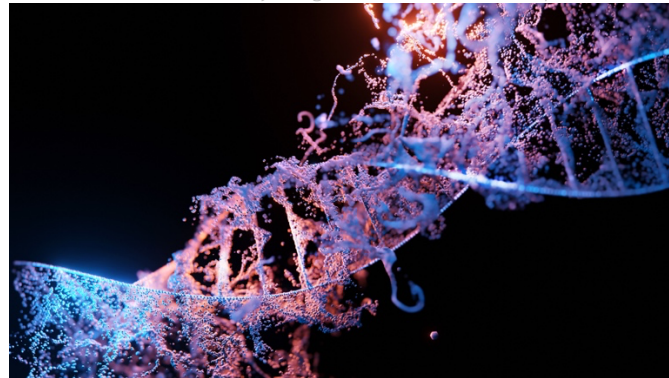




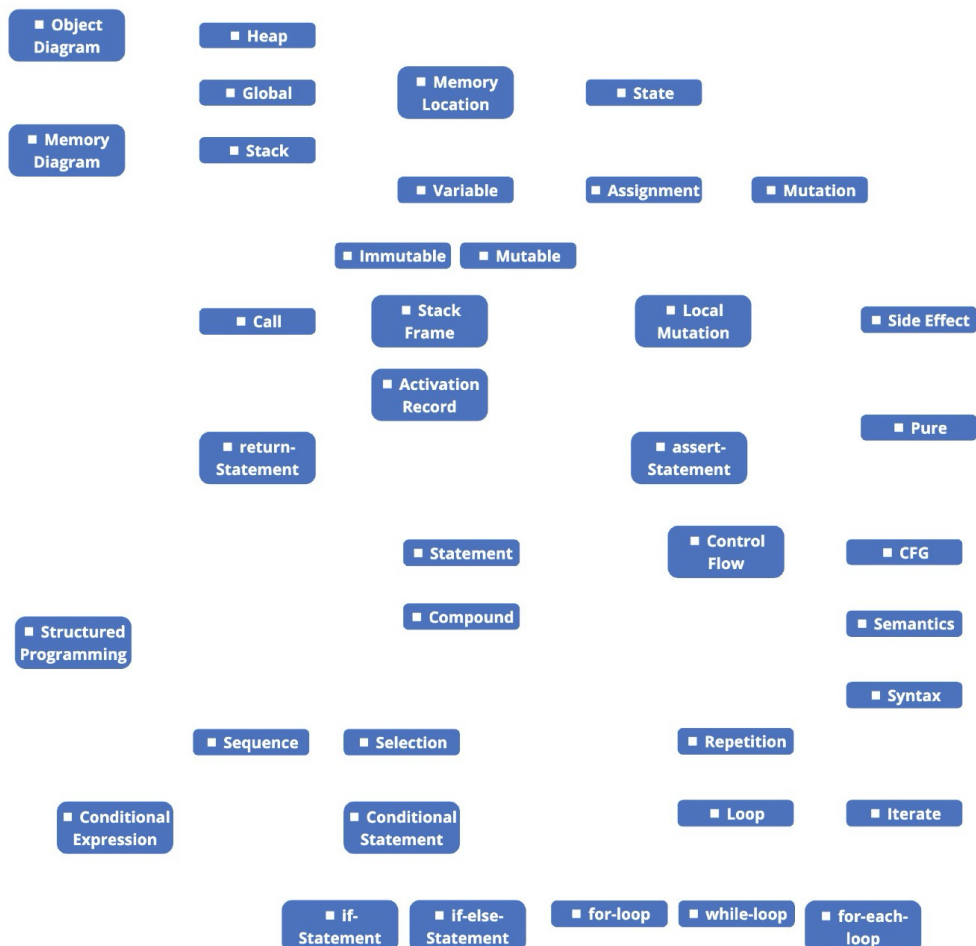
# Statements & Local Mutation

Student name:	TA signature:
---------------	---------------

Based on Photo by Sangharsh Lohakare on Unsplash



Concepts Check off understood concepts, connect related concepts, label connections



Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types  
*There are no new names this week*



### Taking a Snapshot at a Point of a Program Execution

We used **object diagrams** to draw sequences, pairs, and objects with their fields. Let's extend those diagrams into complete **memory diagrams** so we can use them to show the complete picture of the state at a given point in the program execution. They are like a **snapshot of memory**, of every byte of data in use by a program.

