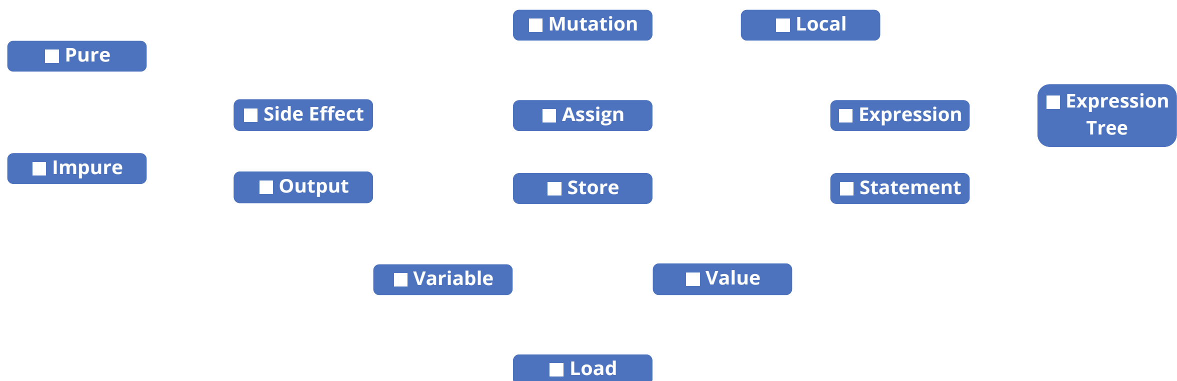
More Mutation, Data Flow

Student name:	TA signature:

Photo by Robert Lukeman 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

`System` • `PrintStream` • `print` • `println`

Ways to Mutate a Variable

To mutate a **variable**, we need to **assign** a new **value** to it. For this, we can use the assignment operator (`=`). Java also provides other operators that mutate variables.

Complete this table. Assume at the start all variables contain the value `10`:

Code	Operator name	Value of variable	Value of expression
`a = 1`	Simple assignment		
`b += 1`	Compound assignment		
`c -= 1`	Compound assignment		
`d++`	Postfix-increment		
`++d`	Prefix-increment		
`d--`	Postfix-decrement		
`--d`	Prefix-decrement		

All those operators have **side effects**: beside evaluating to a value, they also mutate a variable.

Complete this table by refactoring the given expression:

Simple Assignment	Compound Assignment	Increment/Decrement
`a = a + 1`		
`b = b - 1`		
`c = c + 2`		
`d = d - 2`		

Compound assignment operators are rarely used in Java, because the most common case, adding or subtracting 1, is supported by the shorter increment and decrement operators. There are other compound assignment operators than `+=` and `-=`, but they are extremely rarely used.

That's all there is! There's no other way to **mutate** variables in Java.

The Identity of an Object – Comparing References

The `==` operator **compares** two things. But watch out: If you use `==` for comparing reference values, you compare the references (identities, addresses) of the objects, **not** the contents (fields) of the objects!

```
public record Coordinate(int x, int y) { }
```

Code	Stack & Heap	Evaluates to
`new Coordinate(1, 1) ==` `  new Coordinate(2, 2)`		
`new Coordinate(1, 1) ==` `  new Coordinate(1, 1)`		

`Coordinate c =`
`  new Coordinate(1, 1);`
`c == c`		

`Coordinate c1 =`
`  new Coordinate(1, 1);`
`Coordinate c2 = c1;`
`c1 == c2`		

Abuse of Mutation

We started looking at mutation last week. We focused on mutation within a method, i.e., **local mutation** (specifically, mutation of local variables).

