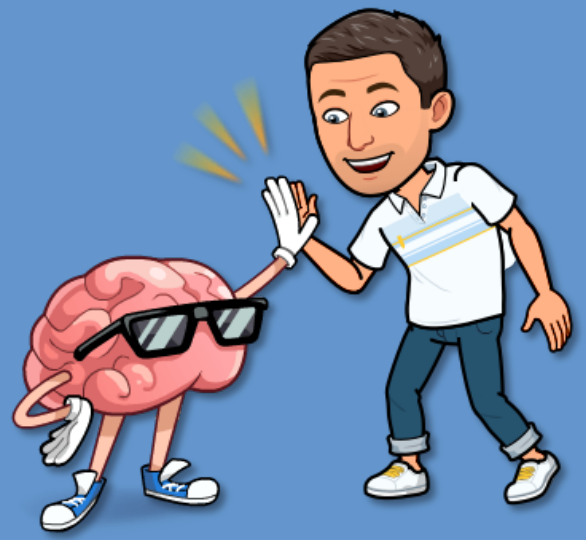
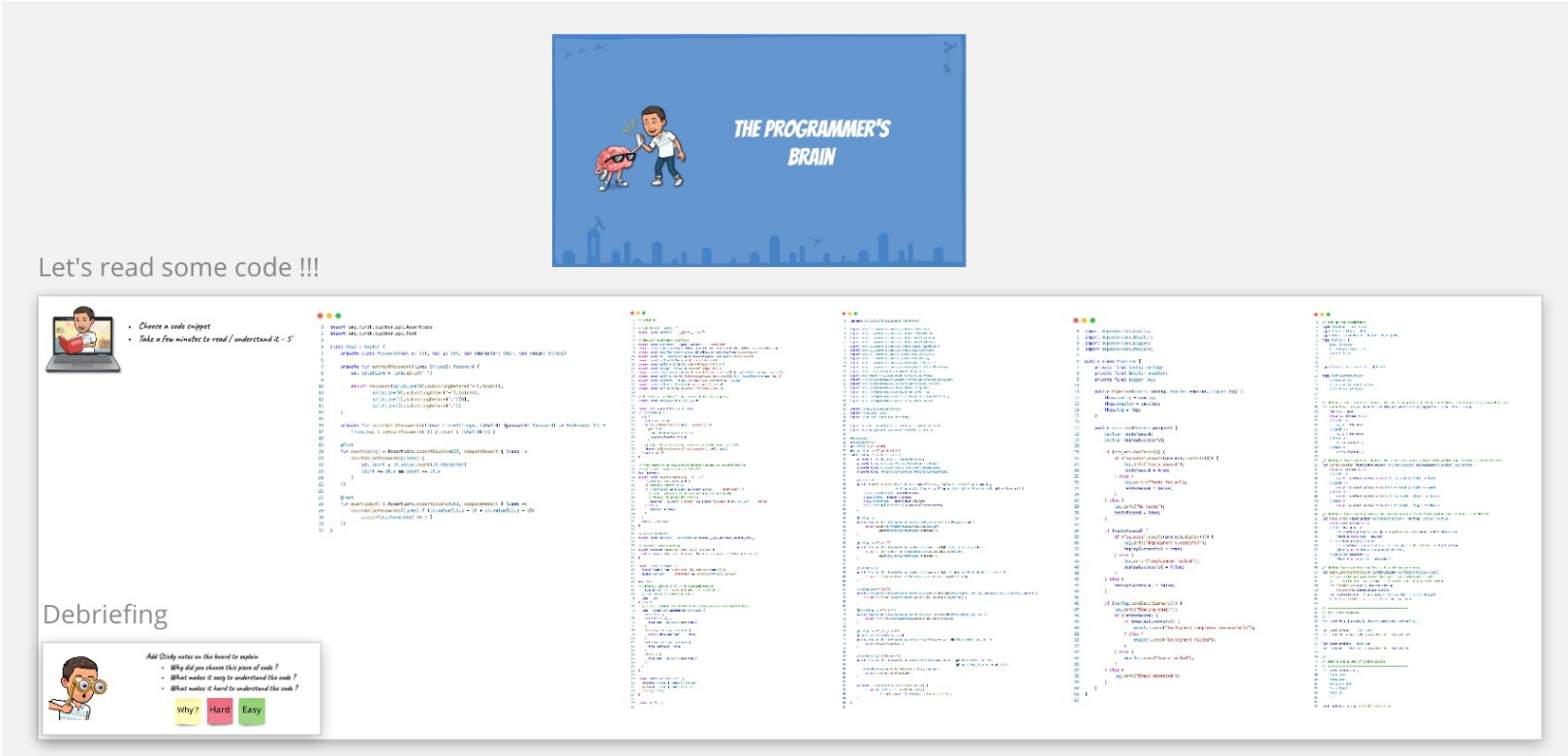