The memory of a running program is organized into three areas:

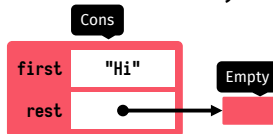
Memory Area	What is stored there?
Heap Memory	Objects (including their instance variables)
Global Memory	Static fields (class variables)
Stack Memory	Call stack frames (method parameters, the magic <code>this</code> variable, local variables)

#### Heap Memory

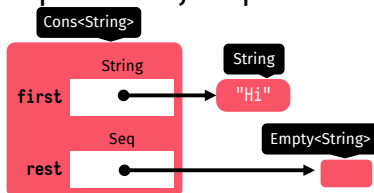
The **heap** contains objects. We sometimes draw those objects in a compact form, with each object being a reddish rounded rectangle, and each field being a white rectangle inside an object:



We can display the same data structure with more detail: we attach black labels to indicate the object's class, and we write the field name to the left of each field.



We can be even more explicit: we write the field's type above the field, for parametric types (like `Cons<T>`), we fill in the type parameter, and we show that strings actually are objects, and thus are not stored *inside* a field, but in a separate object pointed to by a reference (arrow).



One could expand the diagram even more. The `String` object really does not *directly* contain the text; instead, like other objects, it contains a fixed number of fixed-size fields. And one of those fields refers to a data structure containing the characters.

The above three heap diagrams represent **the same** underlying data structure: three objects: a `Cons` pointing to a `String` and to an `Empty`. Depending on the context, we may draw a more compact or more detailed representation.

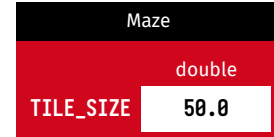
Each object uses up some bytes of heap memory. The size of an object is the sum of the sizes of its fields, plus a few extra bytes for meta information (e.g., about the object's class). Note that the **methods** are **not** stored in the object.



### Global Memory

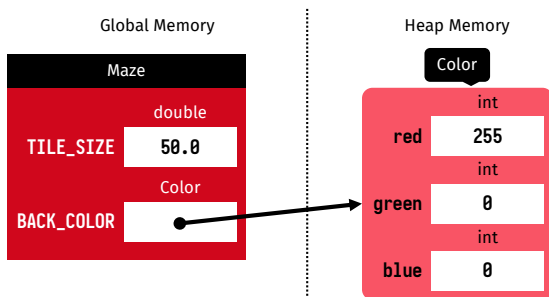
Static fields (class fields) are **not** stored in the objects. Static fields are stored in the **global** memory area. In other languages, Java's static fields could be called "global variables". There is only one variable (one white rectangle) for a static field—it is associated with the class. There can be many variables (many white rectangles) for an instance field—it is associated with each instance (object) of that class. Here is an example class, and the diagram of the global memory:

```
public class Maze {
    public static final double TILE_SIZE = 50.0;
}
```

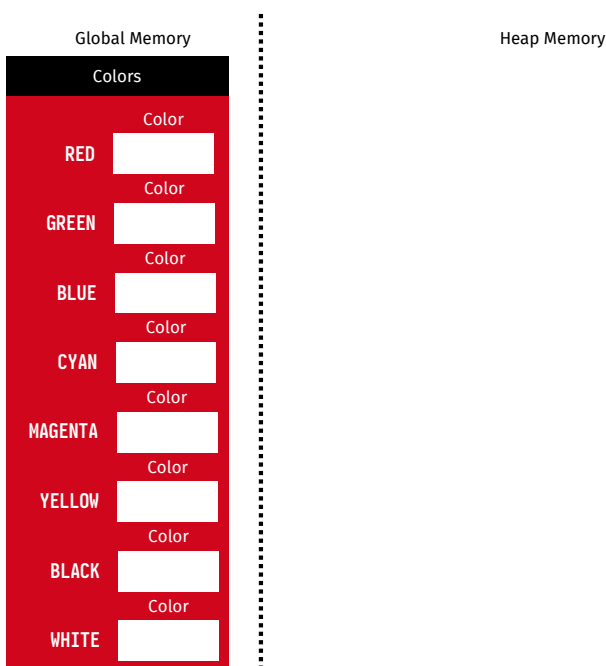


If we want to draw a static field that does not have a primitive type (like double) but a reference type (like Color), we have to draw the **heap memory** (in which the object is placed) in addition to the **global memory** (where the static fields are):

```
public class Maze {
    public static final double TILE_SIZE = 50.0;
    public static final Color BACK_COLOR = rgb(255, 0, 0);
}
```



