CHAPTER 1

Work in Progress

CHAPTER 2

Pharo / Java / C syntax comparison

2.1 Three types of messages

Unary messages: messages without arguments

Pharo

```
list removeAll.
```

Java

```
list.clear();
```

Messages on numbers

In Pharo numbers are plain objects instances of classes `Number`, `Float`, `Integer`. We just send messages to them. A mathematical operation is just a message. It follows the same syntax. There is no such thing as `Math` that holds mathematical operations.

Java

```
Math.sin(200)
```

Pharo

```
200 sin
```

Pharo

```
7 isPrime.
7 asString.
```

Java

```
int x = 7;
(new Integer(x)).toString();
```

In Java primitive types (int, float, ...) are not actually objects, but primitive types. To call methods on them, they must be boxed to classes.

2.2 Binary messages

Binary messages are message with one argument and a non alphabetic name. Messages named with special symbol (+, -, >, ~) and a single argument. Note that binary messages are evaluated left to right without any changes based on the symbols used.

Pharo

```
120 = 5 factorial
>>> true
```

```
2 + 3.
>>> 5
```

```
2 + 3 * 6.
>>> 30
"no special priorities!"
"(2.plus(3)).times(6)"
2 + (3 * 6).
>>> 20
```

Defining binary messages

There is no special syntax to define binary messages. For the operators, we simply define a method in the class and this corresponds to redefining an operator in C++.

Pharo

```
DiceBag >> + aDie
  dice add: aDie
```

Java

```
// not possible
```

C++.

```
class DiceBag {
  DiceBag operator+(const Die& die) {
    dice.push_back(die);
```

```
    return this;
  }
}
```

Note that overloaded C++ operators still follow Math precedence, which doesn't necessarily make sense in a general context.

2.3 Keyword messages

A keyword message is a message with multiple arguments.

Pharo

```
list add: 12.
```

The message is `add:` with argument 12 and it is sent to the `list` variable.

Java

```
list.add(12);
```

Pharo

```
dictionary at: 'key' put: 'value'.
```

The message is `at:put:` with arguments `key` and `value`.

Java

```
dictionary.put("key", "value"); // easy to swap arguments by mistake
```

Pharo

```
3 between: 1 and: 10.
```

Message is `between:and:` with arguments 1 and 10 send to the object 3.

Java

```
(new Integer(3)).between(1, 10); // assuming such method would exist
```

2.4 Message priority

Messages are executed by priority based on their kind (unary, binary, keywords):

Parentheses > unary > binary > keyword

Take care when writing embedded keyword messages:

Without parentheses

```
dictionary at: dictionary at: 'key' put: 12.
```

A non-existing message `at:at:put:` will be sent with arguments `dictionary`, `'key'`, and `12`.

With parentheses

```
dictionary at: (dictionary at: 'key') put: 12.
```

- First the inner send will be evaluated (`dictionary at: 'key'`)
- Then the outer send will be evaluated (`dictionary at: valueFromAbove put: 12`)

2.5 Simple constructs

Statement separator

Every statement is separated from the next one with a `.` dot, equivalent to a `;` semicolon in Java.

Pharo

```
var := 1 + 1.
var2 := var + 2
```

Java

```
var = 1 + 1;
var2 = var + 2;
```

Assignment

- Assignment `:=`
- Equality by value `=` (non-equality `~=`) (do they have the same value?)
- Equality by identity `==` (non-equality `~~`) (are they the same object?)

```
a := 'hello'.
b := 'hello'.
a = b.
>>> true "same value"
a == b.
>>> false "different objects"
```

Strings

- String in single quotes, two single quotes for escaping

```
a := 'string'.
">>> string"
a := 'hello ''world'''.
```

```
">>> hello 'world'"
```

Comments

Comments are in double quotes, two double quotes for escaping.

```
"I am a ""comment"""
```

2.6 Basics of if/while

- Smalltalk doesn't have special syntax for if.
- Everything is object-oriented: just messages sent to objects that execute methods.

if

Java

```
if (person.getAge() > 18) {
  person.chug(beer);
} else {
  bouncer.kickOut(person);
}
```

Pharo

```
(person age > 18) ifTrue: [
  person chug: beer
] ifFalse: [
  bouncer kickOut: person
]
```

- `ifTrue:`, resp. `ifTrue:ifFalse:`, resp. `ifFalse:` are regular methods implemented in the `Boolean`/`True`/`False` objects
- Arguments are regular blocks (see earlier)

while

Java

```
while (!earth.isDestroyed()) {
  people.produceGarbage();
}
```

Pharo

```
[ earth isDestroyed not ] whileTrue: [
  people produceGarbage
]
```

- On every loop the first block is evaluated, and if the result is true, the second block is evaluated
- Implemented recursively in the `BlockClosure` class

for

Java

```
for (int i = 0; i < 50; ++i) {
  ...
}
```

Pharo

```
1 to: 50 do: [ :i |
  ...
]
```

foreach

Java

```
for (Person person : people) {
  ...
}
```

Pharo

```
people do: [ :person |
  ...
]
```

2.7 Basics of Blocks

In other languages also known as anonymous functions, closures, or lambda expressions (note: there are some differences between them, but that's outside of this piece). We pass a function, or a deferred computation as an argument.

