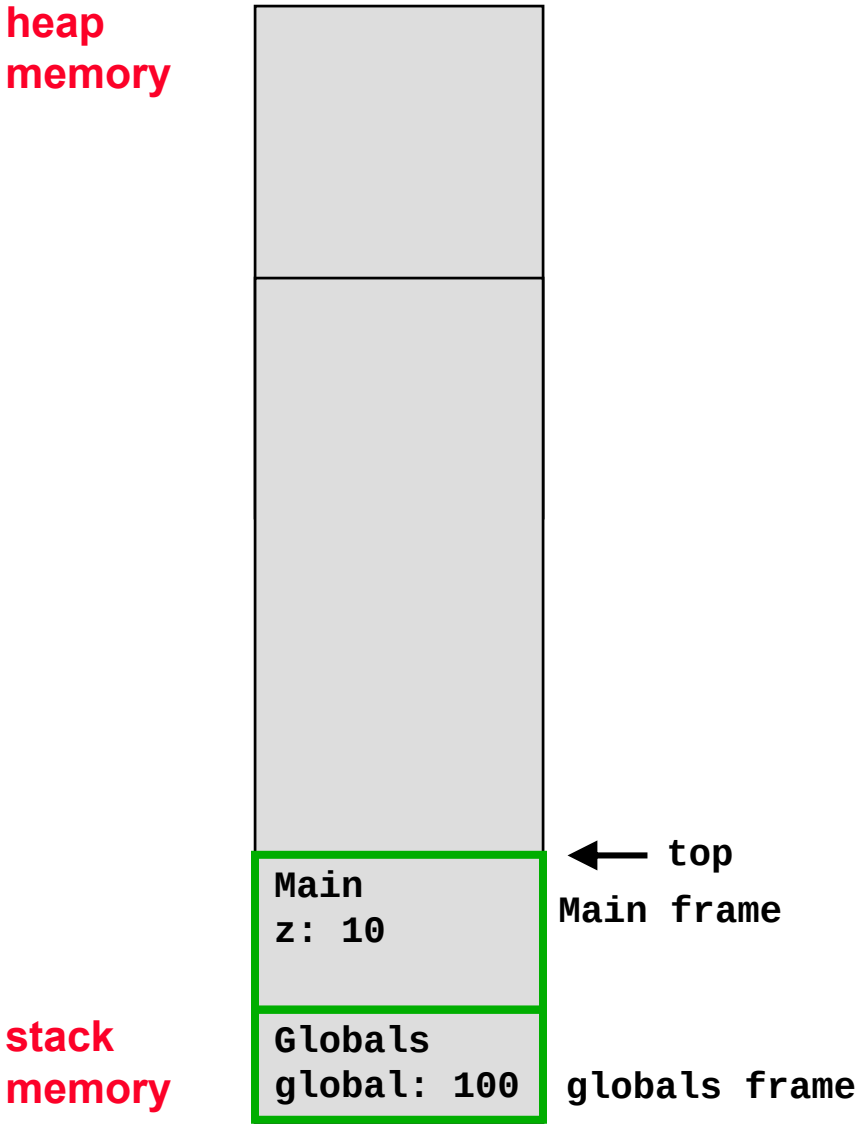
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

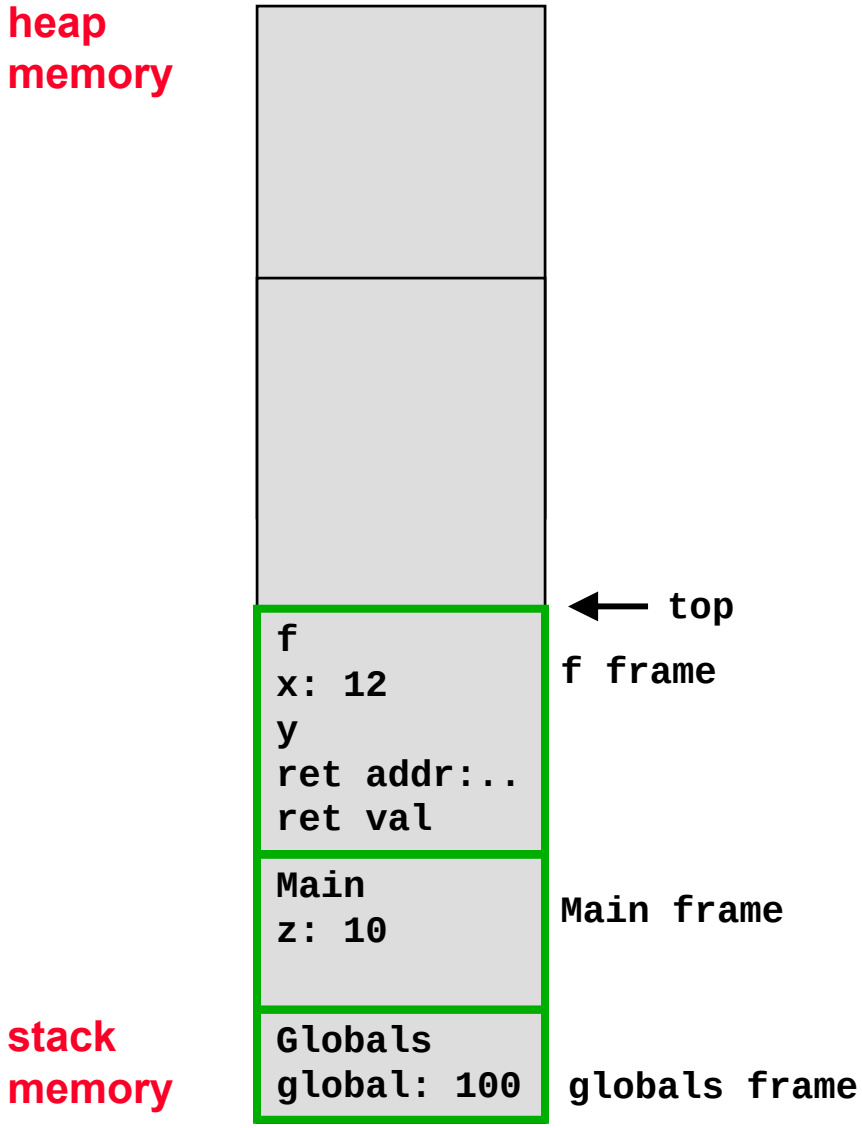
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

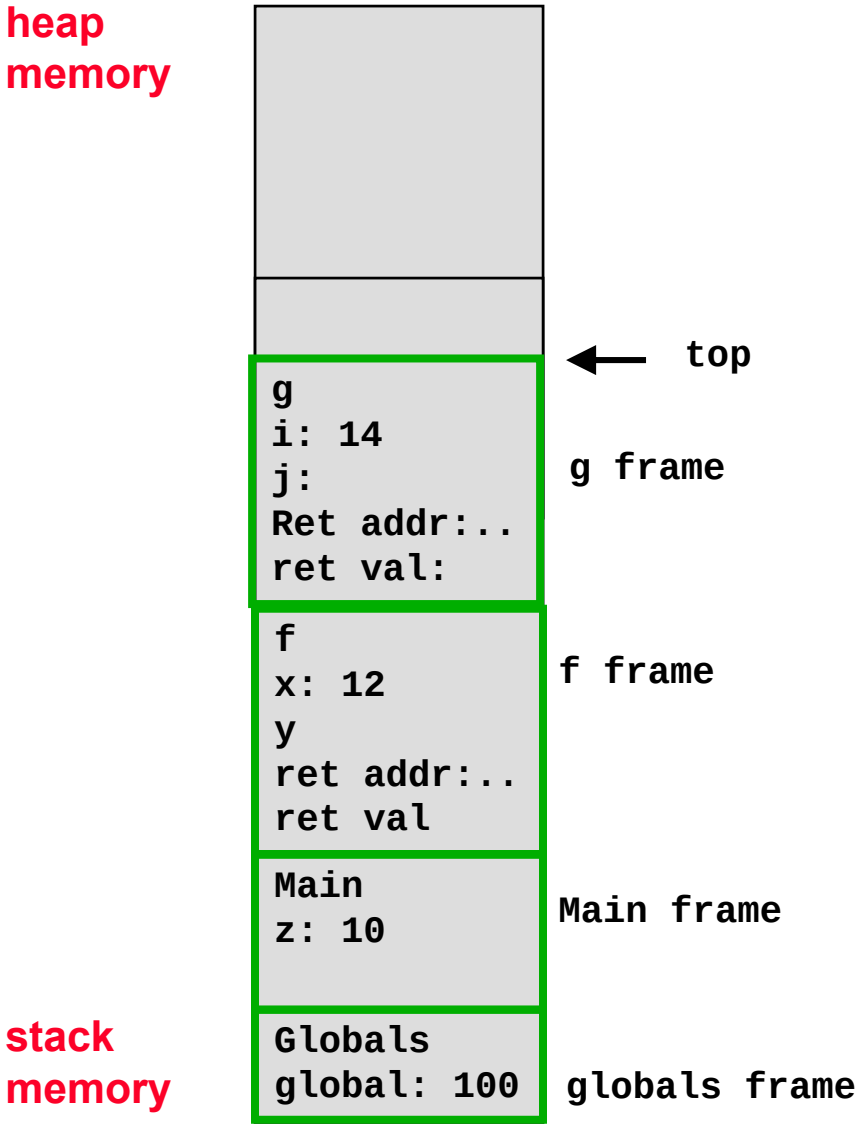
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

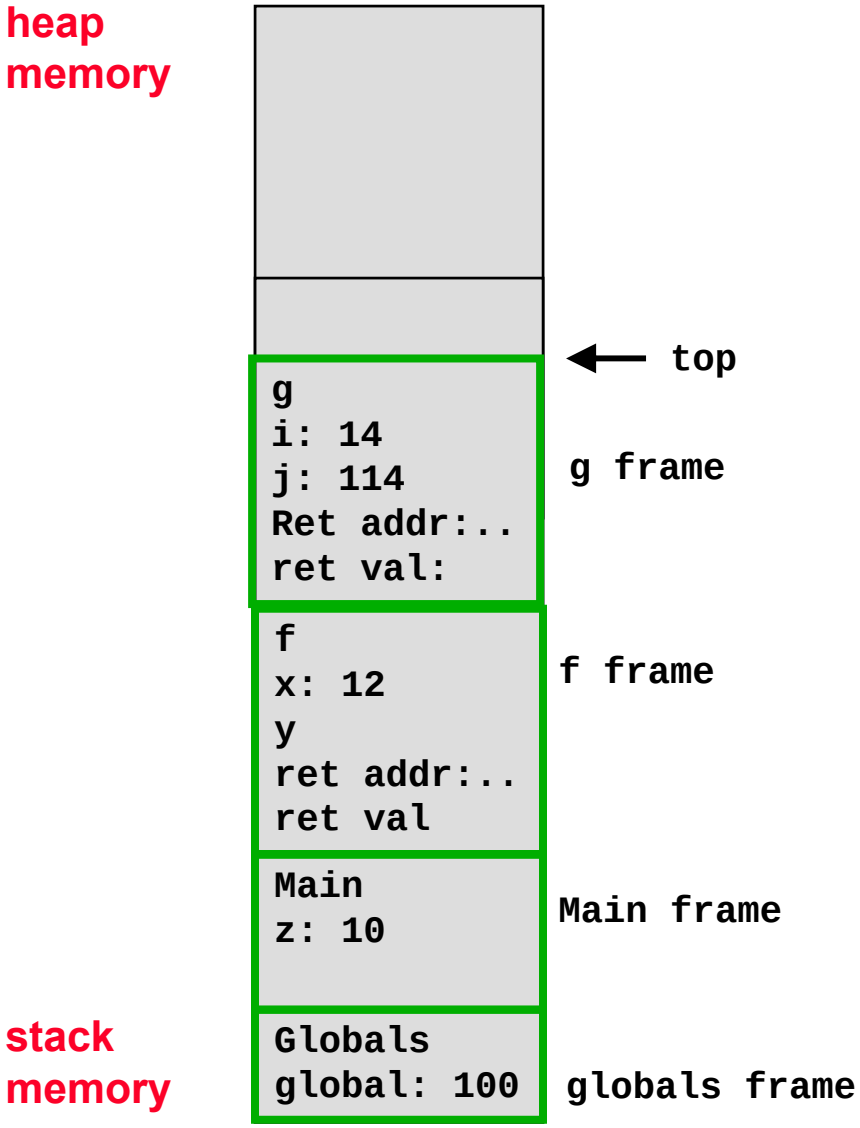
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