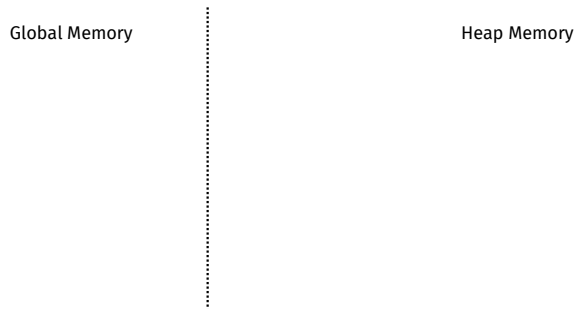
**Draw** the memory diagram showing the static fields of class Colors. Include the eight Color objects on the **heap**. What are the values of all the instance fields?





**Draw the memory diagram** at the time `Num` has been initialized (remember that `Double` is a class, assume it has an instance field named `value` of type `double`):

```
public record Num(int a, String b, Double c) {
    public static final Num ONE = new Num(1, "1", 1.0);
    public static final Num TWO = new Num(2, "2", 2.0);
}
```



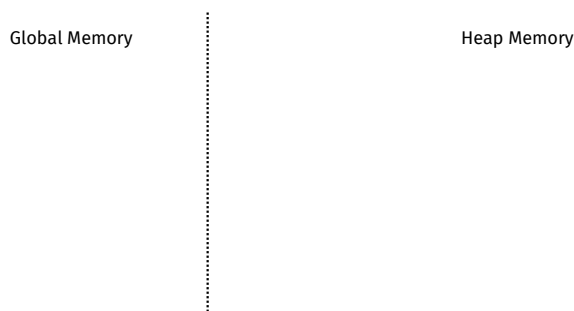
Which of these statements are correct?

- If class `x` declares an **instance field**, each instance of class `x` will contain a variable for that field.
- If class `x` declares a **static field**, the global memory area will contain that one variable for that field.

**Draw the memory diagram** at the time `ReIs` has been initialized:

```
public interface Rel {}
public record LessThan() implements Rel {}
public record Equal() implements Rel {}
public record GreaterThan() implements Rel {}
public class ReIs {
    public static final Rel LT = new LessThan();
    public static final Rel EQ = new Equal();
    public static final Rel GT = new GreaterThan();
}
```

Note that the three record classes don't have any instance fields.



Which of these statements are correct?

- A variable of type `T` can point to an object of a subtype of `T`.
- A variable of type `T` must point to an object of exactly that type `T`.



### Stack Memory

Whenever you call a method, some memory gets allocated on the **call stack** for storing the values of its parameters, the value of `this` (for instance methods), and the values of its local variables. When the method returns, that memory gets freed again. That piece of memory, which holds all the variables for the method to execute, is known as **stack frame** or **activation record**.

