Refactor

Clean code: naming

A good name should answer three questions:

- Why it exists
- What it does
- How it is used

If a comment is needed for answering any of the questions, it is not a good name.

How to not name

- Misleading names: list (array), number, etc.
- ControlForEfficientHandlingOfStrings
- 1l1ll1l1l1l1l1l1l1l1l1l1l1lll1l111ll1l1ll1l1ll1l1l1l1l
  O00O0O0O0O0O0O00O00O0OO0

How to name

- If two things are similar, name them similarly
- If two things are not similar, don’t name them similarly
- Automatic code completion will help you!

int d; // This variable name reveals nothing

int fileAgeInDays; // This reveals what is being measured and unit of measurement

Make meaningful distinctions

- ***moneyAmount*** is indistinguishable from ***money***,
- ***moneyInRupees*** is clearly distinguishable from ***moneyInDollars***.

Not meaningful

- Noise words (product, productInfo, productData)
- Redundant names (nameString, multiplyFunction)
- Different spellings (class, klass, clas, etc.)
- Adding numbers or letters (name1, name2, objectA, objectB)
- Only numbers or letters

Pronounceable, searchable

- No single (double, triple) letters
- No abbreviations
- (Multiple) entire words

```
private Date genymdhms;
```

```
private Date generationTimestamp;
```

```
  for (int j = 0; j < 34; j++) {
  s += (t[j] * 4) / 5;
  }
int realDaysPerIdealDay = 4;
  const int WORK_DAYS_PER_WEEK = 5;
  int sum = 0;
  for (int = 0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
  }
```

Consistent names

- Nouns for classes and objects (customer, page, account, etc.)
- Verbs for methods and functions (delete, save, get, etc.)
- Don’t be funny or cute (holyHandGrenade vs. deleteItems)
- Stick to your first choice (fetch, get, retrieve, etc.)
- Don’t use double meanings (e.g. ‘add’ for summing and appending)

Necessary length

The longer, clearer name the better.

But don’t use anything that is not necessary.

- If everything needs the prefix, nothing needs it
- Don’t be too specific if you will use it later

Pick one word per concept

Pick one word for one abstract concept and stick with it.

Summary

Good Names	Bad Names
Informative	Misleading
Consistent	Undistinguishable
Meaningful	Funny, smart
Clear	Redundant

The Single Responsibility Principle (SRP)

A class should have one and only one reason to change, meaning that a class should have only one job.

```
class Book {
  getTitle() {
    return "A Great Book";
  }
  getAuthor() {
    return "John Doe";
  }
  turnPage() {
    // pointer to next page
  }
  printCurrentPage() {
    return "current page content";
  }
}
```

```
class Book {

  getTitle() {

    return "A Great Book";

  }

  getAuthor() {

    return "John Doe";

  }

}
```

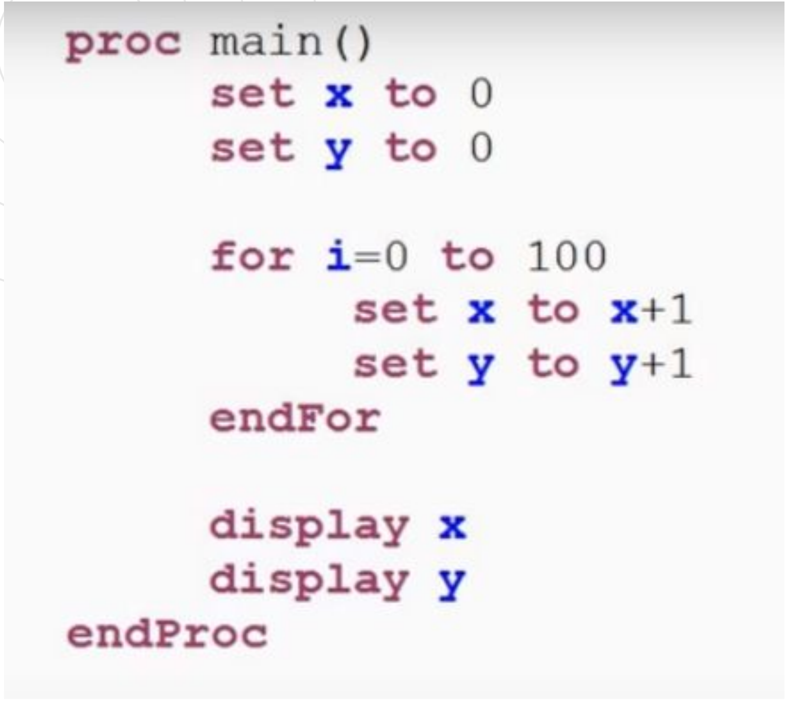
```
class Pager {
  gotoPrevPage() {
    // pointer to prev page
  }
  gotoNextPage() {
    // pointer to next page
  }
  gotoPageByPageNumber(pagerNumber: number) {
    // pointer to specific page
  }
}
```

```
class Printer {
  printPageInHTML(pageContent: any) {
    // your logic
  }
  printPageInJSON(pageContent: any) {
    // your logic
  }
  printPageInXML(pageContent: any) {
    // your logic
  }
  printPageUnformatted(pageContent: any) {
    // your logic
  }
}
```