Mechanics of Procedure Stack (no particular language)

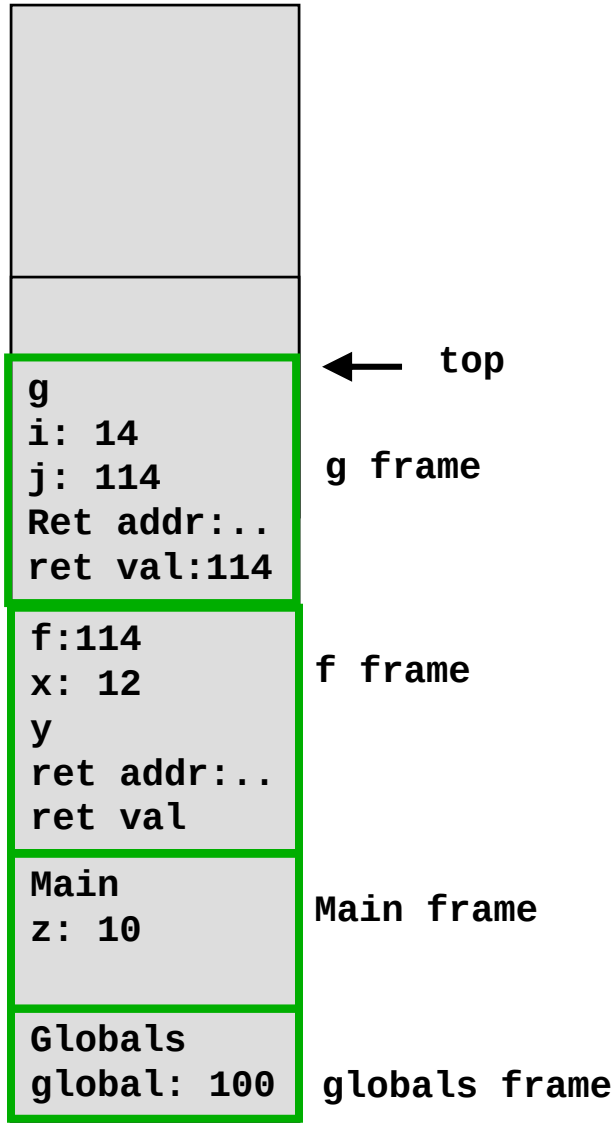



```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

Mechanics of Procedure Stack (no particular language)

heap memory

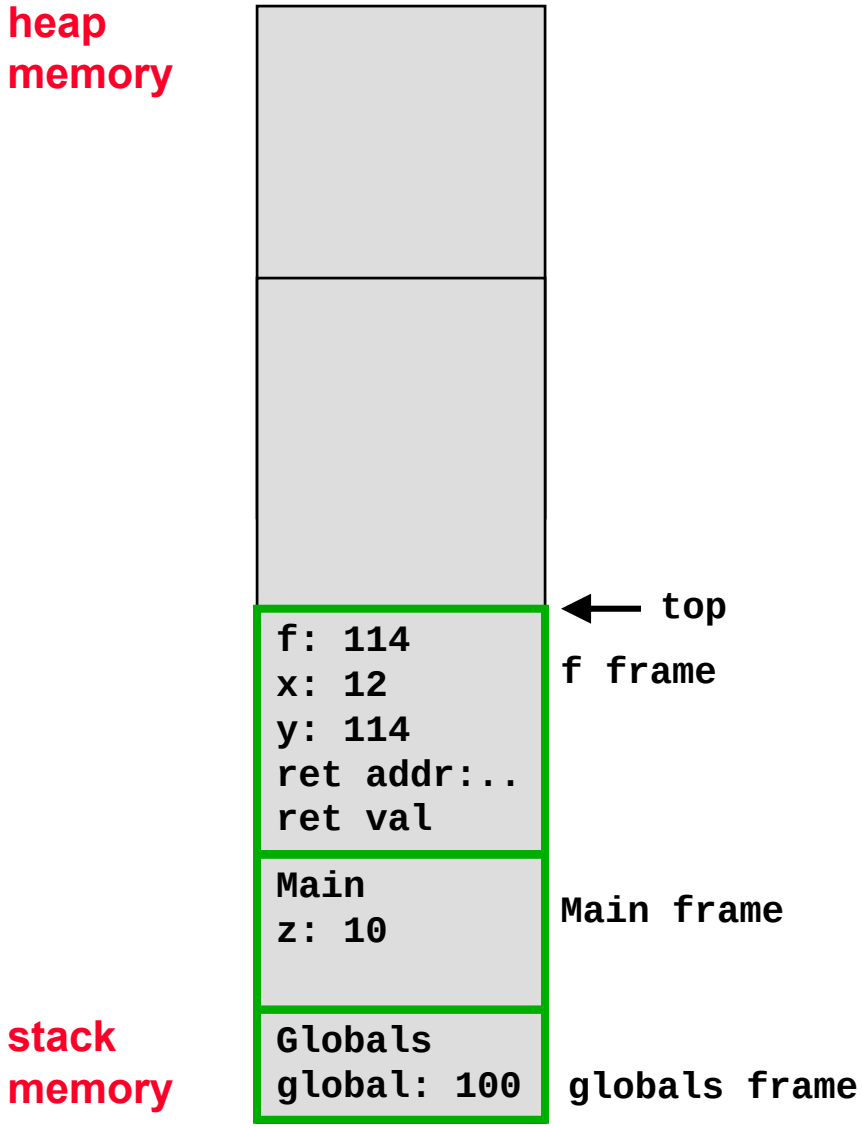


stack memory

```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

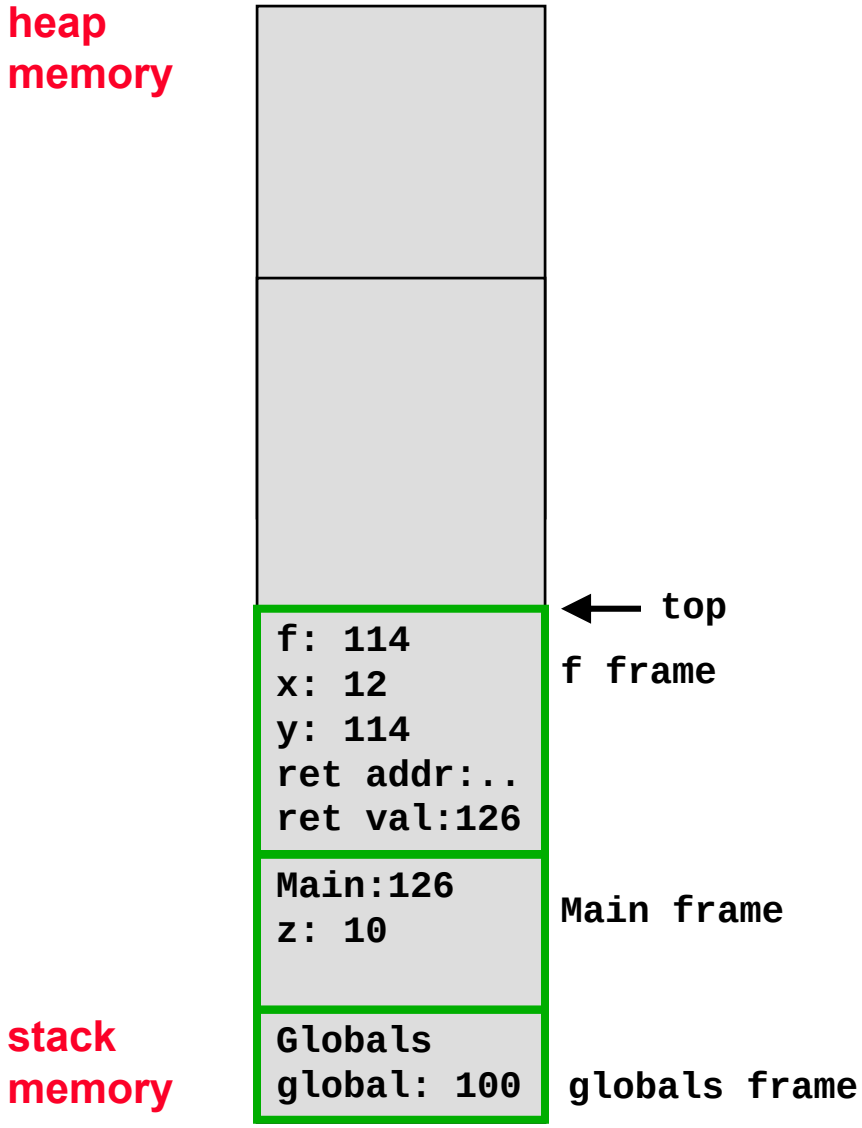
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x){
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
     print(z);
 }
```

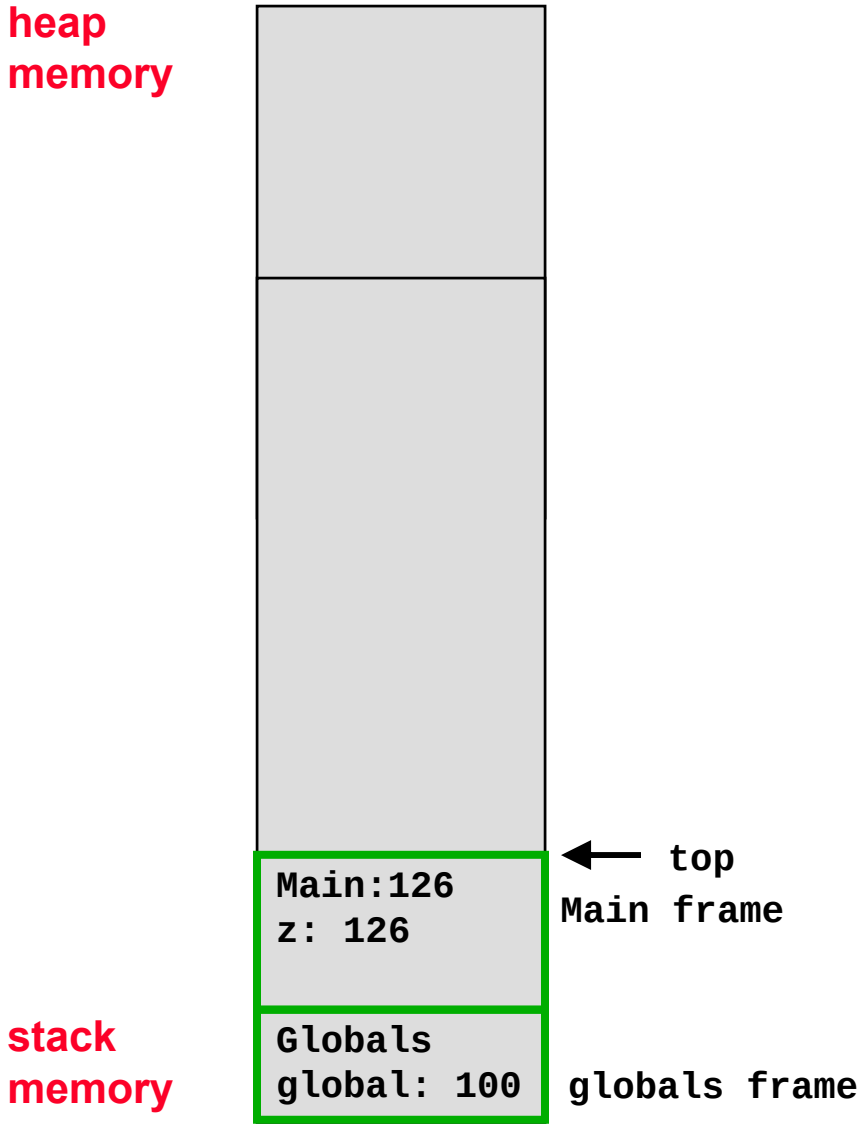
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

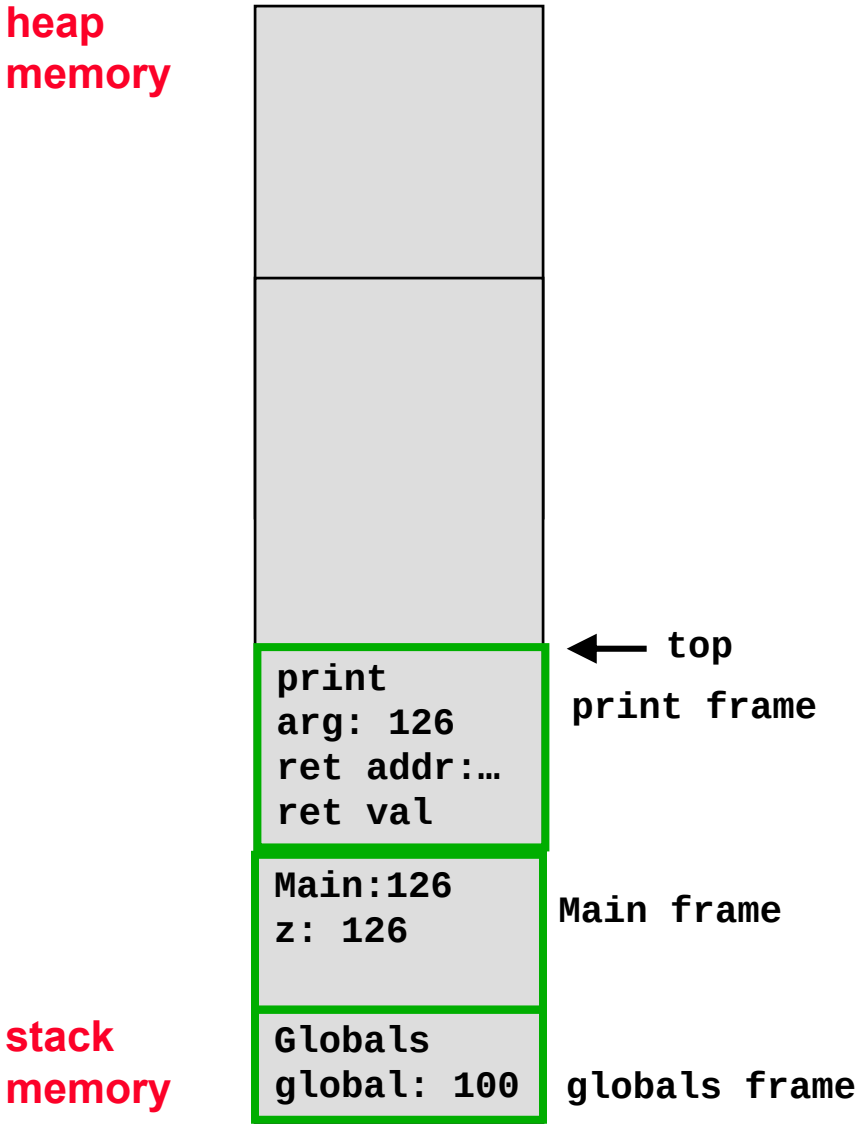
Mechanics of Procedure Stack (no particular language)