THE PROGRAMMER'S BRAIN

HOW TO READ CODE BETTER ?

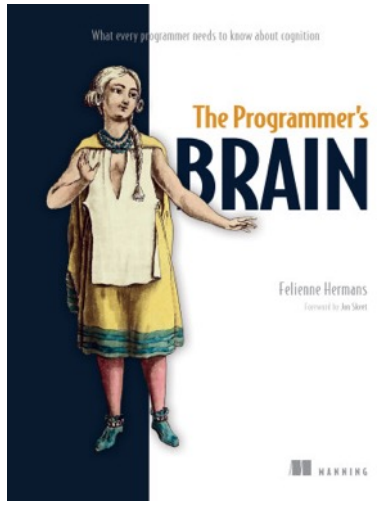
CODE READER

- Let's read some code
- Open the miro board



https://miro.com/app/board/o9J_l0DSaRQ=/

A WORD ON THE BOOK



Dr Felienne Hermans

- ✓ Associate professor at the Leiden Institute of Advanced Computer Science
- ✓ @felienne

Released on September 7, 2021

Learn how to optimize your brain's natural cognitive processes to

- Read code more easily
- Write code faster
- Pick up new languages in much less time

https://www.manning.com/books/the-programmers-brain

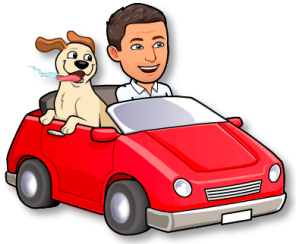
PLAN

How to read code better ?

Speed reading for code
Learn programming syntax more quickly
How to not forget things
Read complex code

On thinking about code

Reaching a deeper understanding of code



On writing code better

Get better at naming things
Avoiding bad code and cognitive load
Getting better at solving complex problems

On collaborating on Code

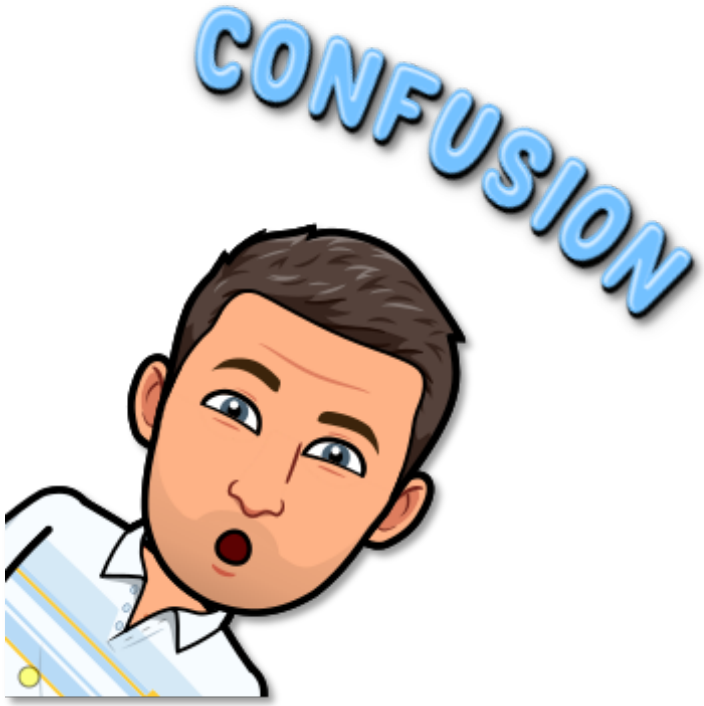
Getting better at Handling interruptions
How to onboard new developers

