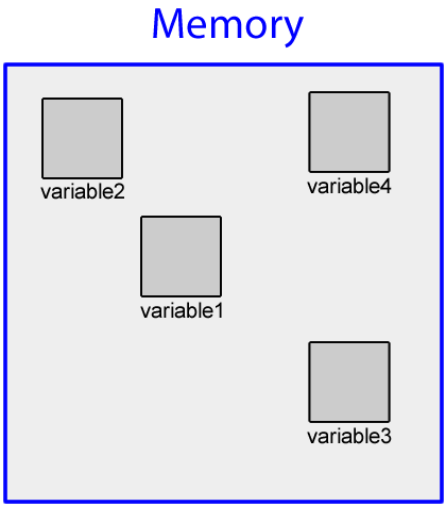
CSC 325 Adv Data Structures

Lecture 3

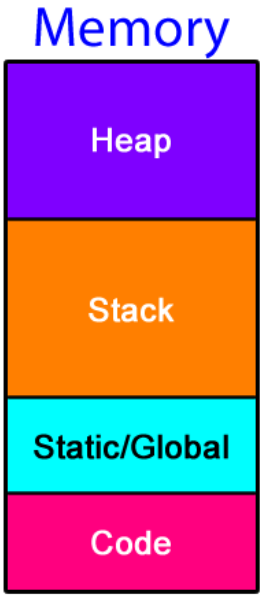
Runtime Stack, Heap and Garbage Collector

Memory Layout

Up until now, you might think that memory works like this:

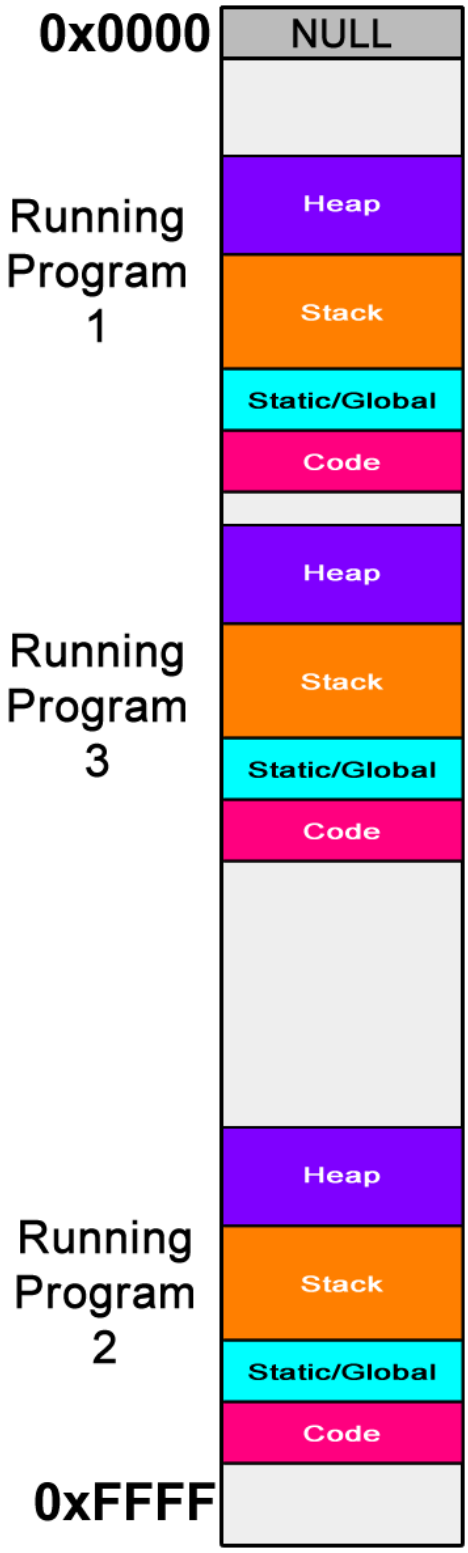


Actually, it is a bit more organized than this. For starters, the code itself must reside in memory if it is to be executed. But what about data? The CPU constantly talks to memory and is constantly asking it to store and grab values at different locations. However, memory is smarter about storing these values than what the picture above shows. Memory actually looks more like this:

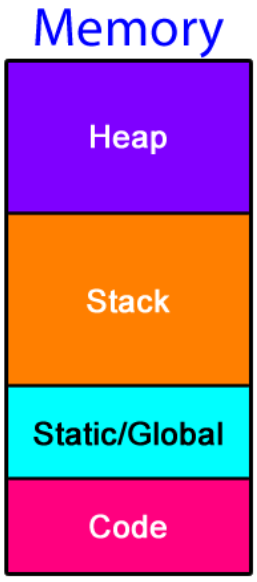


This is what memory looks like for a single running program[1]. Imagine this same set of partitions for each running program on your computer:

[1] Google search: https://tinyurl.com/yckj226s



Now some of the discussion of what goes on in memory is outside of the scope of this class. You may learn some of these concepts in your architecture, OS, or even programming languages course. But for 325, I want to focus mainly on the use of data structures and algorithms here. Let's go back and focus on a single program running in memory:



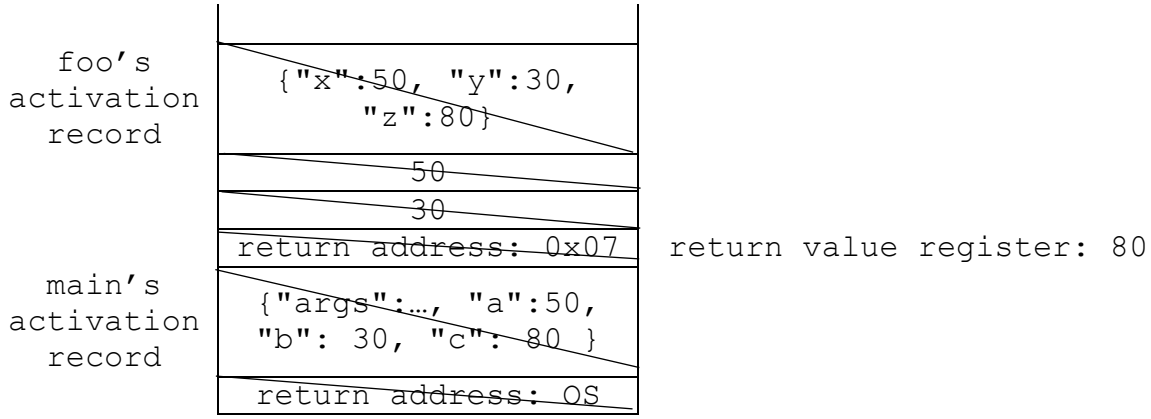
We have learned about stacks and heaps in 220. But here we have the operating system (or the runtime environment, e.g. JRE) using a "runtime stack" and a heap (also known as "free store") to control what we call "memory management" (or "storage allocation"). When you allocate space for a variable, the memory created goes into either the runtime stack or the heap. We can also see a place for the code to reside. Global and static variables reside in a different spot in memory than ordinary variables do. This is because they are ALWAYS accessible throughout the lifetime of the application. Their visibility (public, private, etc.) controls where they can be accessed, but they are still to reside in memory throughout the running of the program. I could have sworn that constant values also reside here, but I couldn't find evidence of that online, so perhaps not.
As stated, when a variable is created, it is stored in either the stack or the heap. Usually (but not always) the stack stores local variables whereas the heap stores values who's memory needs to be kept after a function call.
Let's examine the runtime stack more.

Runtime Stack

It turns out that each time a method is called, a good bit happens. All of this happens behind the scenes so we never really worry about it, but it is still important to understand. We use the runtime stack to keep track of our local variables during program execution. We don't have to explicitly communicate with this stack, as all this happens for us by the operating system or the Java virtual machine in the case of Java.

Here is a full example of the runtime stack in action for a simple program in Java:

```
0x00 public class Main {
0x01     private static int foo(int x, int y) {
0x02         int z = x + y;
0x03         return z;
         }
0x04     public static void main(String[] args) {
0x05         int a = 50;
0x06         int b = 30;
0x07         int c = foo(a, b);
0x08         System.out.println(c);
         }
     }
```

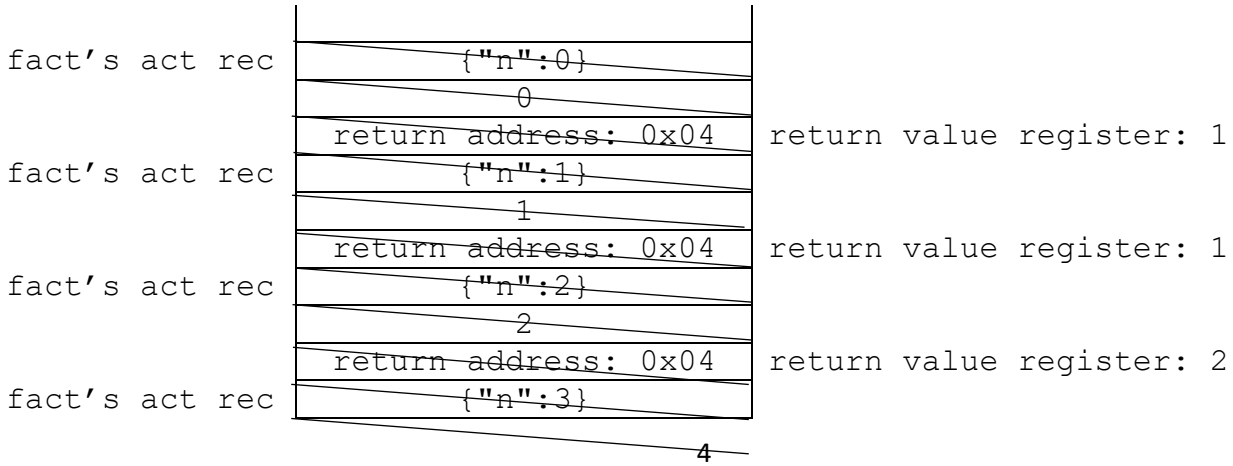


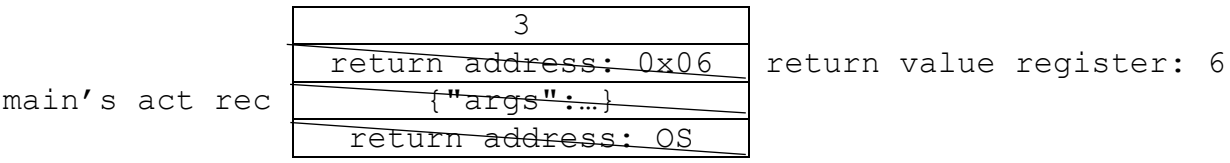
Before the program is executed, the operating system (Windows, Mac, Linux, etc) is busy doing stuff. It would like to return to doing stuff when our program finishes. So it pushes the memory address of the code it was working on, onto the stack (labeled here as “return address: OS”). This is so it doesn't forget what it was doing. Now we need to push on an "activation record" of all local variables and parameters for our main method. This keeps track of the values of each. Think back to our dictionary sequence from Python as this is basically what is being stored here. At line 0x07 we call the `foo` method. So that we can continue where we left off in `main`, we need to push that line’s address onto our stack before we call the method (i.e. “return address: 0x07”). Now we push on the arguments (aka actual parameters) to the method (this is shown as “30” and “50” on the stack).

Talk about the reverse order that the arguments are pushed onto the stack. Refer to c++ code in file “stack_args_backwards.cpp”. talk about how this is language dependent and how “c calling convention” dictates that we put arguments onto the stack (in reverse order) and pop them off to get parameter values.