```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int
    z = 10;
    z = f(z+2);
    print(z);
 }
```

Mechanics of Procedure Stack (no particular language)



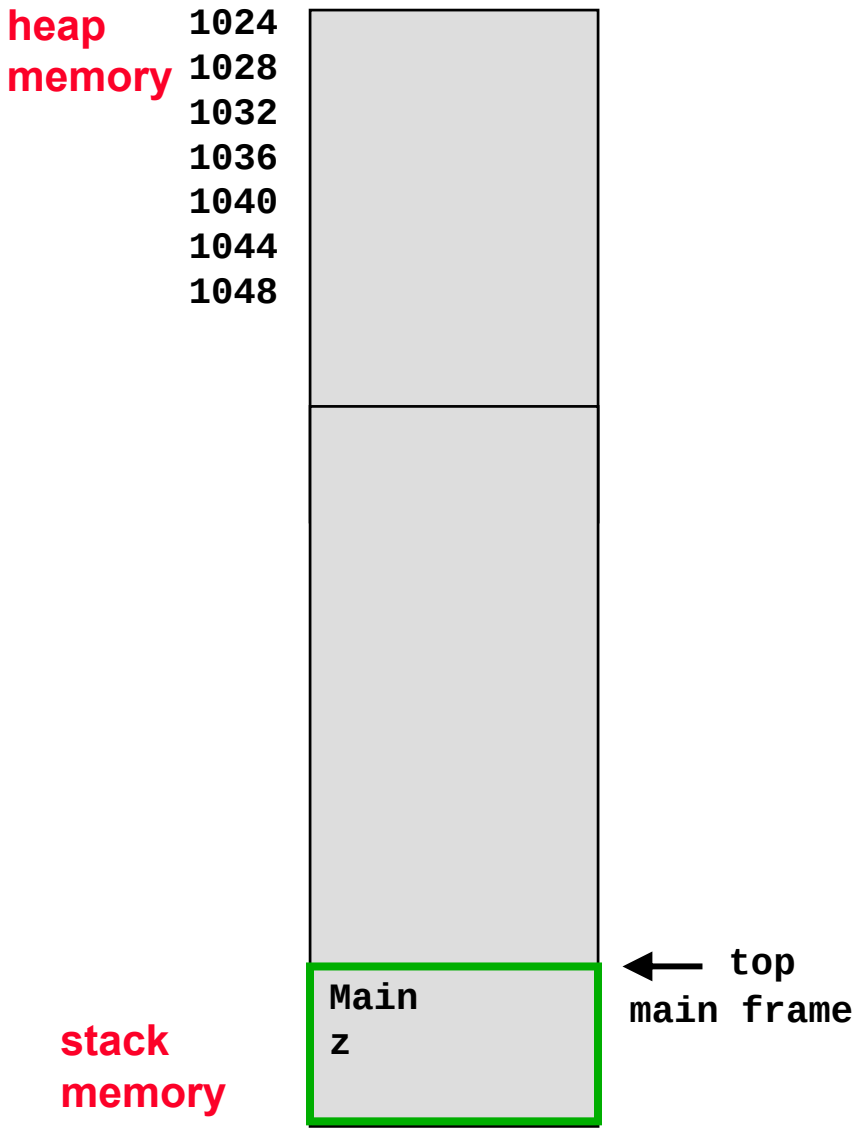

```
int global;