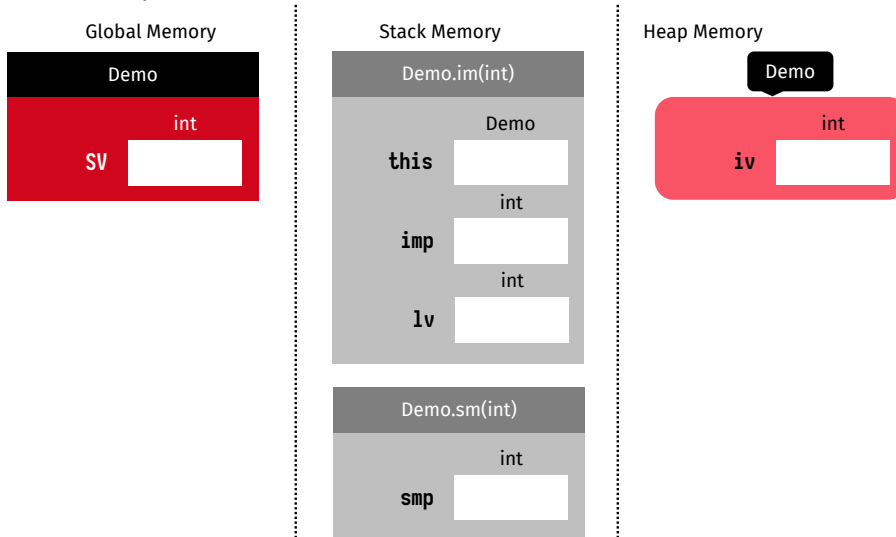
At any given point in time of an execution, the stack contains a number of stack frames, stacked on top of each other. The first method called (to start the execution) is at the bottom of the stack. The most recently called method is on top. Whenever the currently executing method returns, its stack frame (which is on the top of the stack) is erased, and the stack shrinks a bit.

```

public record Demo(int iv) {
    public static final int SV = 2;
    public static int sm(int smp) {
        return new Demo(3).im(4);
    }
    public int im(int imp) {
        final int lv = 5;
        // draw memory diagram at this point
        return lv + imp + SV + iv;
    }
}

```

Here is the memory diagram at the point in the execution indicated by the comment, for the call `Demo.sm(1)`. **Write the value into each variable:**





## Mutation

Up to now our **variables** were not actually **variable**. They were **constant**. They were names we gave **to specific values**. Like in math. If we say  $x = 3$  in algebra, we cannot suddenly say  $x = 4$ . If in a given context we say  $x = 3$ , we can replace  $x$  with 3 or 3 with  $x$ , without this affecting our conclusions.

In algebra there is no notion of time. Things don't change. If in a given context we define something, that definition holds everywhere within that context.

Pure functional programming (e.g., in BSL or Haskell), and the subset of Java we used **up to now**, works exactly like algebra. Whatever we define in a given context will stay like that throughout that entire context.

A name is bound to a value in a variable declaration (e.g., `final double width = 200;`) the same way a name is bound to a value in algebra ( $w = 200$ ).

Now that we introduce **mutation**, this changes! Now **variables** are indeed **variable**. They can change value! If in a given context we read  $x = 3$ , we can **NOT** assume that  $x$  and 3 are interchangeable within that context. We only know that  $x$  is 3 **from that point in the execution until the variable's value gets changed**. We suddenly have to reason about **time!**

We cannot just look at variables as **names for values** anymore, but we have to look at variables as **names for memory locations**. A memory location is a small piece of memory (e.g., one, two, four, or eight bytes). The value of the variable is stored in that memory. If at any time during the program's execution we go and store some other value in that memory, the value of the variable changes.

Our world so far	Our world from now on
<b>IMMUTABLE</b>	<b>MUTABLE</b>
NAME $\rightarrow$ VALUE	NAME $\rightarrow$ MEMORY LOCATION $\leftrightarrow$ VALUE
<pre>public static int run(int distance) {     return step(distance)         + step(distance); }</pre>	<pre>public static void run(int distance) {     step(distance);     distance = distance + 10;     step(distance); }</pre>
<pre>rule [LocalVarDec]:   &lt;k&gt; T:Type X:Id; =&gt; . ...&lt;/k&gt;   &lt;env&gt; Env:Map =&gt; Env[X &lt;- V] &lt;/env&gt;</pre>	<pre>rule [LocalVarDec]:   &lt;k&gt; T:Type X:Id; =&gt; . ...&lt;/k&gt;   &lt;env&gt; Env:Map =&gt; Env[X &lt;- L] &lt;/env&gt;   &lt;store&gt;... .Map =&gt; (L  -&gt; default(T)) ...&lt;/store&gt;   &lt;nextLoc&gt; L:Int =&gt; L +Int 1 &lt;/nextLoc&gt;</pre>

### In Memory Diagrams, Mutability Means Replacing Contents of Boxes

No matter whether we operate in a mutable or an immutable world, throughout program execution, in a memory diagram we have to **create new boxes**, and we have to **initialize** those boxes to some value.