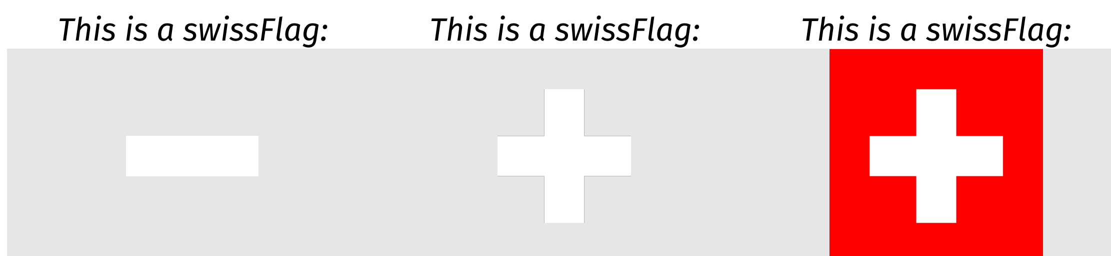
Here are two different ways to implement the Swiss Flag:

<pre>Graphic swissFlag =
  rectangle(200, 60, WHITE);
swissFlag =
  overlay(
    swissFlag,
    rotate(90, swissFlag)
  );
swissFlag =
  overlay(
    swissFlag,
    rectangle(320, 320, RED)
  );</pre>
	<pre>final Graphic horizontalBar =
	  rectangle(200, 60, WHITE);
	final Graphic cross =
	  overlay(
	    horizontalBar,
	    rotate(90, horizontalBar)
	  );
	final Graphic swissFlag =
	  overlay(
	    cross,
	    rectangle(320, 320, RED)
	  );</pre>

The left one uses mutation, the right one does not. There really is little reason for using mutation in this case. We have three different statements, each one executes once, and each one produces a specific Graphic.

- ☐ Defining three immutable variables instead of one mutable variable does not actually increase memory consumption (the compiler will optimize).
- ☐ In the right code, simply by reading the variable name, we know what it contains (if the name is well chosen).
- ☐ In the left code, if we reorder the last two statements, the compiler will not complain, but the code will break (it won't produce a Swiss flag).
- ☐ In the right code, if we reorder the statements in a way that would break the code, the compiler will complain. If we reorder the statements in a way that does not break the code, the compiler will be happy.
- ☐ In the left code, after each statement `swissFlag` means something **different**! First it means a white rectangle, then it means a white cross, and then it means a Swiss flag.

This is a swissFlag: *This is a swissFlag:* *This is a swissFlag:*



Really???

This is a bad use of mutation. It's unnecessary. Either find appropriate names, or inline the subexpressions.

Statements Want Side Effects

Expressions implicitly pass information to other expressions when they are nested.

A subexpression produces a value, and that value flows into the hole of its parent expression. This information flow is well organized into a tree (the **expression tree**), where children pass values to their parents.

Expression	Expression Tree (with values)
`(a + b) > (a * b)`	

<table>
<tr><td>Assume the above expression is placed in the body of a method:
<pre>public static boolean e(
  int a, int b
) {
  return (a + b) > (a * b);
}</pre></td><td>Draw the stack frame of <code>e(2, 3)</code>:</td></tr>
</table>

Statements do not have that kind of connection with each other: A statement that sits above another statement, and thus executes before the other statement, does **not** implicitly pass values to the other statement.

If you want to pass data from one statement to the other, you **have to** do that explicitly via **variables**: one statement needs to **store** the value in a variable, and the other statement can then **load** that value from the variable.

```
int sum = a + b;       // store value in variable sum (side effect)
int product = a * b;   // store value in variable product (side effect)
return sum > product; // load value from variables sum and product
```

<table>
<tr><td>Assume the above statements are placed in the body of a method:
<pre>public static boolean s(
  int a, int b
) {
  int sum = a + b;
  int product = a * b;
  return sum > product;
}</pre></td><td>Draw the stack frame of <code>s(2, 3)</code>:</td></tr>
</table>

A statement **without** a **side effect** cannot pass information to other statements:

```
Math.sin(x);           // no effect (computes sin(x), throws result away)
return "I don't know";
```

Information-Flow Into and Out Of Pure Methods

A **pure** method takes parameters and produces a result that it returns (the return value). Parameters and return values are the **only** way for information to flow into and out of a **pure** method.