int g(int i) {
      int j = i + global;
      return j;
   }
  int f(int x) {
     int y = g(x + 2);
     return x + y;
  }
  main(){
    global = 100;
    int z;
    z = 10;
    z = f(z+2);
    print(z);
 }
```

Review of how Procedure Call Stack and Heap Work

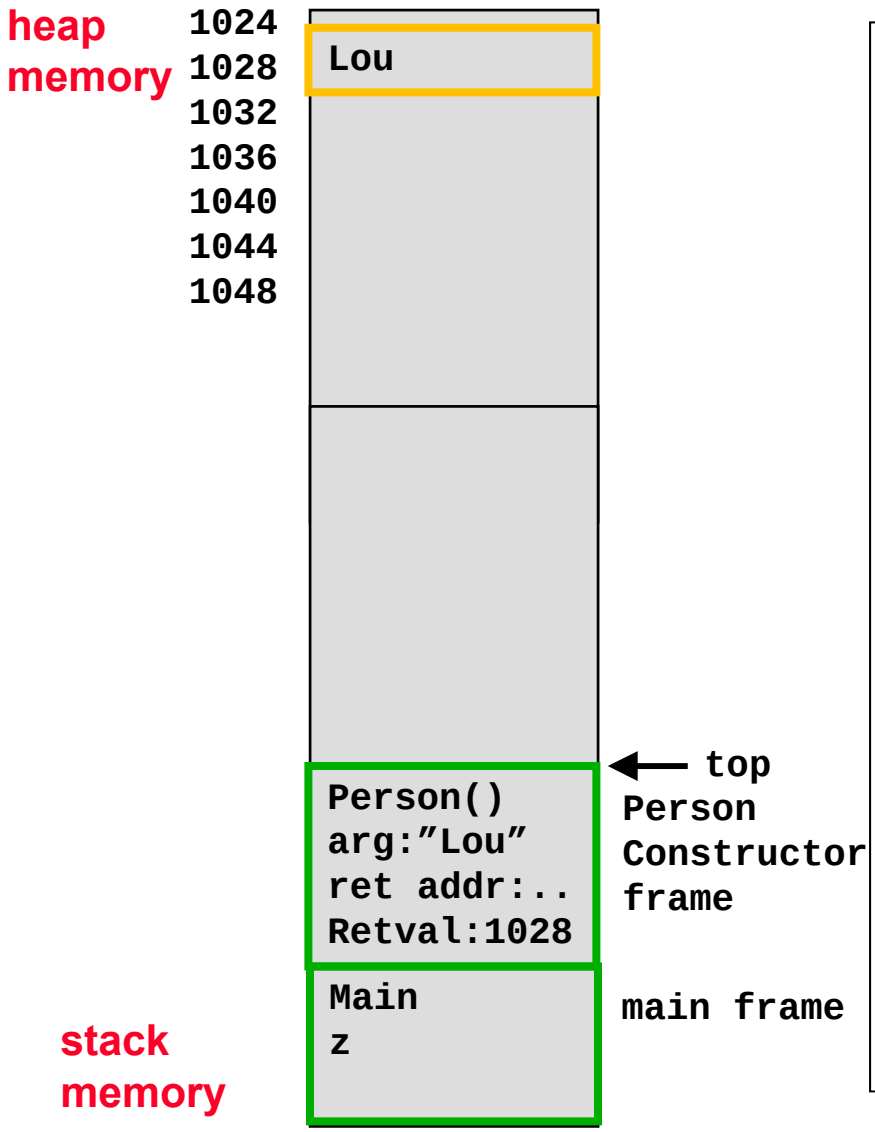
- **Accounting for objects created on the heap**

Mechanics of Procedure Stack (With heap objects)



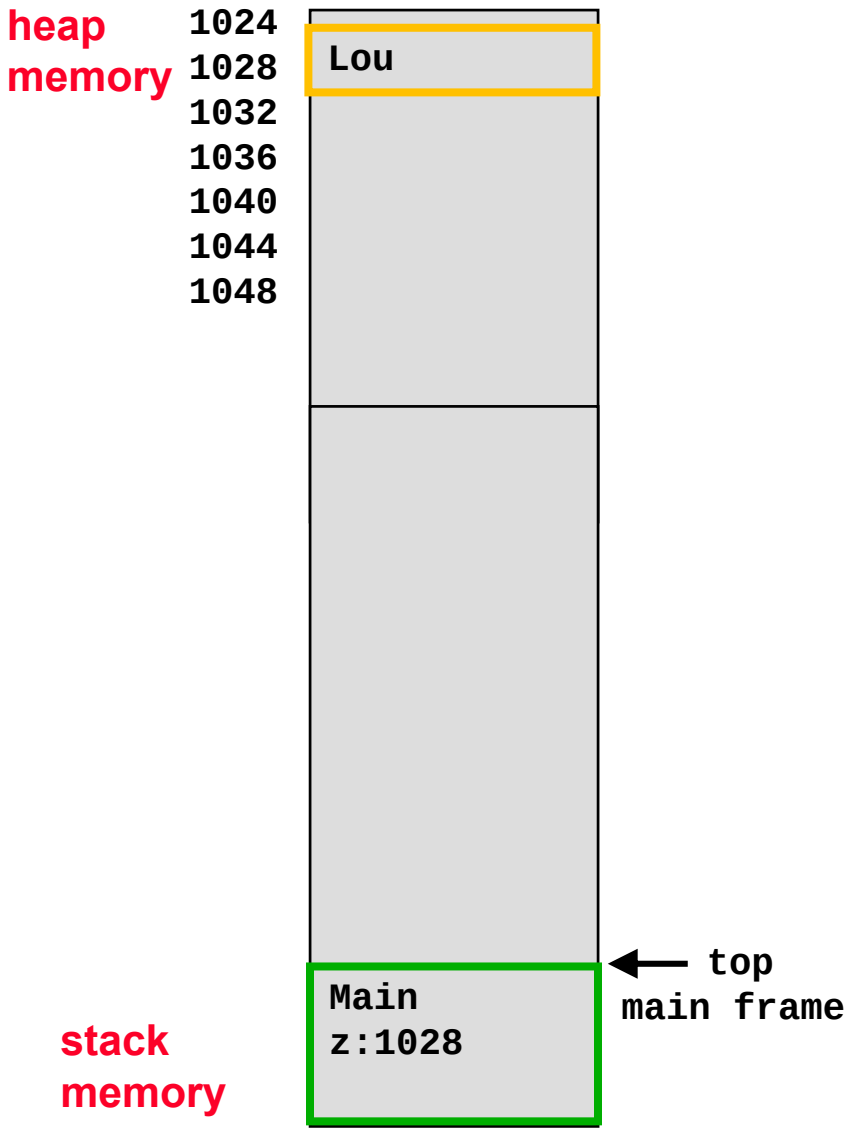

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



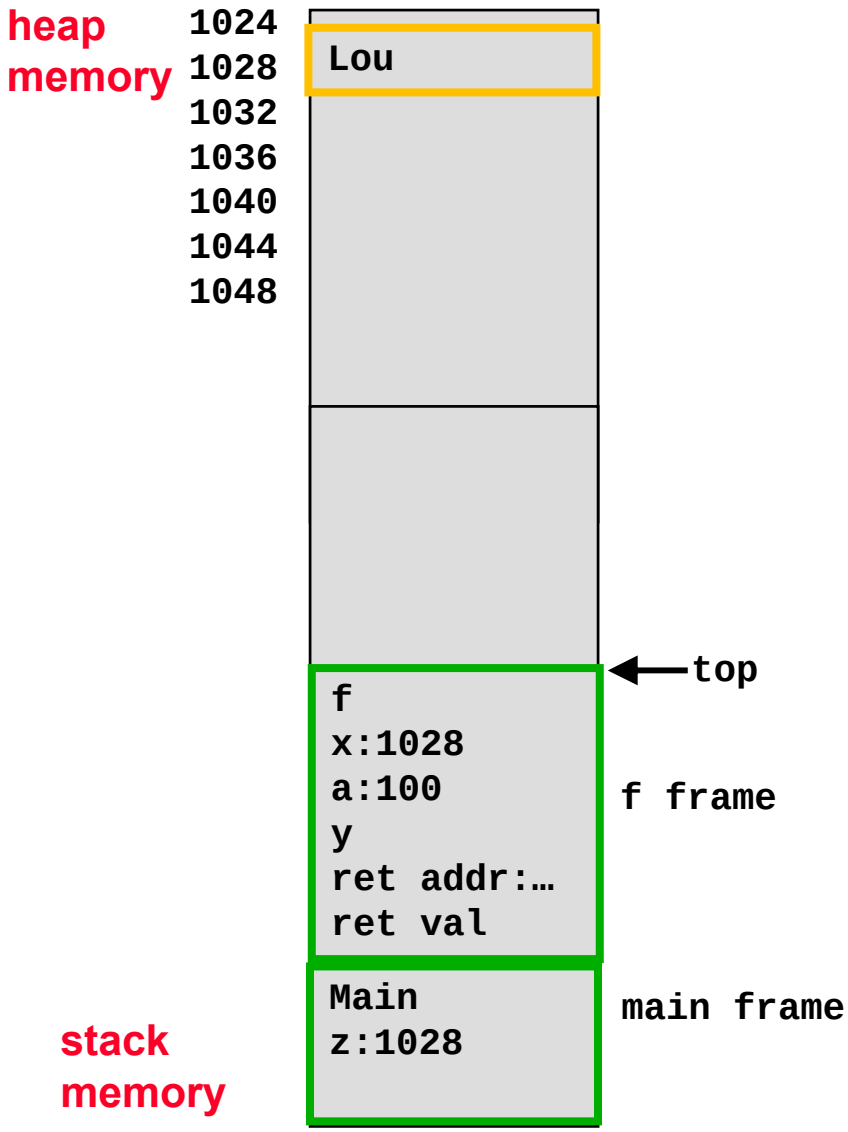

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



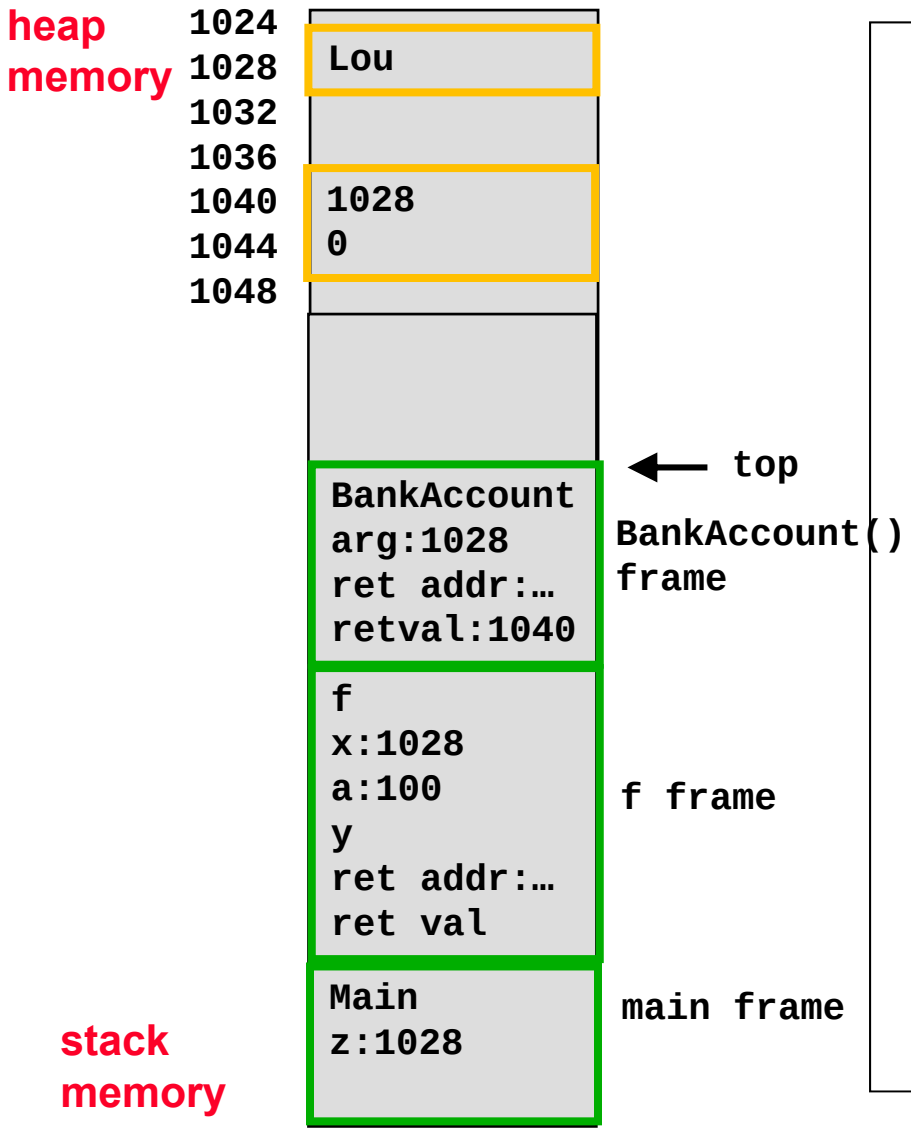

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main()
   Person z;
   z = new Person("Lou");