Activity	Memory Area	Kind of boxes created
Loading a class		
Allocating an object		
Calling a method		



In our **immutable** world, in memory diagrams the content of a rectangular box **never** changed. When we created a new box, we wrote a value in it, and we **never** had to change what we wrote.

Now that we enter a **mutable** world, and we look at the rectangular boxes as memory locations, we have to worry about the contents of boxes getting overwritten (by some code using an assignment, =).

For now, we limit mutation to the stack. We call this **local mutation**. If we change the value of a **local variable**, we only need to worry about this for the duration of the method call. Once the method returns, its stack frame gets popped off the call stack (and thus its local variables vanish). Local mutations won't affect other code in other methods! They are less dangerous.

### In a Mutable World, Order Matters

Given that now time matters (when figuring out the value of a variable), **order matters** as well. The following two code snippets contain the same three statements, just in a different order. **Write down** the value of *y* at the end:

1	<code>int x = 10;</code>	<code>int x = 10;</code>
2	<code>int y = 2 * x;</code>	<code>x = 5;</code>
3	<code>x = 5;</code>	<code>int y = 2 * x;</code>
	y now is:	y now is:

**Draw** the stack frames (for a memory diagram), after executing each code line:

1		
2		
3		

In a language without mutability, we wouldn't really need to worry about order (we couldn't write this kind of code, and we wouldn't really *need* to write it).

### Mutation is a Side Effect

Our immutable world was **pure**. If we evaluated an expression, we got back a value, and that's **ALL** that happened. The process of evaluation didn't change anything. It just "read" things and produced a result.

In the new, mutable world, we can't rely on this anymore. Besides producing a value, evaluating an expression can also mutate state (change variable values). We say that the expression has a **side effect**. In a pure world, there are **no** side effects.

**Draw** the expression tree of this **crazy** but legal expression:

$$x + (x = 2) + (x = 9 + x)$$

Assume `int x = 5;`

Annotate each node with its **value**.  
And what's the value of *x* afterwards?



## A Helpful Loop Statement: For-Each Loops

Let's meet the probably most common kind of loop: the `for`-each-loop. Let's look how we can use such a loop for mapping.

### Three Ways to Map

Let's look at **three different ways** (recursion, HoF, loop) to implement the same mapping computation. The computation should convert a sequence of integers into a sequence of strings, by calling `String.valueOf(...)` on each element.

In the methods below, **underline the similar pieces**:

#### Using recursion:

```
public static Sequence<String> is2ss(Sequence<Integer> numbers) {  
    return isEmpty(numbers)  
        ? empty()  
        : cons(  
            String.valueOf(first(numbers)),  
            is2ss(rest(numbers))  
        );  
}
```

#### Using a HoF – the higher-order function map:

```
public static Sequence<String> is2ss(Sequence<Integer> numbers) {  
    return map(n -> String.valueOf(n), numbers);  
}
```

#### Using a for-each loop:

```
public static Sequence<String> is2ss(Sequence<Integer> numbers) {  
    Sequence<String> result = empty();  
    for (final Integer number : numbers) {  
        result = cons(String.valueOf(number), result);  
    }  
    return result;  
}
```

How does this last one, with the loop, work? The first **statement declares** a local variable `result` of type `Sequence<String>` and **initializes** its value to an empty sequence. The second statement is a **compound statement** (a statement that may contain other statements). More specifically, it is a **for-each loop**. The last statement is a **return-statement**, which returns whatever value is stored in the local variable `result` **at this time**.

**for** (**HEADER**) **BODY**

Let's look at the for-each loop in detail. It consists of the keyword **for**, a header, and a body. The **header** specifies over which data structure to iterate (in this case, we iterate over the `numbers`). The **body** specifies what to do **for each** element.

**for** (   
 ) **BODY**    <- **write** the header of the above loop





The header of a for-each loop consists of two parts: it **declares** a local variable (here of type `Integer` with name `number`), and after the colon (`:`) comes an expression providing an iterable data structure (e.g., a `Sequence`). The for-each loop will then **iterate** through the iterable, and **for each** element it will (1) assign the element to the specified local variable and (2) execute to loop body. The loop body may read and use the current value from that local variable.