The rules of functions

- Should be small
- Functions should do one thing
- They should do it well
- They should do it only
- Use Descriptive name
- Keep an eye on code duplications and side effects
- Function arguments (0 is the best, 3 is too many)

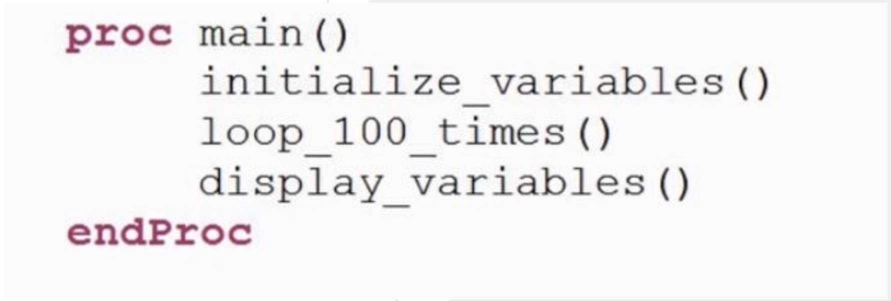
One level of abstraction per function



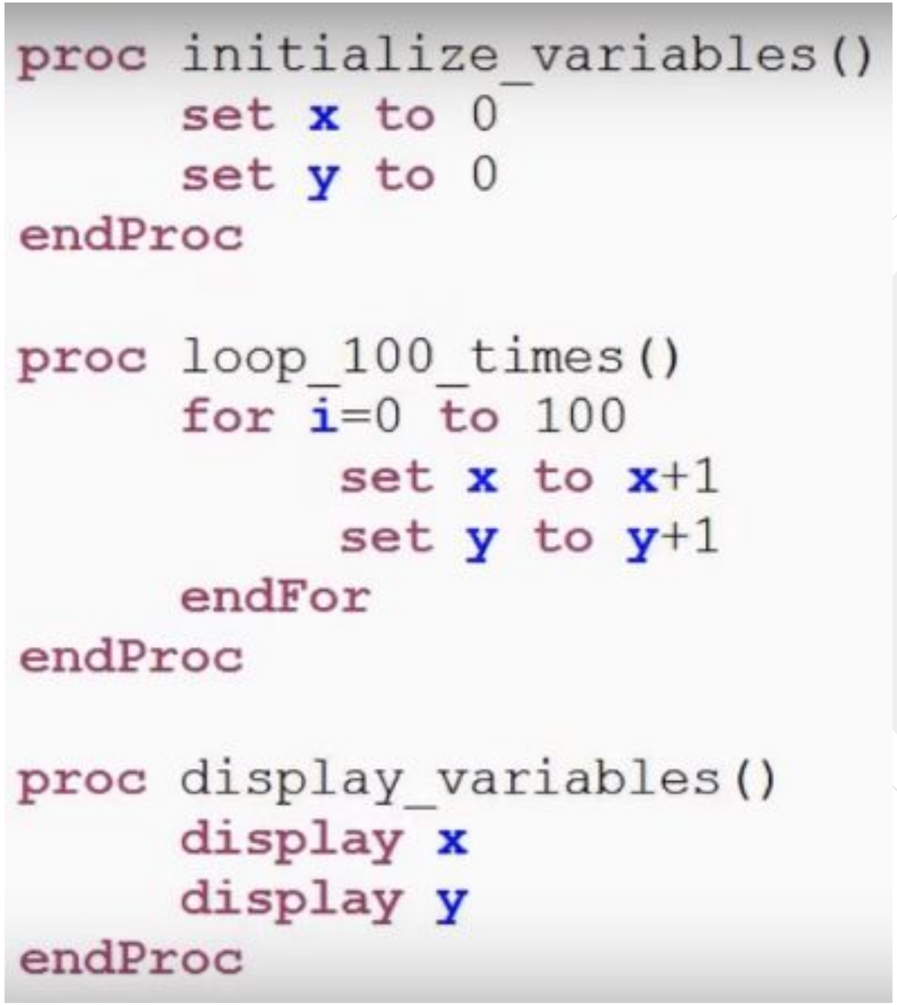
```
proc main()
      set x to 0
      set y to 0

      for i=0 to 100
            set x to x+1
            set y to y+1
      endFor

      display x
      display y
endProc
```



```
proc main()
      initialize_variables()
      loop_100_times()
      display_variables()
endProc
```