   f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



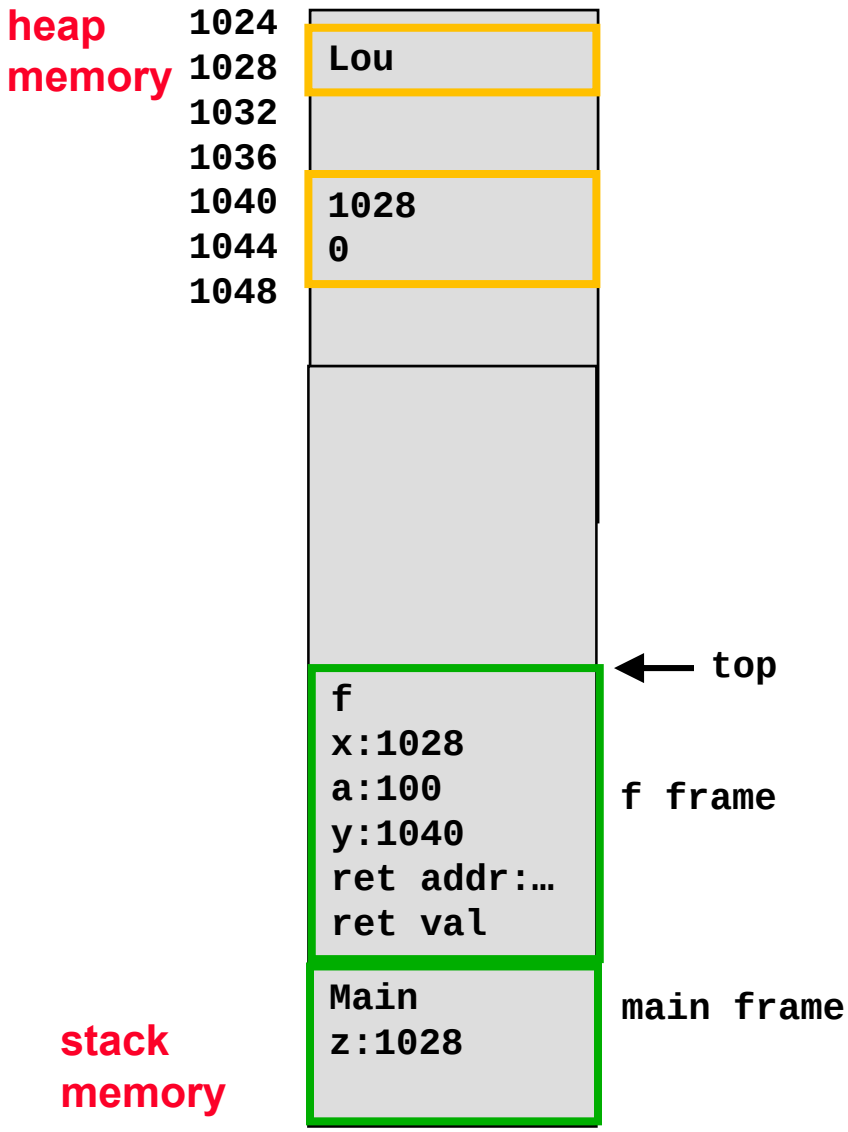

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



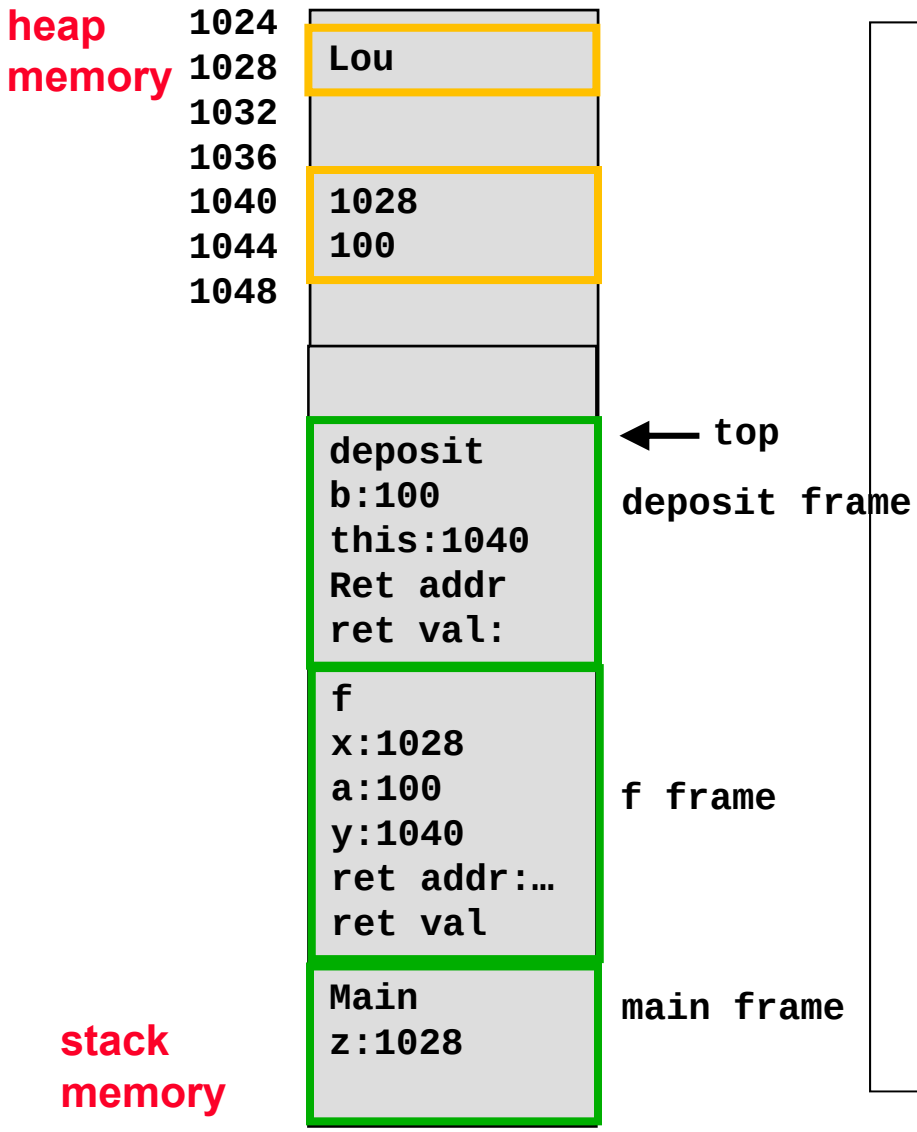

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
  print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



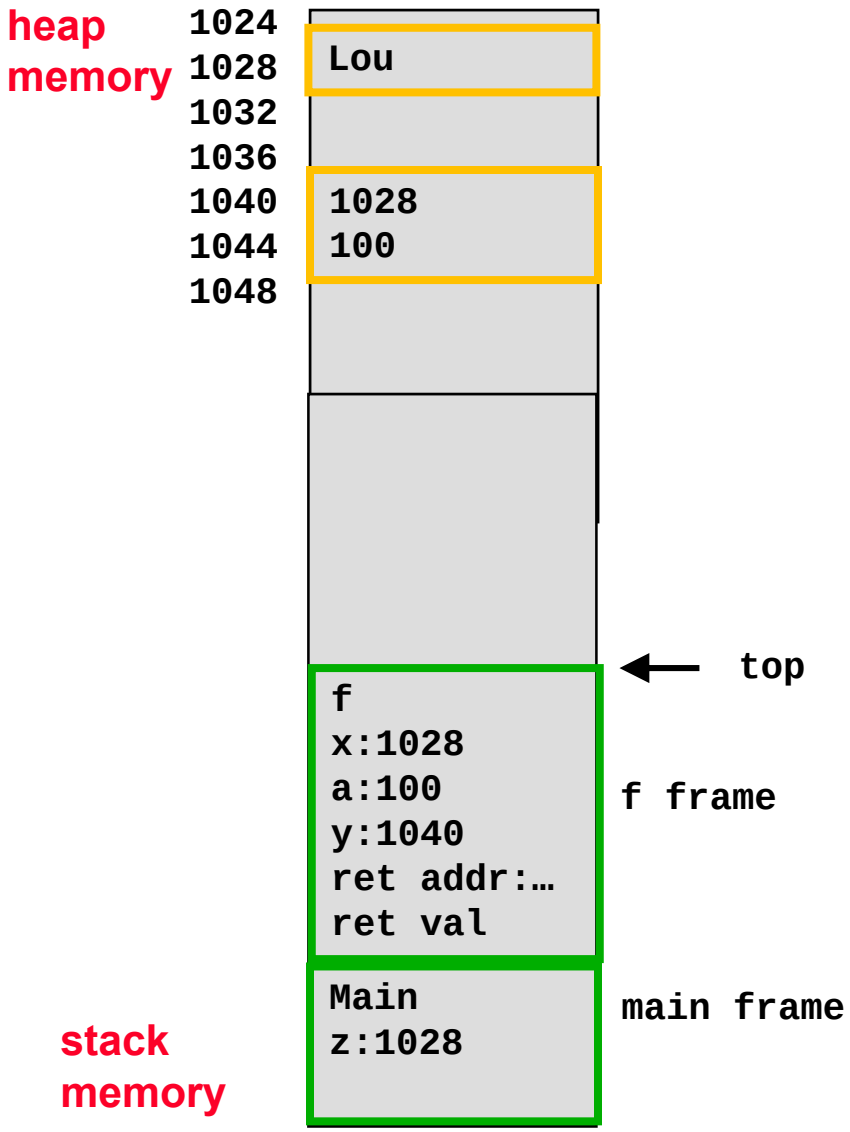

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



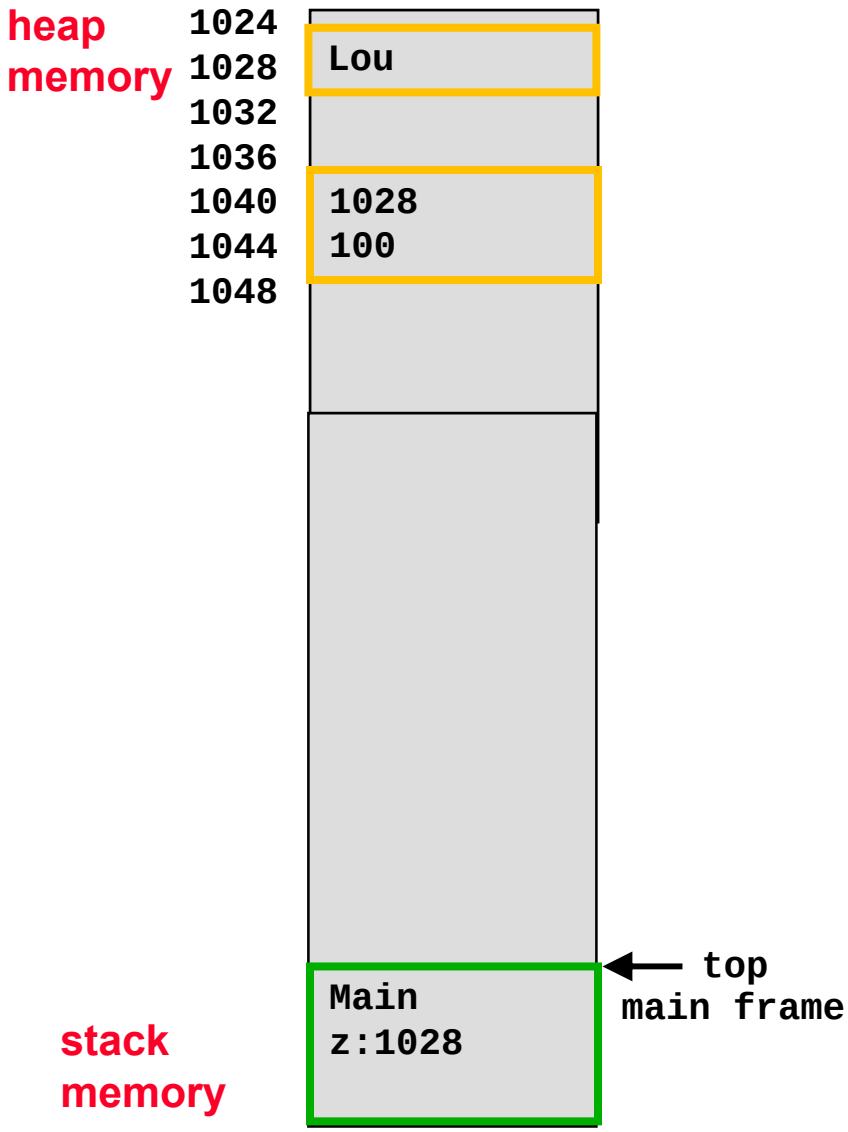

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



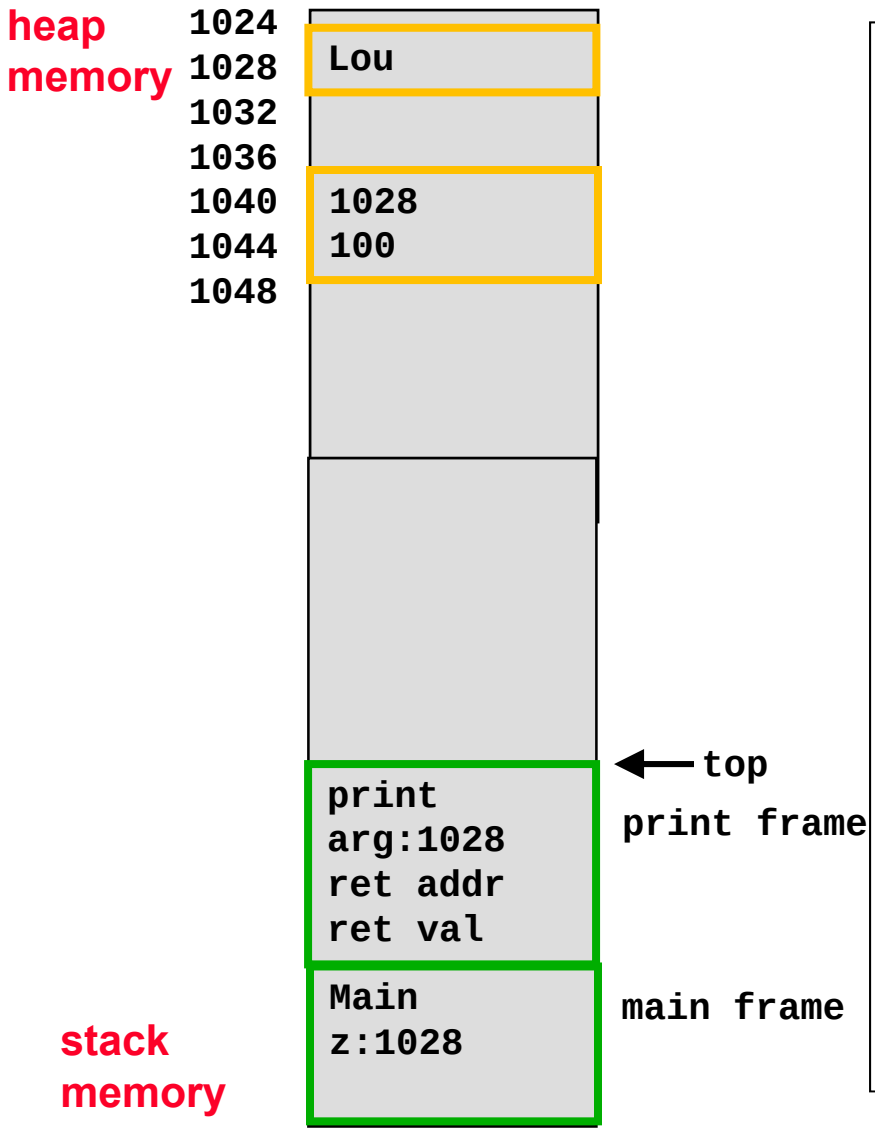

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



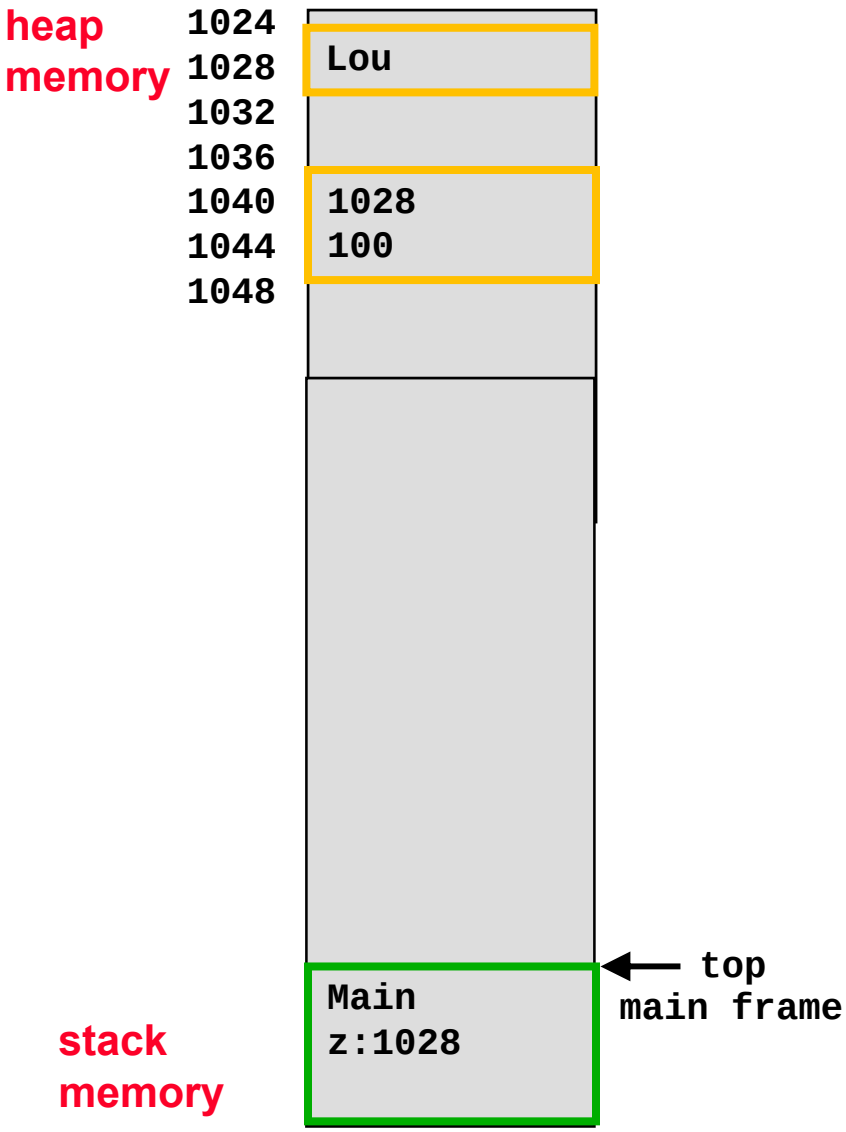

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)



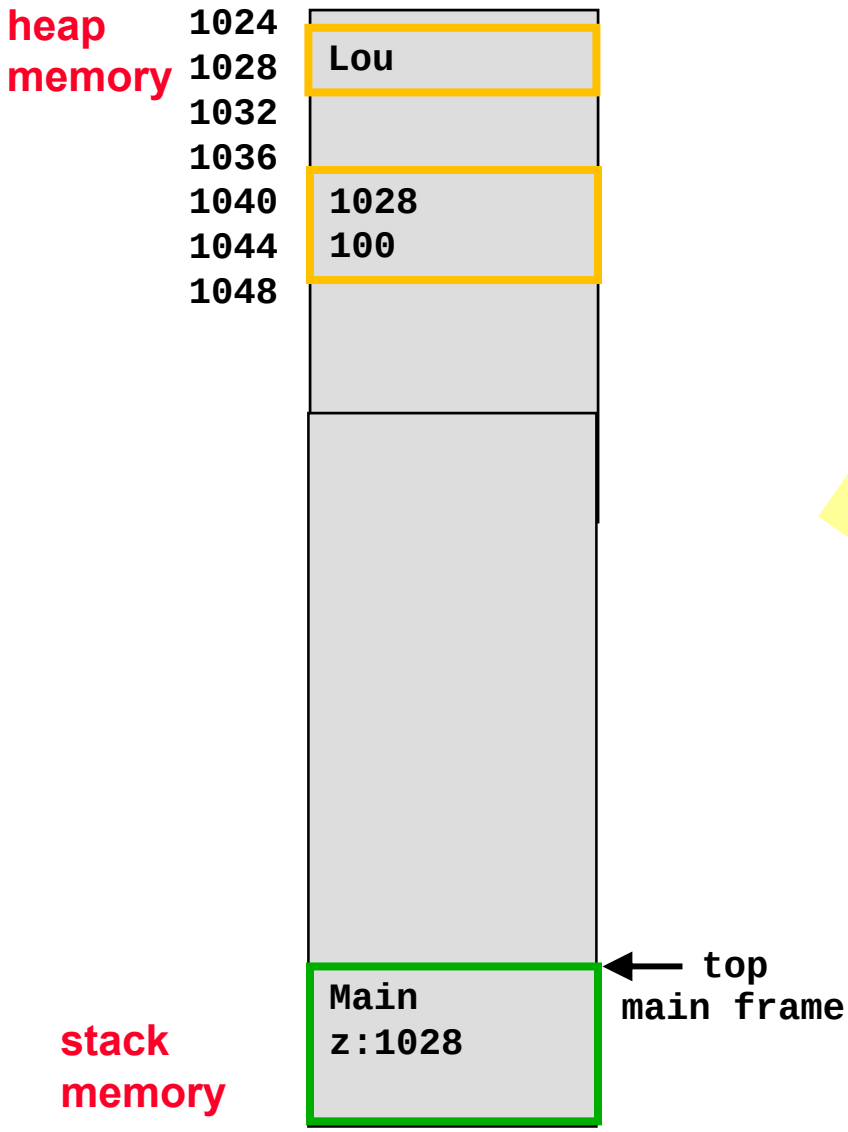