Pharo

```
"declaration of a block; the result will be the last statement
    within the block"
msg0 := [ 'hello ' repeat: 2 ].
msg1 := [ :arg1 | arg1 + 10 ].
msg2 := [ :arg1 :arg2 | arg1 + arg2 ].
msg3 := [ :arg |
      | localVar1 localVar2 |
      localVar1 := arg + 1.
      localVar2 := localVar1 + 2.
```

```
      localVar2 ].

"execution"
msg0 value.
">>> 'hello hello '"
msg1 value: 2.
">>> 12"
msg2 value: 2 value: 3.
">>> 5"
msg3 value: 0.
">>> 3"
```

JavaScript

```
var msg3 = function(arg) {
  var localVar1 = arg + 1;
  var localVar2 = localVar + 2;
  return localVar2;
};
msg3(0); // -> 3
```

Java

```
class Msg3 {
  public int value(int arg) {
    int localVar1 = arg + 1;
    int localVar2 = arg + 2;
    return localVar2;
  }
}
(new Msg3()).value(0); // -> 3
```

2.8 Using blocks as Callbacks

Pharo

```
transformText := [ :text :callback | |result|
  result := text asUppercase.
  callback value: result value: text ].

transformText value: 'hello' value: [ :newText :originalText |
  Transcript show: newText ; cr. "'HELLO'"
  Transcript show: originalText; cr. "'hello'" ]
```

JavaScript

```
var transformText = function(text, onCompleteCallback) {
  var result = text.toUpperCase();
  onCompleteCallback(result, text);
}
```

```
transformText('hello', function(newText, originalText) {
  // newText == 'HELLO'
  // originalText == 'hello'
});
```

Using blocks as comparators (sorting by string length)

Pharo

```
cities := #('Antwerpen' 'Prague' 'Lille').
cities sort: [ :a :b | a size < b size ].
>>> #('Lille' 'Prague' 'Antwerpen')"
```

JavaScript

```
var cities = ['Antwerpen', 'Prague', 'Lille'];
cities.sort(function(a, b) { return a.length - b.length; }); //
    #('Lille' 'Prague' 'Antwerpen')
```

Java

```
List<String> cities = Arrays.asList("Antwerpen", "Prague", "Lille");
Collections.sort(cities, new Comparator<String>() {
  @Override
  public int compare(String a, String b) {
    return a.length() - b.length();
  }
}); // {"Antwerpen", "Prague", "Lille"}
```

2.9 Class Creation

Java

```
package people;

class Person {
  String name;
  double age;

  // BEGIN: Accessors
  String getName() {
    return name;
  }
  String setName(String name) {
    this.name = name;
  };
  // END: Accessors

  // BEGIN: testing
  bool isAdult() {
```

```
    int legalAge = 18;
    return age > legalAge;
  }

  bool isAgeBetween(int min, int max) {
    return (age >= lower) && (age <= max);
  }
  // END: testing
}
```

Pharo

```
Object subclass: #NameOfSubclass
  slots: {#name. #age}
  classVariables: {}
  package: 'People'

"accessing"
Person >> name
  ^ name

"accessing"
Person >> name: aName
  name := aName

"testing"
Person >> isAdult
  |legalAge|
  legalAge := 18.
  ^ age > legalAge

"testing"
Person >> isAgeBetween: aMin and: aMax
  ^ age between: aMin and: aMax
```

- ^ is a symbol for return
- `name` variable and methods do not clash
  - because you have to send a message to an object, and all instance variables are protected
  - do NOT use get/set prefixes
- local variables between pipes | (same as inside block earlier)

2.10 Basics of Syntax - Cascade

Also known as fluent interface, but naturally supported in Smalltalk.

In Java, he method must always return itself, otherwise you break the chain.

Java

```
class Person {
  String name;
  double age;
  setName(String name) {
    this.name = name;
    return this;
  }
  setAge(double age) {
    this.age = age;
    return this;
  }
}

Person person = new Person().setName("Ben").setAge(25);
```

In Smalltalk the method can return whatever it wants.

```
person := Person new
  name: 'Ben';
  age: 25;
  yourself
```

- `new` is an unary method that is sent to the `Person class` object and creates a new instance
- the result of the entire statement is the result of the last message send
- `yourself` always returns itself, so the result will be `Person` even if `age:` didn't return the object
- the default return value is `self` (the object), cf. `null` in Java/C++