**for** (**HEADER**) \_\_\_\_\_ <- **write** the body of the above loop

The loop body usually is a **statement sequence** (zero or more statements put inside curly braces `{...}`).

Let's trace through the execution of `is2ss(of(1, 2))`, showing the three involved variables and their values at each point in the execution:

Statement \ Variable values after statement:	numbers	number	result
<code>Sequence&lt;String&gt; result = empty();</code>	<code>of(1, 2)</code>		<code>empty()</code>
<b>for header:</b> <code>number = 1</code>	<code>of(1, 2)</code>	1	<code>empty()</code>
<code>result = cons(String.valueOf(number), result)</code>	<code>of(1, 2)</code>	1	<code>of("1")</code>
<b>for header:</b> <code>number = 2</code>	<code>of(1, 2)</code>	2	<code>of("1")</code>
<code>result = cons(String.valueOf(number), result)</code>	<code>of(1, 2)</code>	2	<code>of("2", "1")</code>
<b>for header:</b> determines that the repetition is done	<code>of(1, 2)</code>		<code>of("1", "2")</code>
<b>return</b> <code>result;</code>	<code>of(1, 2)</code>		<code>of("1", "2")</code>

**Indicate** which of the following observations are correct:

<input type="checkbox"/>	There are three variables in the stack frame of this method: the parameter <code>numbers</code> , the local variable <code>result</code> , and the local variable <code>number</code>
<input type="checkbox"/>	The black = are <b>initializations</b> , the red = are <b>changes</b>
<input type="checkbox"/>	The parameter <code>numbers</code> comes into existence and gets <b>initialized</b> each time the method is called
<input type="checkbox"/>	Within a single execution of the method, the value of the parameter <code>numbers</code> stays constant
<input type="checkbox"/>	The local variable <code>result</code> comes into existence, and gets <b>initialized</b> , when the first statement gets executed
<input type="checkbox"/>	The for header executes <b>multiple</b> times
<input type="checkbox"/>	The for body executes <b>multiple</b> times
<input type="checkbox"/>	The for header executes once more than the body
<input type="checkbox"/>	The local variable <code>result</code> gets <b>initialized</b> before the loop, and gets <b>changed</b> each time the loop body executes – we use it to accumulate the result
<input type="checkbox"/>	The local variable <code>number</code> does <b>not</b> exist before and after the loop
<input type="checkbox"/>	The local variable <code>number</code> comes into existence, and gets <b>initialized</b> , each time the loop header executes (except the last time)
<input type="checkbox"/>	Within a single iteration of the loop body, the value of the local variable <code>number</code> remains constant
<input type="checkbox"/>	Within a single iteration of the loop body, the value of the local variable <code>result</code> gets changed (i.e., the variable is not guaranteed to have the same value throughout an individual execution of the loop body)



Let's juxtapose the for-each-loop and the use of map:

map HOF	for-each loop
<pre>return map(   (Integer n) -&gt; String.valueOf(n),   numbers )</pre>	<pre>Sequence&lt;String&gt; result = empty(); for (final Integer n : numbers) {   result = cons(     String.valueOf(n),     result   ); } return result;</pre>

Can you see the **similarities**?

For each element in the numbers sequence, we assign it to variable `n` (which has type `Integer`) and we convert the `Integer` into a `String`.

Can you see the **differences**?

What is **different** in the method that uses a **loop**?

- The method body consists of **multiple** statements
- We declare and **initialize** a local variable (`result`)
- We **change** the value (**mutate**) of the local variable using an **assignment**

On a deeper level, using loops necessitates the following concepts:

- **mutable state** – the name `result` refers to a **variable**, thus, if we look at the value of `result` **at different points in time during a single method execution, we will see different values**
- **control-flow** – the execution of a method body now cannot be understood anymore by a simple bottom-up evaluation in a **tree**, but it now requires the more complex—possibly cyclic—execution of statements in a **graph**, where each of those statements can contain multiple expressions.




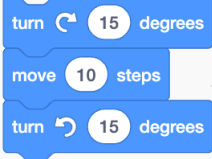
## Statements

Imperative languages like Java provide different kinds of **statements**. We learned about **return statements** in our first week. Later we learned about **assert statements**. Now we learn about several other kinds of statements.