When we move control over to the method, we pop off the arguments and save them as values to our parameters (i.e. `x` and `y`). We then push a new activation record for our `foo` method, one that contains those parameters `x` and `y` that we just initialized. During the method, we declare and initialize a variable called `z`. This variable gets added to the activation record (shown at the top of the stack). When we are finished with our `foo` method, we pop its activation record off the stack (shown by putting a diagonal line through it), set a special value on the CPU to be the value we want to return (shown as “return value register: 80”) and then finally we return to the line address 0x07 as that is what is on the stack (i.e. “return address: 0x07”). At this point, the top four items shown above have been popped off the stack and the top of the stack shows the activation record for `main`. This allows us to continue working in the `main` method. When we are done, we pop off `main`'s activation record and pop off the return address (i.e. “return address: OS”) we stored earlier that contains what the operating system was working on. This allows the operating system to go back to what it was doing before the program executed. Let’s look at an example of recursion with our runtime stack:

```
0x01 public class Main {
0x02     private static int fact(int n) {
0x03         if (n == 0) { return 1; }
0x04         else        { return n * fact(n - 1); }
         }
0x05     public static void main(String[] args) {
0x06         System.out.println(fact(3));
         }
     }
```





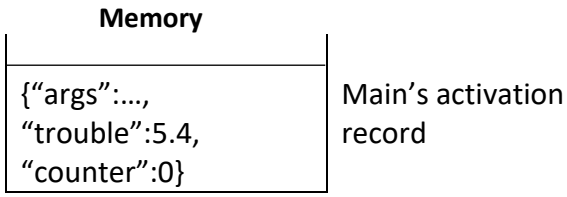
Note that "act rec" is shorthand for "activation record". You should be able to see now how expensive recursion can be. The stack resides in main memory and reading/writing to main memory can be expensive, that is it can take more time than we would like to spend. Note that there are optimizations that use the registers in the CPU[2], but recursion can still be expensive at times.

One way to speed up recursion is to use tail recursion. Recall that with head recursion, when we return from a recursive call, there is still more work to do. Therefore it is imperative that we remember the state of every variable from every recursive call. The runtime stack helps with this. However with tail recursion, we have absolutely no work to do when we return from a recursive call. This means that we actually don't need the activation record of the previous recursive calls! A smart compiler can recognize tail recursion and transform the code into using a loop instead. This makes recursion no more expensive than iteration! If possible, we should use either tail recursion or iteration to solve a problem and only rely on head recursion if we need to.

Back to Memory Layout

So, back to the question of what gets stored on the runtime stack and what goes on the heap. We already know that local variables go on the stack. For example, after running the following code, here is what the stack would look like:

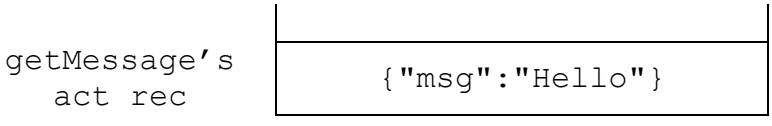
```
public class Program {
  public static void main(String[] args) {
    double trouble = 5.4;
    int counter = 0;
  }
}
```



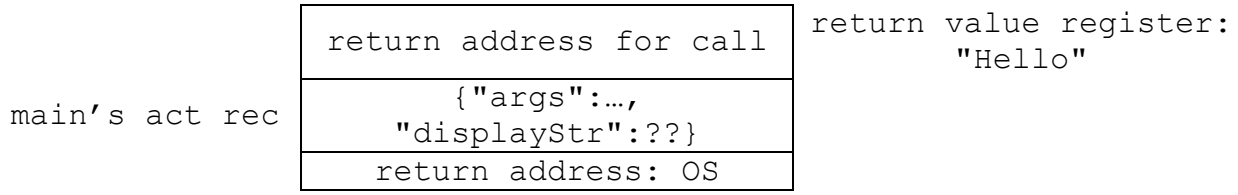
However, we might have an issue with certain variables. Imagine the following code:

```
public class Program {
  private static getMessage() {
    String msg = "Hello";
    return msg;
  }
  public static void main(String[] args) {
    String displayStr = getMessage();
    System.out.println(displayStr);
  }
}
```

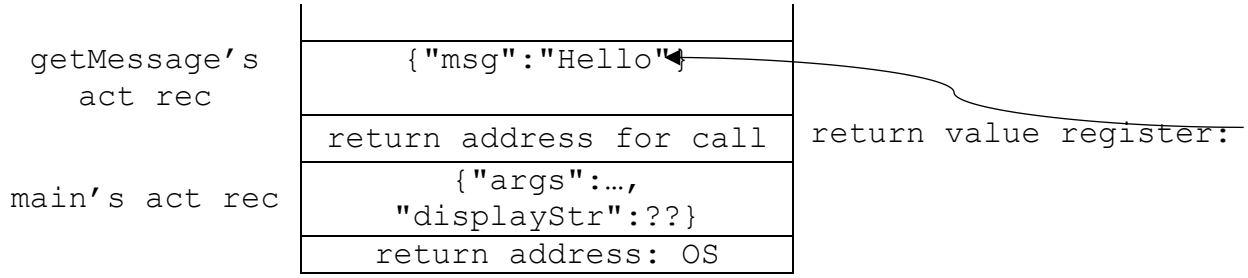
If we store all variable values on the stack, we would have the following:



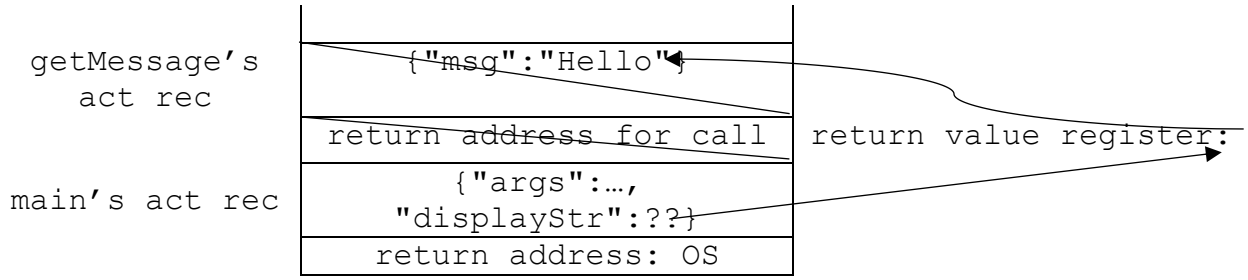
[2] In some languages, the first 4 parameter values are held in CPU registers and memory is only used starting with the 5th parameter