No Parameters

If we do **not** want parameters, we can define a method with an empty parameter list. Developers usually don't write pure methods that don't take parameters, because those methods are boring: they always return the same value. Developers replace such methods with constants. Still, we *can* use them, e.g., if somewhere we need a function object that produces a constant:

```
public static int one() { return 1; }
```

No Return Value

If we do **not** want a return value, we can write the word `void` where the return type goes. Developers often say "the return type is void", but strictly speaking, in Java `void` is not a type. A `void` method does not return a value. If we use a `return`-statement inside a `void` method, the `return`-statement cannot contain an expression (because we can't return a value).

```
public static void noRetVal(int p) { return; }
```

Does it make sense to write a **pure** `void` method? Why or why not?

No Parameters, No Return Value

We could even define a method that does **not** have parameters and does **not** have a return value:

```
public static void noParamsNoRetVal() { return; }
```

Write a pure method that does **not** have parameters and does **not** have a return value, and that contains more than one statement:

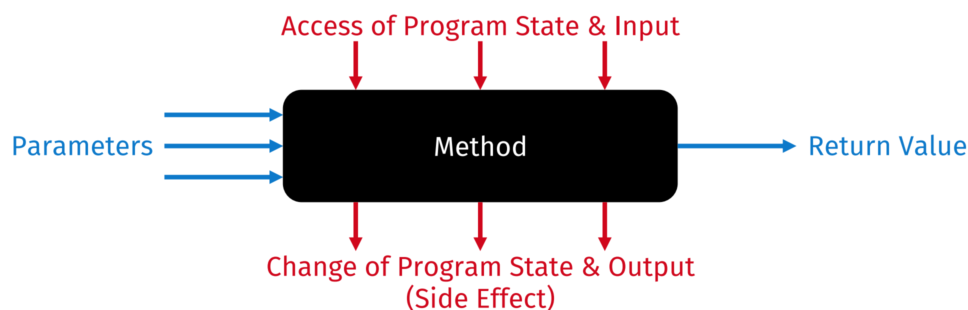
```
public static
```

We haven't used `void` methods in prior worksheets, because all the methods we created in the worksheets were pure. The only place where we wrote `void` methods was in some labs, where we wrote "main" methods. Those are methods that get called when a Java program starts, that finish when the program ends, and that **cannot** return any value. We need "main" methods to run Java code from the command line with `java`.

Information-Flow Into and Out Of Impure Methods

Impure methods can do everything that **pure** methods can do, but additionally, they also can:

- **read** class and instance variables
- **input** data from the outside world (e.g., from a file)
- **write** class and instance variables
- **output** data to the outside world (e.g., to a file)



These four kinds of impurities wield great power; but they are not without risks. They make reasoning about your program harder. Use them with caution.

Output

Let's focus on one of the four kinds of impurities: output. In Java you can **output** text to the terminal in various ways; the most common is to call the `println` method of class `PrintStream`. The `System` class provides a static field named `out`, which refers to a `PrintStream` object that is connected to your terminal (to what is known as "stdout" of your running Java process). If you call `println` on that object, the `String` you pass to `println` will be printed in your terminal.

Write a statement that prints "I have an effect on the world!":

The name `println` stands for "print line". This is because at the end of the text, it prints a magical "newline" character, so that subsequent output will appear on the next line. The `PrintStream` class also has a method `print`, which works like `println` but does **not** print a "newline" in the end.

Write three statements, one that prints "No", one that prints a space, and one that prints "Way!". They must produce all output on the same line, and the last one must move to a new line:

The `PrintStream` class provides many overloaded versions of `println` and `print`. Besides the one that takes a `String`, there also is one for most primitive types.