Expressions and Symbols

1. Expression vs Statement

Expressions and statements are two distinct language constructs.

1. Statements are commands that you **execute** and ask to compute a certain action.
2. Expressions **return values** to you.

For example:

```
def exampleFunction():
        x = 4 # statement (assignment statement)
        if x is 10: # statement
                print(x*2) # statement
        else:
         return x + 4 # expression
```

While in many languages you have expressions and statements. Some languages have chosen to do away with any statements. Everything in those languages are expressions, all the commands are built with expressions. And you compose an application in such a language solely based on expressions.

What are the benefits of that?
By treating everything as expressions you are able to provide a lot more **conciseness** in your code.

- This is because if you have an expression can you can ignore the result if you don't want it. However in a statement, there is no result to be used -- thus statements are fairly limiting.

> Ruby and Groovy are examples of languages that can use everything as expressions. e.g. if/else, for loops, while loops, or even try and catch.

```
def isEven(number)
        result = "odd"
        result = "even" if (number % 2 == 0)
        result #remember that there's no need for a return keyword here.
end
```

```
puts isEven(4)
puts isEven(3)
```

We could take that if statement and treat it as an expression as we have below:

```
def isEven(number)
        result = if (number % 2 == 0)
                "even"
        else
                "odd"

        "The given number is #{result}"
end

puts isEven(4)
puts isEven(3)
```

How about a for loop as an expression?

```
def foo(number)
        for i in 1..4
                puts i
        end

        puts "hello"
end

puts foo(40)
```

> `puts` is an expression itself, and always returns a `nil`.

2. Interning

String interning is a method of storing only one copy of each distinct string value which must be immutable.
Interned strings are stored in a special memory region of the JVM. This memory region is of a fixed size.

Interning of a Sting in Java

When you use expressions your code becomes a lot more concise and simple. You won't spend time making temporary variables, assigning those variables to values, and so on.

```
public class Sample {
        public static void main(String[] args) {
                String bad = new String("hello");

                String greet = "hello";
                System.out.println(bad);
                System.out.println(greet);

                String greet2 = "hello";
                System.out.println(greet2)
        }
}
```

`greet` and `greet2` point to exactly the same instance in memory, whereas `bad` is pointing to a difference instance of `String` in memory.

- You can tell that `bad` is pointing elsewhere as it uses the `new` keyword to allocate new space in memory.

Now let's check if `bad`, `greet`, and `greet2` are equal to one another.

```
public class Sample {
        public static void main(String[] args) {
                String bad = new String("hello");

                String greet = "hello";
                System.out.println(bad);
                System.out.println(greet);

                String greet2 = "hello";
                System.out.println(greet2)

                System.out.println(greet == bad); //outputs false
                System.out.println(greet.equals(bad)) //outputs true

                System.out.println(greet.equals(greet2)) //outputs true
                System.out.println(greet == bad) //outputs true
        }
}
```

> Recall that in Java `==` does a referential comparison while `equals()` does a value based comparison.

It tells us while `greet` and `bad` are equal in value, `greet` and `bad` aren't pointing to the same location. While `greet1` and `greet2` are pointing to the same location.

3. Symbols

Symbols provide you interned values for any variable in memory.

Think of a symbol as a reference to a variable.

```
class Car
        def tow
                puts "tow called..."
        end
        private :tow
end

car = Car.new
car.tow #error message from private member tow called.
```

On line 5 we are sending a symbol, a reference to the function called `tow`. This is not `tow` but however the reference to `tow`.

- We are not using the function, we are not invoking the function, we are referring to the function. And that is all that we are doing.

```
x = 4
puts :x
puts :x.class
```

Here we see we just created a `Symbol` called `x` on line 2.

If you're writing a function that's using meta-programming and you want to receive function references then it becomes effective to use symbols.

```
class Person
        def method_missing(name, *args)
                puts "You called #{name}"
        end
end

sam = Person.new
sam.sing
```

> Examine what `name` is. `name` is not a string it is a symbol.

> Do not confuse these references with the references in C++. This is using interning more than references in C++. You could modify values in that memory location in C++ unlike symbols in Ruby.