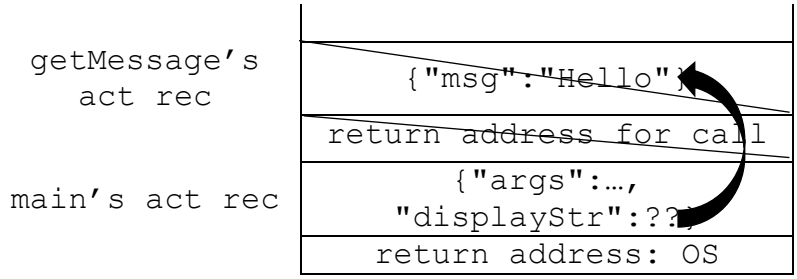
This so far doesn't seem to cause any concern. However, let's think about this for a second. The return value register has a fixed size. How could it possible hold on to an entire string, especially when strings can come in different sizes (i.e. can have different number of characters). Some could be quite large! This actually doesn't make sense. The reality is that, since the register has a fixed size, holding reference values like this is better done by just holding onto the reference (i.e. the address where this value is stored in memory). So let's alter our stack some:



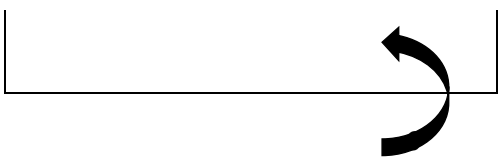
Ah, that's better. Nothing bad can happen now. Let's go ahead and return from the getMessage function and give the value in the return value register to the variable "displayStr" in main:

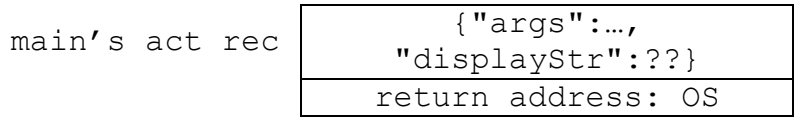


So now the displayStr variable holds the memory address that the return value register had. Let's clean up the stack above to see this better:

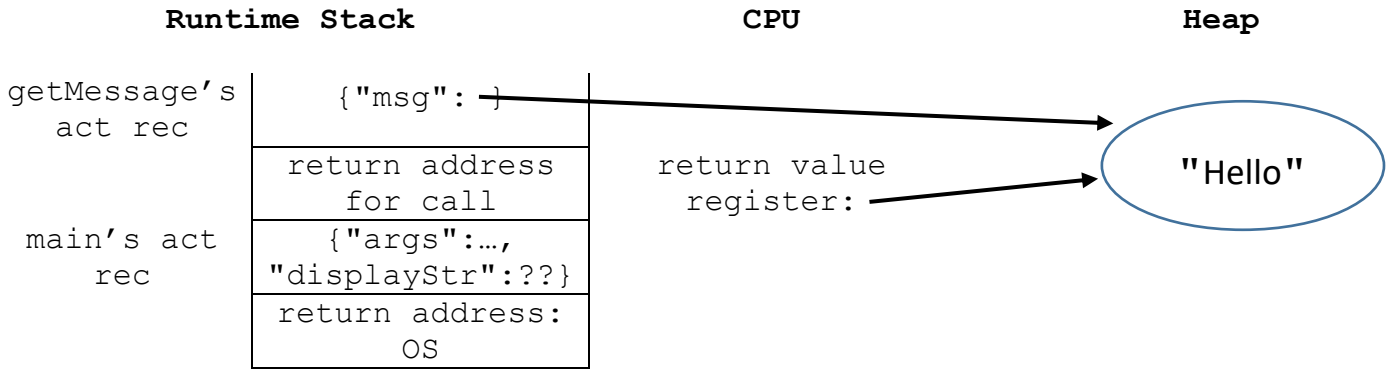


There we go. Wait a minute!!! Didn't we just pop getMessage's activation record from the stack? That means the image should really look like this:

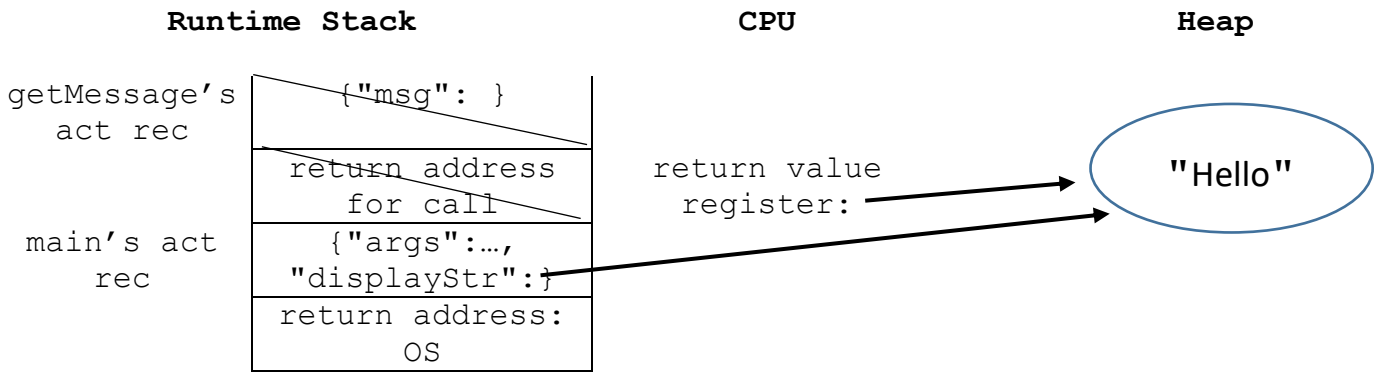




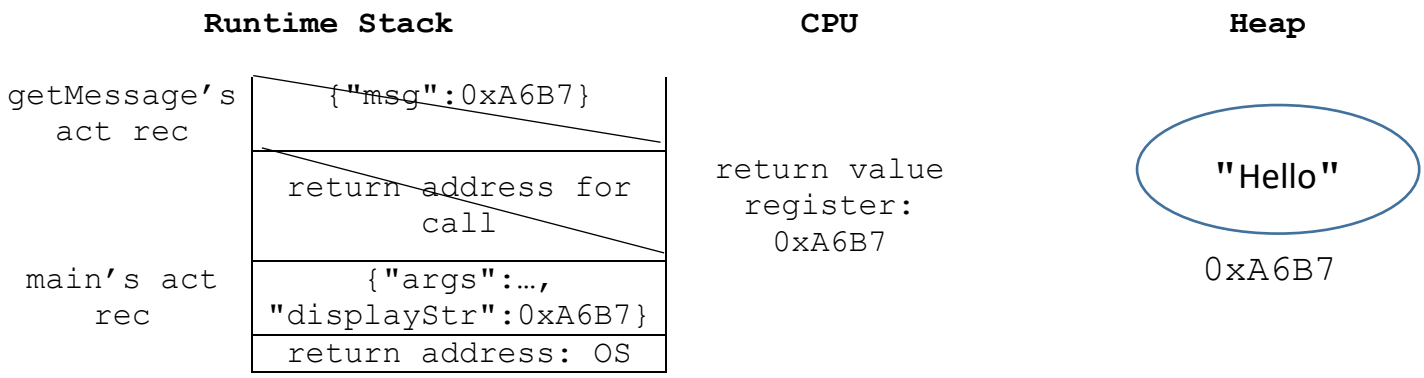
Oh crap! The memory address we copied from the return value register now points to unallocated memory. This is bad! This shows that using a stack alone for every variable value is a bad idea. It just doesn't work. So any value that must be retained after the local variable is destroyed, must be stored in a different area. In comes the heap to save the day! This is actually how things look:



Now I think we are ok. Let's pop getMessage's activation record and return from the function back to main:



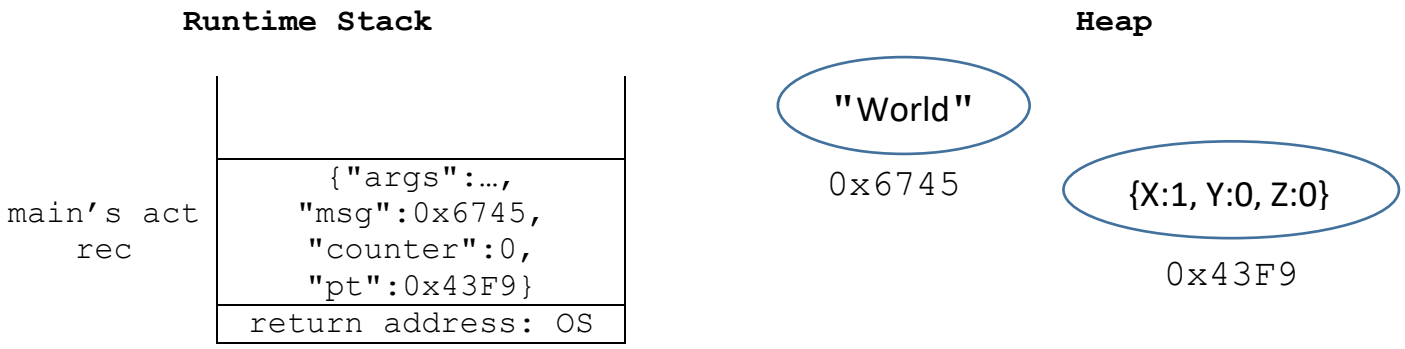
This seems to work. Now we are pointing to the same memory the return value register points, which is NOT on the stack, so popping the activation record from the stack has no negative effects on returning the value. Now, to be clear, we don't actually have pretty arrows in memory. So really, we are actually storing hex value addresses:



Just a quick recap. Recall that memory (i.e. runtime stack and heap) communicate with the CPU via the "data bus". It's important to recall past information as computer science concepts definitely build off of each other. So don't forget the things we have taught you in past classes!

Let's look at another example:

```
public class Program {
  public static void main(String[] args) {
    String msg = "World";
    int counter = 0;
    Vector3 pt = new Vector3(1, 0, 0);
  }
}
```



It turns out that any dynamic memory needing to be allocated (normally with the "new" keyword, but Strings count too) are allocated on the heap and the pointer is stored on the stack.

This is an elegant solution and allows local, primitive value types to be cleaned up when the function returns, yet dynamic, reference type values aren't. It is true that the variable holding the pointer is removed from memory, but the data it points at isn't. For example, when we return from main, the variables "msg" and "pt" will be removed but the memory they point to, i.e. "World" and {X:1, Y:0, Z:0} will not be removed. If the program continues to run, these values will continue to be in the heap. Actually, even if there are no references to them, they still could reside in memory. This isn't too good. Let's fix it. Hey Garbage Collector, it's your turn!

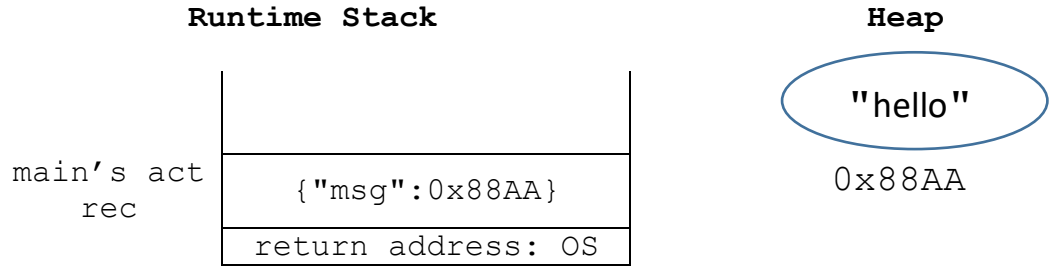
Garbage Collector

In some languages, for example C++, when we create memory we must delete it. This is done by the programmer. So, for example, we can have the following code:

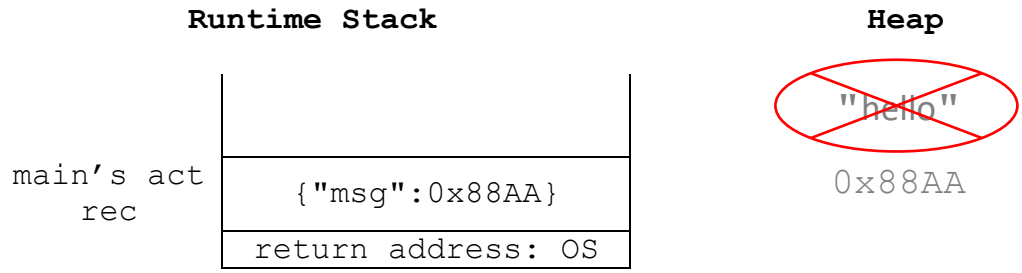
```
int main() {
  // create a pointer and allocate
  // memory for it to point to
  char* msg = new char("hello");

  /* do things with msg */
```

```
  // manually deallocate memory
  delete msg;

  // be safe about it
  msg = NULL;

  /* continue on with the program */
}
```