```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
   print(z);
}
```

Mechanics of Procedure Stack (With heap objects)




```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
 }
int f(Person x, float a) {
   BankAccount y;
   y = new BankAccount(x);
   y deposit(a);
   return;
}
main(){
  Person z;
  z = new Person("Lou");
  f(z,100);
  print(z);
}
```

Heap Garbage

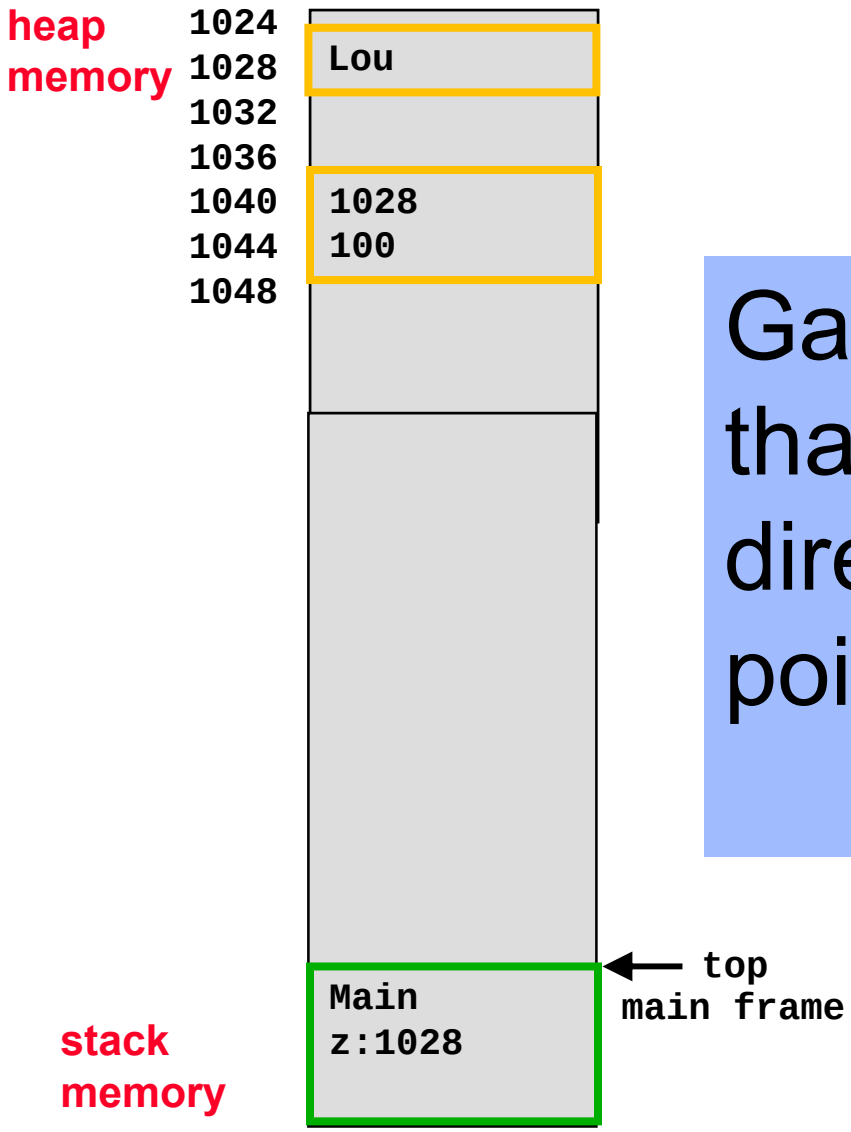