### Syntax: Composing Statements


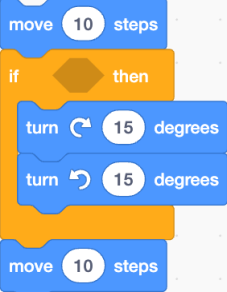
The body of a method consists of a sequence of statements. Each statement could be preceded by and followed by another statement. The statements of a method are executed in order from top to bottom.

Here is a single statement and a sequence of three statements, in Java and in the "block-based" visual language Scratch. Scratch was developed for beginners. To make it obvious how to compose statements, the blocks in Scratch look a bit like puzzle pieces: statement blocks snap together in a top-to-bottom sequence.

	Java	Scratch
<b>Statement</b>	<code>move(10);</code>	
<b>Statements composed into a sequence</b>	<code>turnRight(15); move(10); turnLeft(15);</code>	

Some statements, such as `if`, `while`, `for`, and `for-each`, are **compound statements**. Compound statements, in addition to being preceded by and followed by a statement, also **contain** other statements.

Here is an example of a compound statement (an `if`-statement), in Java and in Scratch. Note the hexagonal hole in Scratch, which corresponds to the place where you need to plug in the condition (an expression producing a boolean):

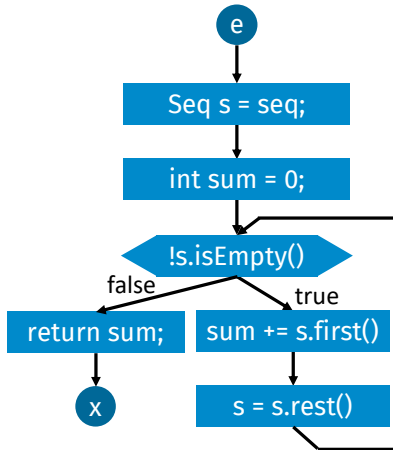
	Java	Scratch
<b>Compound Statement</b>	<code>if (CONDITION) BODY</code>	
<b>Compound Statement composed with other statements</b>	<code>move(10); if (CONDITION) {     turnRight(15);     turnLeft(15); } move(10);</code>	

So we just learned how we can compose statements. This description and the pluggability-constraints provided by Scratch blocks focus on **syntax**: how statements can be composed. The description did not really explain the **semantics**: what those statements **mean**, what they will do.



### Semantics: Control-Flow Graphs

With compound statements, the **control flow** becomes more complex than a straight-line sequence. We can represent the control flow within a method with a directed graph: a **control-flow graph**. Non-programmers might call this a “flow chart”. Compiler people call it a **CFG**.



The **nodes** of a control-flow graph are statements. The **edges** tell us from which statement execution can go to which next statement. Most nodes are drawn as **rectangles**.

A control-flow graph must have one **entry node** (where the execution of the method starts), and one **exit node** (where the execution of the method ends). We draw those special nodes as **circles** containing the letter "e" (entry) and "x" (exit).

Some nodes will have **multiple outgoing edges** (meaning multiple possible next statements). In assembly code, such nodes correspond to branch instructions (e.g., IFEQ in IJVM), where, depending on some condition, they either go one way or another. In source code, such nodes correspond to the headers of loops and conditional statements. In a CFG, such nodes are drawn with a **diamond** shape, and their outgoing edges are labeled with the value of the condition.

**Execute** the above control-flow graph showing the values in all variables after each node executed:

After this node	Value of seq	Value of s	Value of sum
e	of(1, 2)		
Seq s = seq;			
int sum = 0;			
!s.isEmpty()			
x			



## Structured Programming

A control-flow graph can look like a heap of spaghetti. If you look at programs written in assembly languages, this can often be the case. There can be edges from any node to any other nodes. Figuring out the order in which things execute can be like following a spaghetti on your plate from its start to its end. It's a mess. This kind of code is called "spaghetti code".

Most modern languages are designed so that you CANNOT write spaghetti code. You cannot **go** from any statement **to** any other statement (i.e., there is no goto-statement, no arbitrary jumping around). Modern languages force you to **structure** the control flow. This is called **structured programming**.

In structured programming there are **three** constructs for structuring control flow: **sequence**, **selection**, and **repetition**.