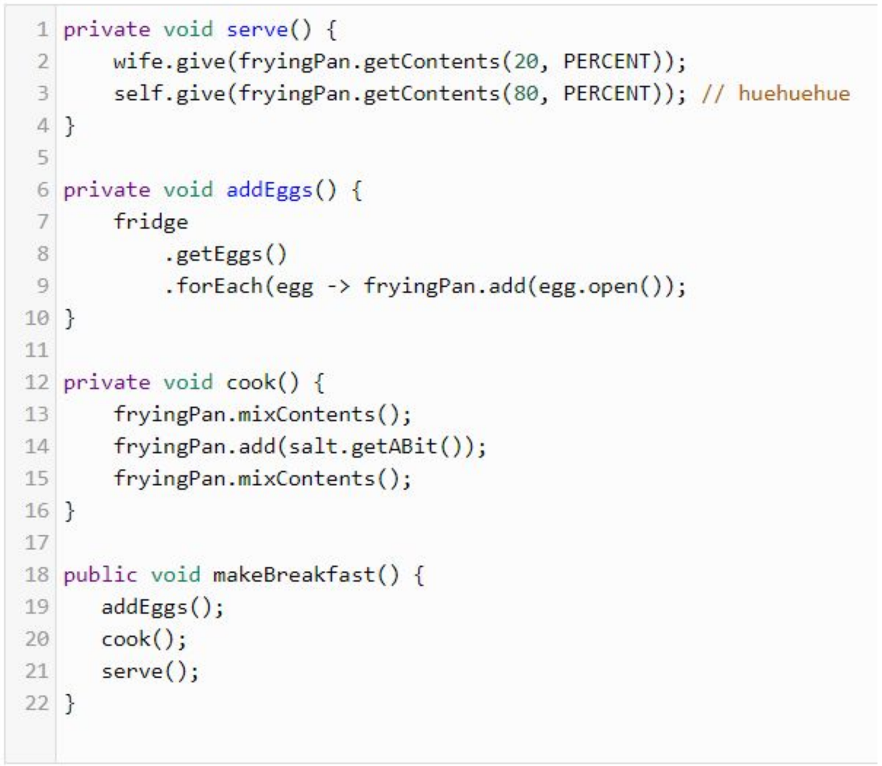
```
proc initialize_variables()
     set x to 0
     set y to 0
endProc

proc loop_100_times()
     for i=0 to 100
          set x to x+1
          set y to y+1
     endFor
endProc

proc display_variables()
     display x
     display y
endProc
```