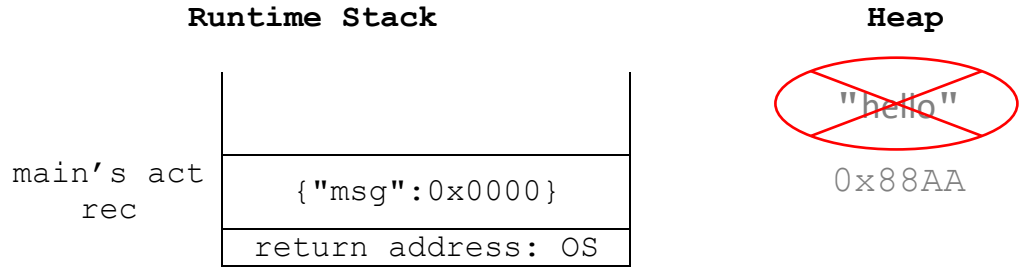
With this code, the variable “msg” is stored on the stack along with its value, which is a pointer to the word “hello” stored on the heap. I.e.:



When we use the “delete” keyword, we are manually removing the memory from the heap:



Now “msg” points to reclaimed memory. We call this a “dangling pointer”. To keep things safe and prevent us from referring to memory that has already been reclaimed (i.e. deallocated), we can set the pointer to null:



Recall that null is really just memory location zero. Now if we want to refer to the memory that “msg” points to, we can do an if statement first:

```
if (msg != NULL) {  // this will work to->  if (msg != 0) {
  /* do something with msg */
}
```

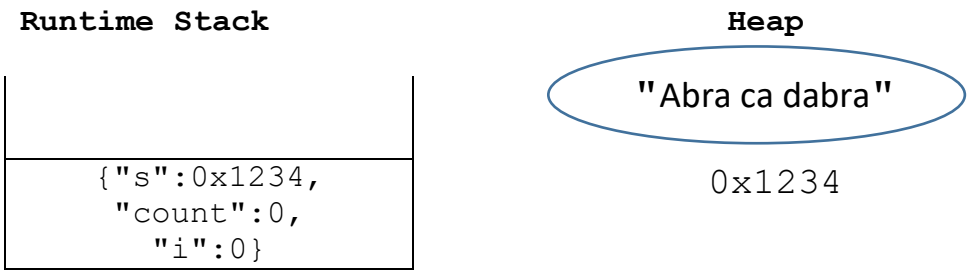
Making msg point to null helps the if statement prevent us from accessing invalid memory. If we no longer use the memory for the rest of the application, yet we don’t deallocate it, this is known as a “memory leak”. C++ is

actually a very hard language to learn and master, and part of the reason is because you have to do things like this manually[3]. You have to delete the memory you create, or risk a memory leak, which can cause major issues in your application if large amounts of memory remain unclaimed yet are not being used. Fortunately, languages like Java, Python and C# have what's called a "garbage collector". This is a program that runs when the application runs, and attempts to identify "garbage", or memory on the heap that is no longer needed but hasn't been deallocated. It will then deallocate this garbage so the programmer doesn't have to worry about it.

Consider the following code:

```
String s = "Abra ca dabra";
int count = 0;
for(int i = 0; i < s.size(); i++) {
  if (s.charAt(i) == 'a') {
    count++;
  }
}
System.out.println(count);
```

This will create a variable on the stack called 's' that will point to memory on the heap that stores the string "Abra ca dabra".

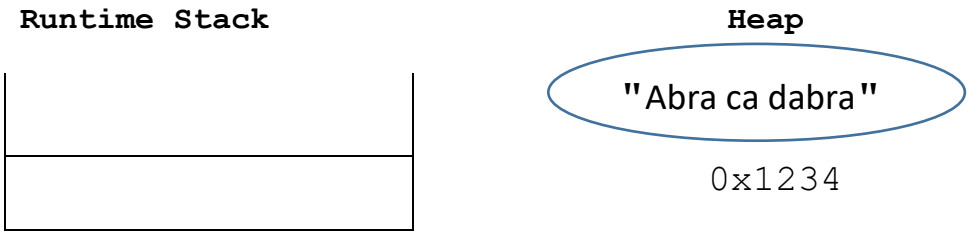


There are two ways this memory on the heap could be considered garbage. One is if the variable "s" is destroyed. For example, let's put this code into a function:

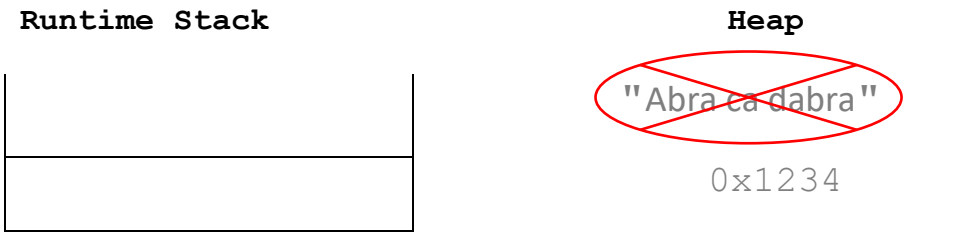
```
void doMagic() {
  String s = "Abra ca dabra";
  int count = 0;
  for(int i = 0; i < s.size(); i++) {
    if (s.charAt(i) == 'a') {
      count++;
    }
  }
  System.out.println(count);
}
```

Since "s" is a local variable, it will become part of doMagic's activation record, which is popped from the stack when the function exits. This leaves the following:

[3] To be fair, it is also a very powerful language with runtime speeds faster than most any other language out there! So there is a tradeoff here.



With our stack variable gone, the heap memory can be reclaimed as it would be considered garbage.