### Sequence

The simplest structured programming construct is the **sequence** of statements.

**Draw** the control-flow graph of this method (include the entry and exit nodes):

```
public static int m(int a, int b) {
    int sum = a + b;
    int product = a * b;
    return analyze(sum, product);
}
```

### Selection

Conditional statements, such as **if-statements** and **if-else-statements**, allow us to execute certain other statements only if a given condition holds.

**Draw** the control-flow graph of this method (include the entry and exit nodes):

```
public static int m(int a, int b) {
    if (b == 0) {
        return 0;
    }
    return a / b;
}
```

**Draw** the control-flow graph of this method (include the entry and exit nodes):

```
public static int m(int a, int b) {
    if (a > b) {
        return "GT";
    } else {
        if (a < b) {
            return "LT";
        } else {
            return "EQ";
        }
    }
}
```



We have been using conditionals for a long time. However, we limited ourselves to **conditional expressions**. Now we can also use **conditional statements**. They simply provide a different way to express conditional computations.

**Refactor** between the two variants:

Conditional Expressions	Conditional Statements
<pre>return isEmpty(numbers)     ? empty()     : cons(         "" + first(numbers),         i2s(rest(numbers))     );</pre>	<pre>if</pre>
<pre>return</pre>	<pre>if (isEmpty(numbers)) {     return empty(); } else {     if (first(numbers) &gt;= 0) {         return cons(             first(numbers),             pos(rest(numbers))         );     } else {         return pos(rest(numbers));     } }</pre>

In professional programming it's customary to use conditional operators for **short** expressions. When expressions are **longer**, and especially when we need to **nest** conditionals, we usually use nested `if`-statements instead.

### Repetition

The structured programming construct we can use for repetition is the **loop**. Java provides several kinds of loops, like **for-each-loops**, **for-loops**, or **while-loops**.

**Draw** the control-flow graph of this while-loop:

<pre>public static int sum1(     Sequence&lt;Integer&gt; seq ) {     int total = 0;     Sequence&lt;Integer&gt; r = seq;     while (!r.isEmpty()) {         total = total + r.first();         r = r.rest();     }     return total; }</pre>	<p><b>CFG:</b></p>
--	--------------------

**Draw** the control-flow graph of this while-loop:

<pre>public static int sum2(     Sequence&lt;Integer&gt; seq ) {     int total = 0;     int i = 0;     while (i &lt; length(seq)) {         total = total + get(i, seq);         i = i + 1;     }     return total; }</pre>	<p><b>CFG:</b></p>
---	--------------------



Indicate which of the following claims are correct:

- Methods `sum1` and `sum2` on the previous page are semantically equivalent
- Methods `sum1` and `sum2` have the same algorithmic complexity
- Methods `sum1` and `sum2` both mutate local variables
- Method `sum1` does not use an index to access the elements
- Method `sum2` uses an index to access the elements

Refactor the above methods to use `reduce` or `map` or `filter`:

```
public static int sum3(Sequence<Integer> seq) {
    return
}
```

Some repetitive computations **cannot** be expressed with `reduce`, `map`, or `filter`. But every repetitive computation **can** be expressed using recursion.

Refactor this recursion into a `while`-loop:

Recursion	while-Loop
<pre>public static int sum(int i) {     return i != 0         ? sum(i - 1) + i         : 0; }</pre>	<pre>public static int sum(int i) {     int s = 0;     while (i &gt; 0) {         s = s + i;         i = i - 1;     }     return s; }</pre>

Instead of a `while`-loop with index, it usually makes sense to use a `for`-loop.

Refactor this `while`-loop with index into a `for`-loop with index:

while-Loop	for-Loop
<pre>public static String stars1(int n) {     String s = "";     int i = 0;     while (i &lt; n) {         s = s + "*";         i = i + 1;     }     return s; }</pre>	<pre>public static String stars2(int n) {     String s = "";     for (int i = 0; i &lt; n; i++) {         s = s + "*";     }     return s; }</pre>
CFG:	CFG:

Indicate which of the following claims are correct:

- Methods `stars1` and `stars2` are semantically equivalent
- Given a CFG, we cannot tell whether it came from a `while`-loop or a `for`-loop