The stepdown rule

Bad: Wrong Order



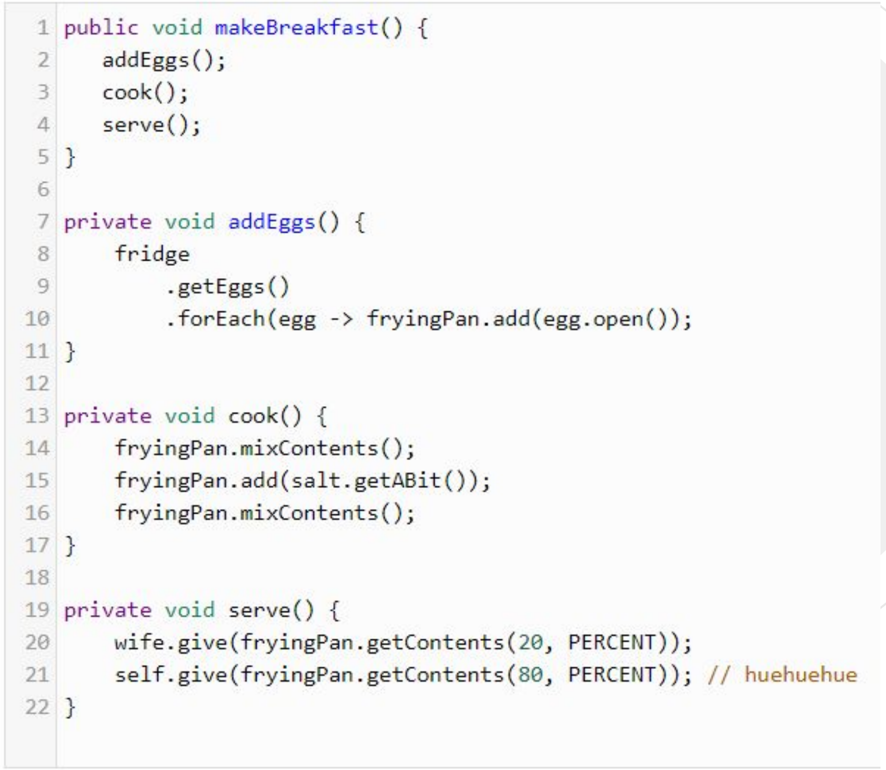
```java
private void serve() {
    wife.give(fryingPan.getContents(20, PERCENT));
    self.give(fryingPan.getContents(80, PERCENT)); // huehuehue
}

private void addEggs() {
    fridge
        .getEggs()
        .forEach(egg -> fryingPan.add(egg.open());
}

private void cook() {
    fryingPan.mixContents();
    fryingPan.add(salt.getABit());
    fryingPan.mixContents();
}

public void makeBreakfast() {
   addEggs();
   cook();
   serve();
}
```

Good



```java
public void makeBreakfast() {
   addEggs();
   cook();
   serve();
}

private void addEggs() {
    fridge
        .getEggs()
        .forEach(egg -> fryingPan.add(egg.open());
}

private void cook() {
    fryingPan.mixContents();
    fryingPan.add(salt.getABit());
    fryingPan.mixContents();
}

private void serve() {
    wife.give(fryingPan.getContents(20, PERCENT));
    self.give(fryingPan.getContents(80, PERCENT)); // huehuehue
}
```

Complexity

In short cyclomatic complexity is a number which indicates how many execution scenarios there might be inside your code.

```
function getName(firstName, lastName) {
 //cyclomatic complexity always starts from 1
 if (firstName && lastName) { //if operator, +1
   return firstName + ' ' + lastName;
 } else if (firstName) { // +1
   return firstName;
 } else if (lastName) { // +1
   return lastName;
 } else if (!firstName && !lastName) { // +1
   return 'stranger';
 }
 //total complexity is 5
}
```

```javascript
function getName(firstName, lastName) {
 //complexity starts from 1
 let name = '';
 if (firstName) { //if operator, +1
    name = firstName;
 }
 if (lastName) { // +1
    name += ' ' + lastName;
 }
```

Styleguide

- consistency
- make it more readable
- guideline
- follow best practises
- simplicity
- guideline for code reviews

Code smells

In computer programming, code smell is any symptom in the source code of a program that possibly indicates a deeper problem.

Types:

1. Purposeless conditions
2. Multiple return statements
3. This or That
4. Equality
5. Broken Promises

Links

https://www.codingblocks.net/podcast/clean-code-writing-meaningful-names/

https://samueleresca.net/2016/08/solid-principles-using-typescript/

https://webuniverse.io/cyclomatic-complexity-refactoring-tips/

https://github.com/airbnb/javascript

https://medium.freecodecamp.org/google-publishes-a-javascript-style-guide-here-are-some-key-lessons-1810b8ad050b

https://github.com/mohuk/js-code-smells