This process isn't always so straightforward though. What if we have the following:

```
String doMagic() {
  String s = "Abra ca dabra";
  int count = 0;
  for(int i = 0; i < s.size(); i++) {
    if (s.charAt(i) == 'a') {
      count++;
    }
  }
  System.out.println(count);
  return s;
}
```

Since the function now returns "s", we can't reclaim that memory. The variable "s" is still destroyed after the function exits, but the memory address that it points to (the memory location of "Abra ca dabra" on the heap) is returned. If we reclaim this memory, we will be returning a dangling pointer. So how does Java tell the difference since sometimes we should reclaim the heap memory and other times we shouldn't.

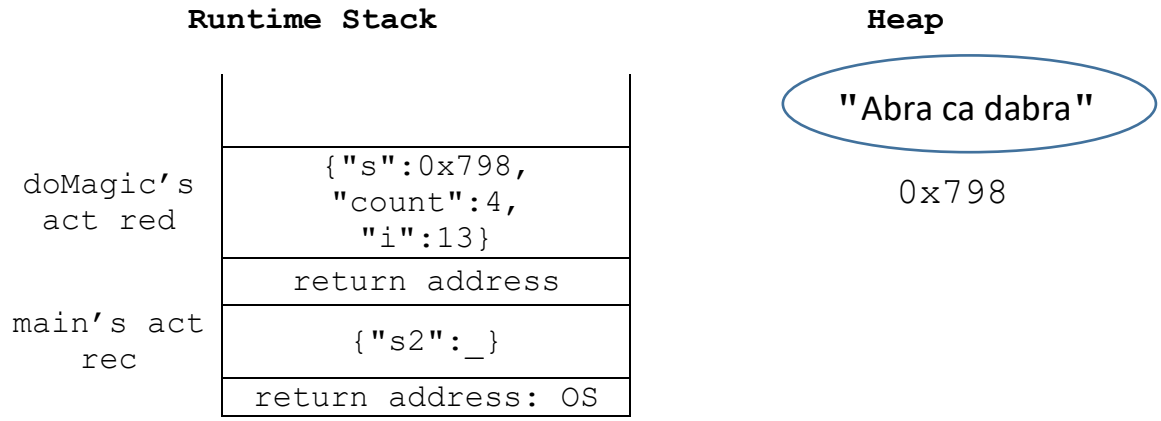
Java (and many other languages with Garbage Collectors) use something called "reference counting". The idea is simple, keep track of the number of references to a particular piece of memory on the heap. If that count ever hits zero, then there are no more active references to the memory and that memory can be reclaimed.

Now consider this code (we are adding a call to the function):

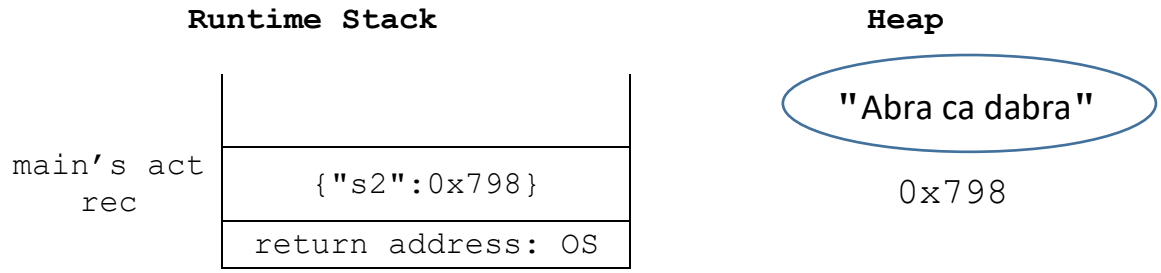
```
class Main {
  public static void main(String[] args) {
    String s2 = doMagic();
  }
  private static String doMagic() {
    String s = "Abra ca dabra";
    int count = 0;
    for(int i = 0; i < s.size(); i++) {
```

```
      if (s.charAt(i) == 'a') {
        count++;
      }
    }
    System.out.println(count);
    return s;
  }
}
```

At the end of the doMagic function (before we return), we have the following:



As you can see, we have 1 reference to our heap memory, that being "s" in doMagic's activation record. When we return from the function, we have the following:



Since doMagic returns the reference to the string, and since "s2" in main is assigned to that reference, our reference count remains at 1 and we do NOT reclaim the memory. This is how we know whether to deallocate the string or not. If we were to create more variables, the reference count would go up:

```
public static void main(String[] args) {
  String s2 = doMagic();
  String s3 = s2;
  String s4 = s2;
}
```

Since s2, s3 and s4 all point to the string, its reference count is now 3. The only way to decrease a reference count is when either the variable goes out of scope (such as when a local variables is destroyed by a function return, as what happened with "s" in doMagic), or an existing reference is set to a different memory location.

For example:

```
// allocates memory on heap with "a" pointing to it
// ref count: 1 for "hello"
String a = "hello";

// allocates memory on heap with "b" pointing to it
// ref count: 1 for "world"
String b = "world";

// c points to a
// ref count: 2 for "hello"
String c = a;

// d points to b
// ref count: 2 for "world"
String d = b;

// c now points to b (since d points to b)
// ref count: 3 for "world"
// ref count: 1 for "hello"
c = d;

// c now points to null, decreasing ref count by 1 for "world"
// ref count: 2 for "world"
// ref count: 1 for "hello"
c = null;

// a now points to null, decreasing ref count by 1 for "hello"
// ref count: 2 for "world"
// ref count: 0 for "hello" (this memory can now be reclaimed)
a = null;

// b and d now point to null, decreasing ref count by 2 for "world"
// ref count: 0 for "world" (this memory can now be reclaimed)
b = null;
d = null;
```

So in Java, to suggest to the Garbage Collector that memory is no longer needed, either a local variable is destroyed when exiting a function (and that causes its reference count to go to zero), or we set the last reference to null (or to some other memory location). We can even alter our code above to the following:

```
class Main {
  public static void main(String[] args) {
    doMagic();
  }
  private static String doMagic() {
    String s = "Abra ca dabra";
    int count = 0;
    for(int i = 0; i < s.size(); i++) {
      if (s.charAt(i) == 'a') {
        count++;
```