```
void Ba  count::deposit(float b)
{
    t  = b;

}
main(){
  Person z;
  z = new Perso
  f(z,100);
  print(z);
}
```

What about Garbage Collection? How does the Heap get cleaned?

Mechanics of Procedure Stack (With heap objects)

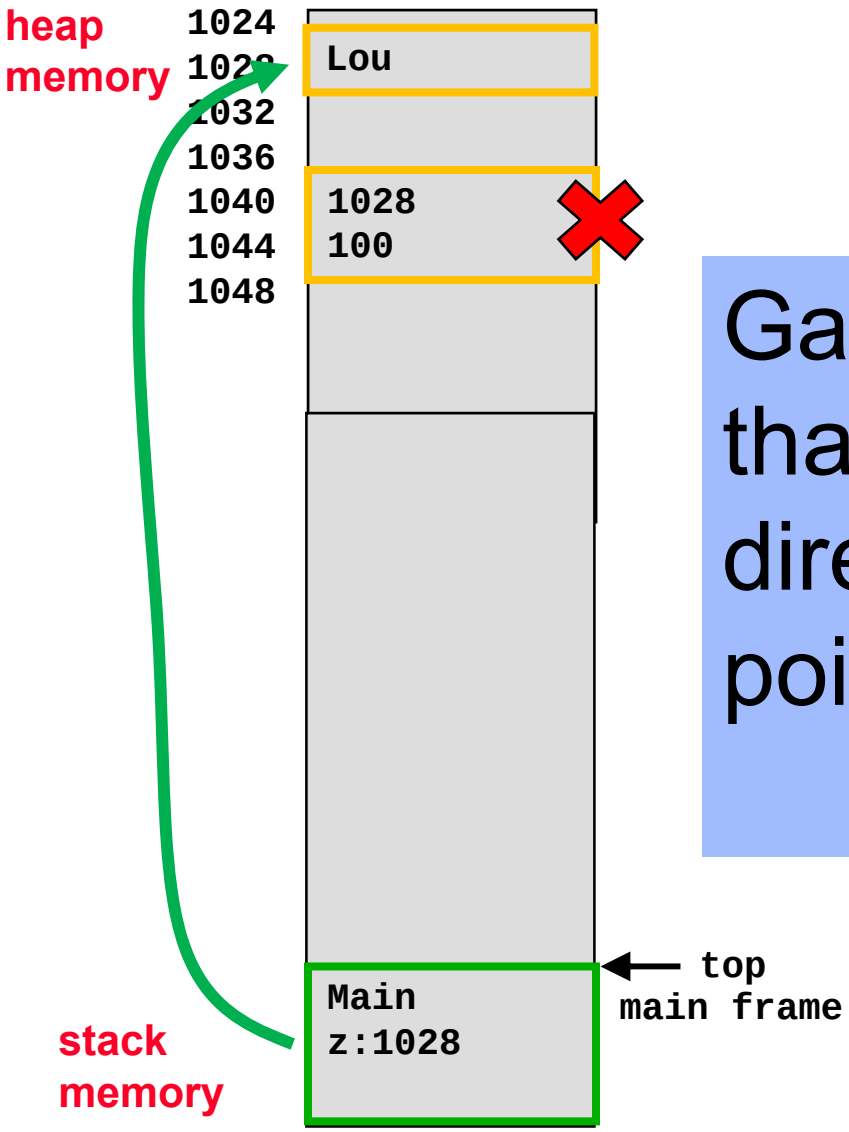



```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
```

Garbage objects are those that cannot be reached directly, or indirectly, from pointers on the stack

```
    z = new Person( "Lou" );
    f(z,100);
    print(z);
}
```

Mechanics of Procedure Stack (With heap objects)

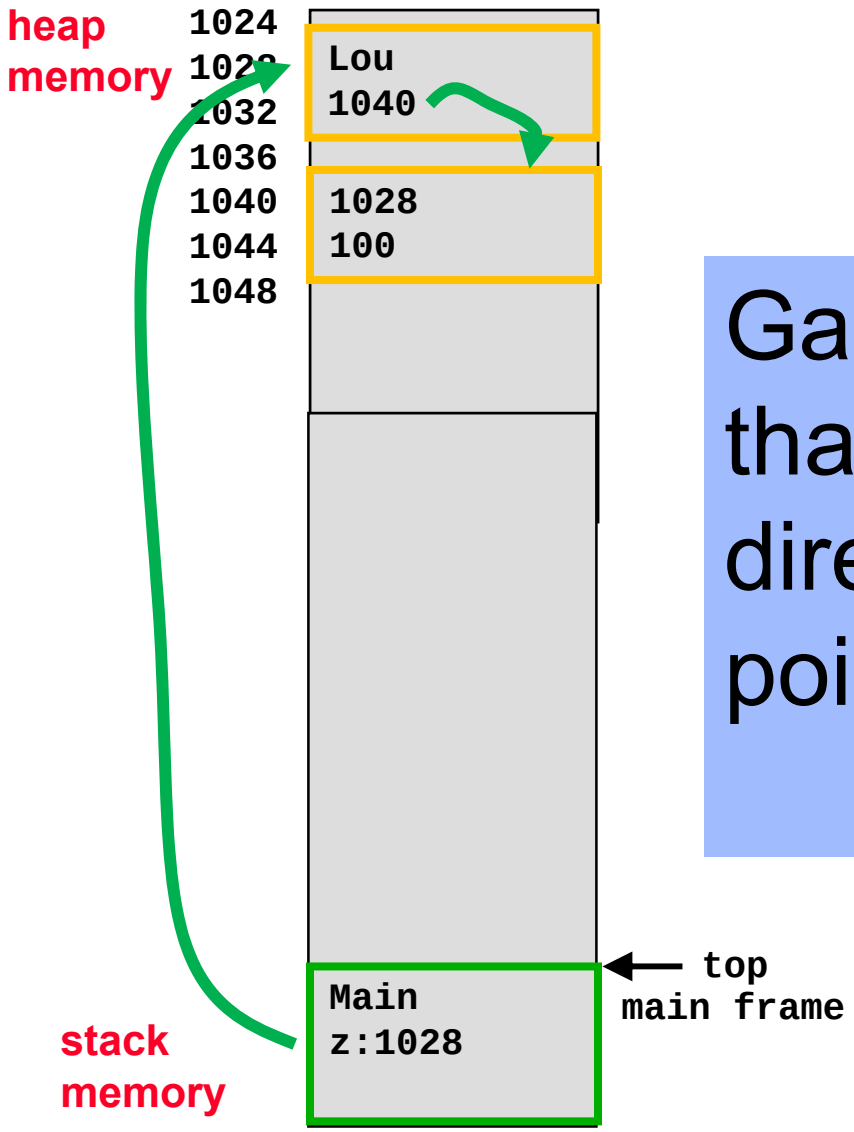



```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
```

```
    z = new Person( "Lou" );
    f(z,100);
    print(z);
}
```

Garbage objects are those that cannot be reached directly, or indirectly, from pointers on the stack

Mechanics of Procedure Stack (With heap objects)

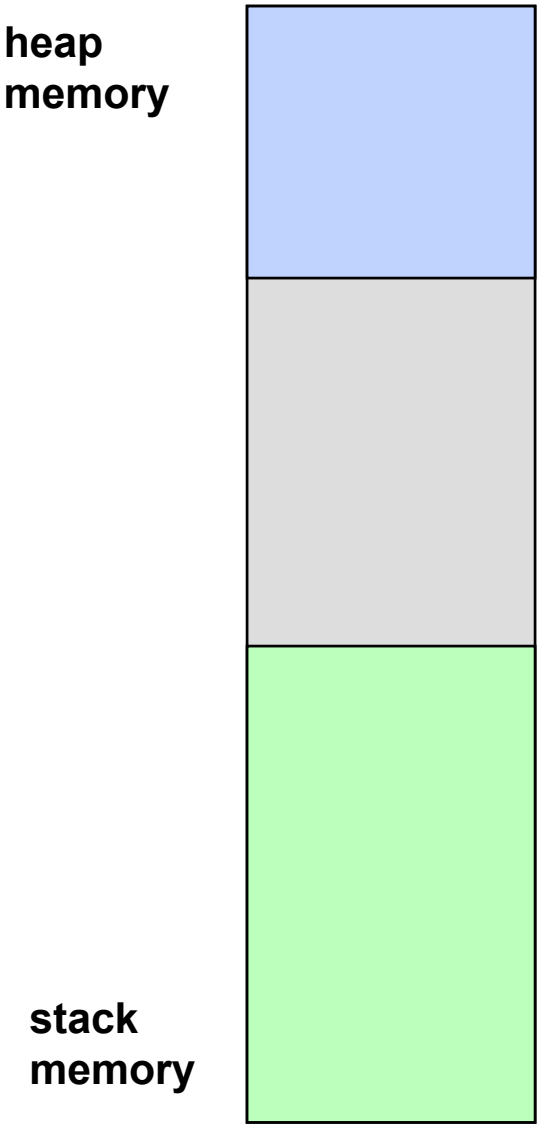