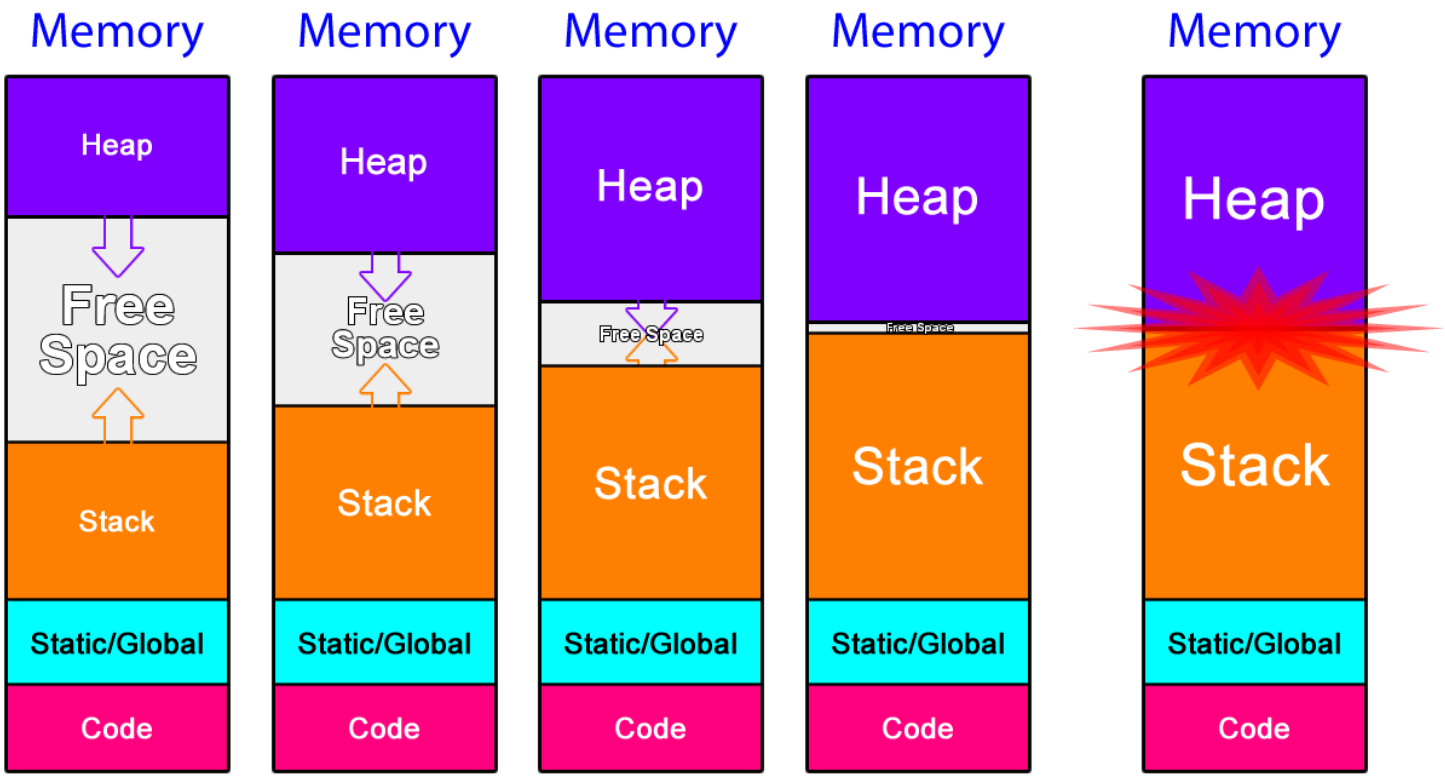
```
        }
      }
      System.out.println(count);
      return s;
    }
}
```

Notice how we don't capture the returned value of doMagic. This means that the reference "s" in doMagic is destroyed and no other reference takes its place. This is legal in Java and would result in a reference count of 0 for the "Abra ca dabra" memory, causing it to be garbage collected.

Garbage Collectors aren't flawless and reference counting isn't a perfect way to detect garbage. Actually, Garbage Collectors use a bunch of different techniques to try and prevent memory leaks, and yet memory leaks are still possible in languages that have Garbage Collectors. While they aren't perfect, they do a pretty decent job, and are getting more intelligent all the time.

Back to Memory Layout (again)

So why is it so important to reclaim this unused memory. After all, when a program exits, all memory it used will be reclaimed anyway. The reason why this is so important, is that as the heap grows, the stack also grows. Each of these only have so much room in memory to grow (normally decided by either the operating system or the runtime environment, i.e. JRE). So what happens when they run out of room?



If the heap tries to expand and meets the stack, we get an OutOfMemoryError exception. Consider the following code that shows an example of this by allocating an ArrayList object (whose pointer variable "list" would be stored on the stack but whose contents would be stored on the heap) and filling it with too much

memory. The underscores in the numbers is a feature added in modern Java versions to denote digit separators, so the number can be more easily seen as 2 billion rather than writing it without underscores.

```
import java.util.ArrayList;

public class OOM {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    for(int i = 0; i < 2_000_000_000; i++) {
      list.add(i);
    }
    for(int i = 0; i < 2_000_000_000; i++) {
      list.add(i);
    }
  }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.Arrays.copyOf(Arrays.java:3511)
        at java.base/java.util.Arrays.copyOf(Arrays.java:3480)
        at java.base/java.util.ArrayList.grow(ArrayList.java:237)
        at java.base/java.util.ArrayList.grow(ArrayList.java:244)
        at java.base/java.util.ArrayList.add(ArrayList.java:454)
        at java.base/java.util.ArrayList.add(ArrayList.java:467)
        at OOM.main(OOM.java:7)
```

On the other hand, if the stack tries to expand and meets the heap, we get a famous website:

```
public class OOM {
  private static void InfiniteRecursion() {
    InfiniteRecursion();
  }
  public static void main(String[] args) {
    InfiniteRecursion();
  }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
        at OOM.InfiniteRecursion(OOM.java:3)
        at OOM.InfiniteRecursion(OOM.java:3)
        at OOM.InfiniteRecursion(OOM.java:3)
        at OOM.InfiniteRecursion(OOM.java:3)
        at OOM.InfiniteRecursion(OOM.java:3)
        at OOM.InfiniteRecursion(OOM.java:3)
          .
          .
          .
```

References

- https://www.youtube.com/watch?v=TfJrU95q1J4