```
void BankAccount::deposit(float b)
{
    this.balance = b;
    return;
```

Garbage objects are those that cannot be reached directly, or indirectly, from pointers on the stack

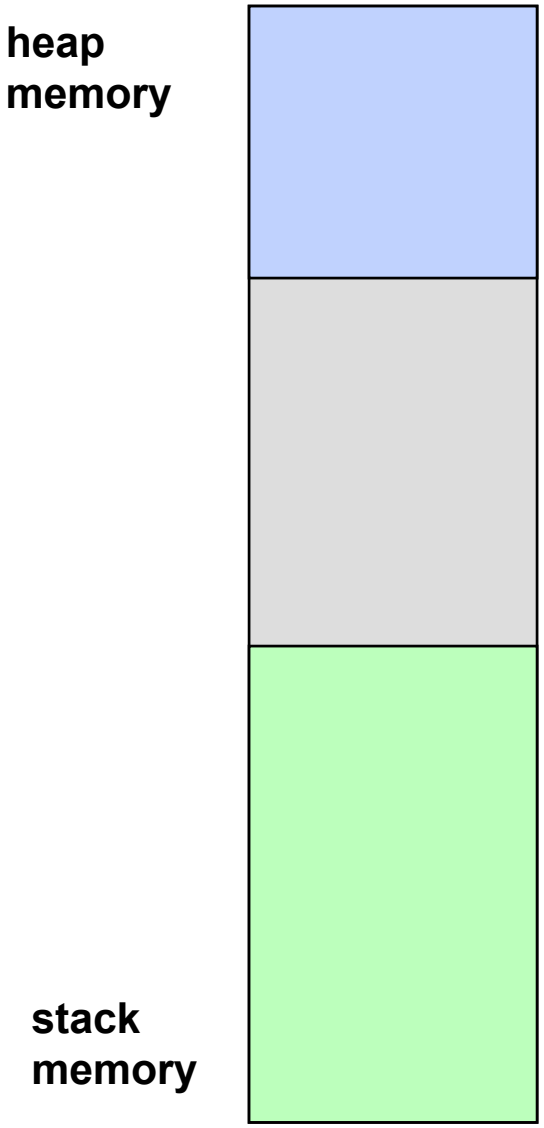
```
    z = new Person( "Lou" );
    f(z,100);
    print(z);
}
```

Stack and Heap



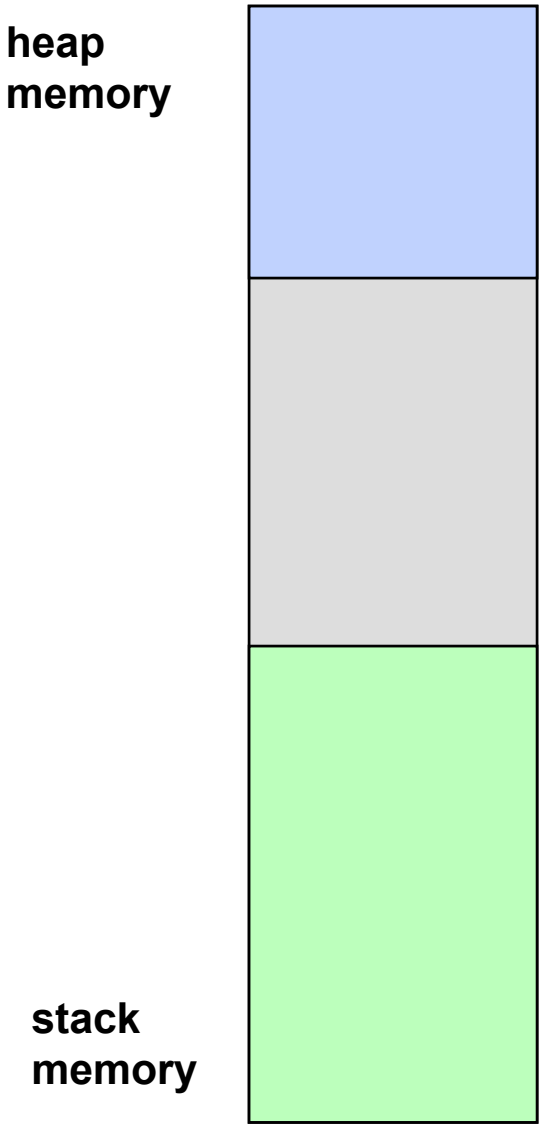
- When a program runs the OS grants it some memory to work within.
- The memory is used in two distinct regions: the stack and the heap.
- A typical way to implement the stack and the heap is to use the contiguous allocated memory and grow the stack from one end and the heap from the other.
- If the two ever collide an “out of memory” or “stack overflow” run-time error results.
- Memory can overflow because of a bug, or because a program simply needs more memory than is available.
- Some programing techniques, like recursion, can easily get out of hand and use up a lot of stack space to represent sub-problems.

Review of how Procedure Call Stack and Heap Work



- The heap is used for items, most often created with `new`, that have a lifetime independent of the function call that created them.
- Items on the heap remain there until they are explicitly deleted.
- In some languages, like C++, C, items are removed by the programmer explicitly.
- In others, like Java, items are removed by a garbage collector when they are no longer accessible.
- Garbage collectors are often viewed as slower but safer.
- There is no specific order to where things are placed in the heap. (In fact a "heap manager" might run in the background and continuously re-arrange and defragment the heap.)

Review of how Procedure Call Stack and Heap Work



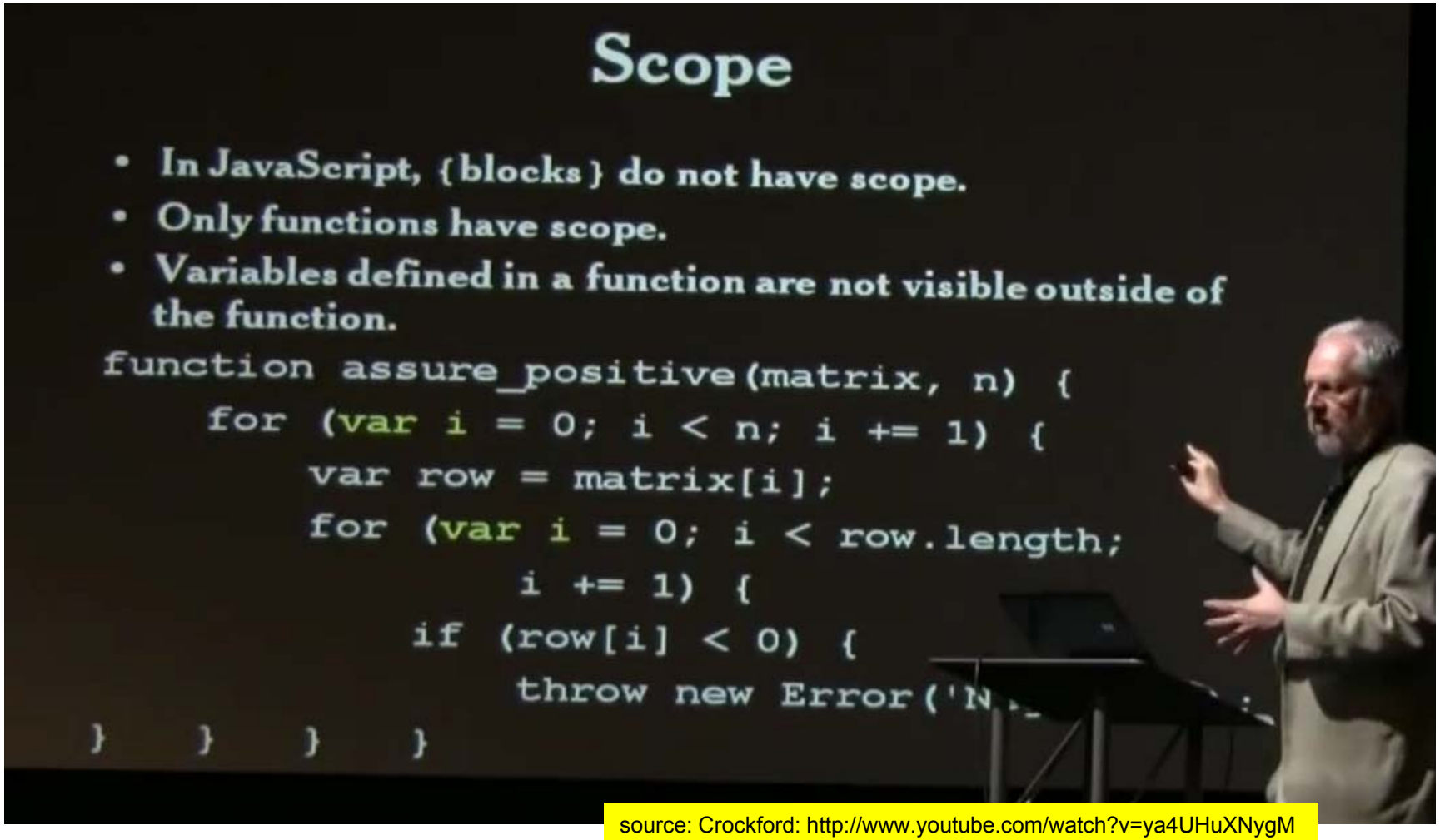
- The stack is used to implement procedure, or function, calls.
- When a procedure is called a "stack frame" representing that procedure is placed on top of the stack; when the procedure returns its stack frame is popped off the stack.
- The stack grows and shrinks in a "first in – last out" sequence.
- So the lifetime of stack frames matches exactly the calls and returns of the procedures they represent. (By contrast, the lifetime of items on the heap is independent of the function calls that placed them there.)
- Since procedures can be called at any time and from anywhere, they cannot make assumptions about what is "below them" on the stack.

Additional Notes

Variable Hoisting

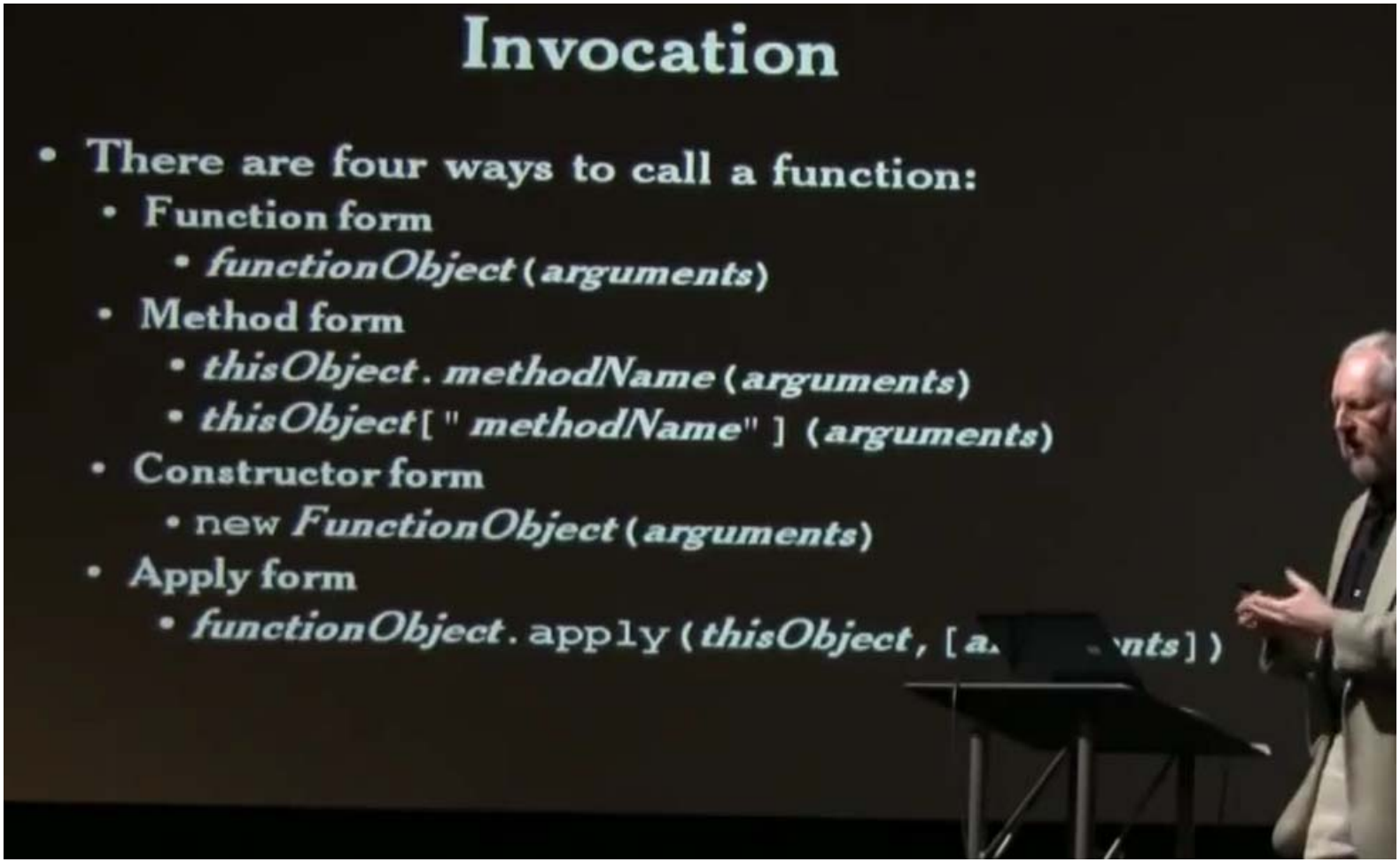
Variables are hoisted to top of functions.

Variables are hoisted out of blocks to tops of enclosing function.



source: Crockford: http://www.youtube.com/watch?v=ya4UHuXNygM

Function Invocation



source: Crockford: http://www.youtube.com/watch?v=ya4UHuXNygM