How to read code better ?

CONFUSION

Part of programming

- When you learn a new programming language, concept, or framework
- When reading unfamiliar code or code that you wrote a long time ago
- Whenever you start to work in a new business domain

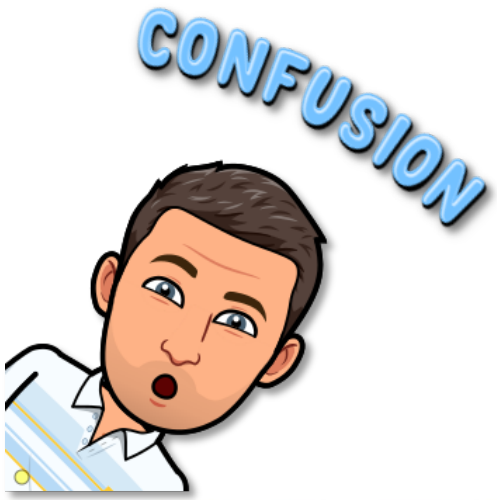


3 TYPES OF CONFUSION

APL program `2 2 2 2 2 T n`

What's this language ?
What T means ?

Lack of knowledge

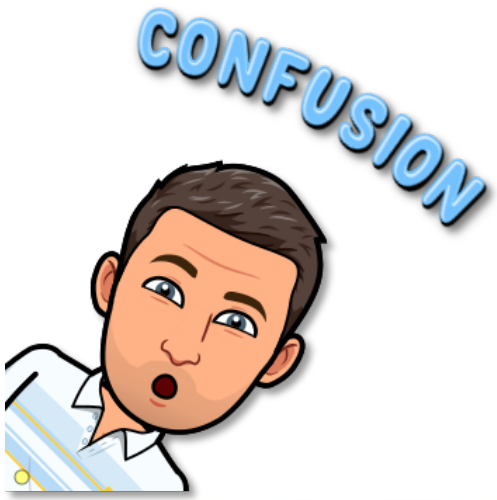


3 TYPES OF CONFUSION

```
public class BinaryCalculator {
    public static void mian(Int n) {
        System.out.println(Integer.toBinaryString(n));
    }
}
```

How exactly toBinaryString() works is not readily available
Needs to be found somewhere else in the code

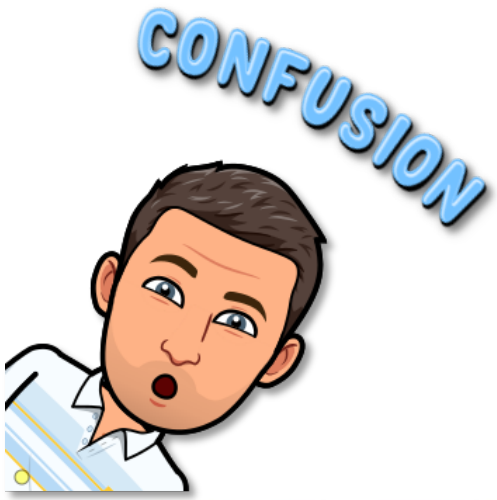
Lack of easy-to-access information



3 TYPES OF CONFUSION

```
LET N2 = ABS (INT (N))
LET B$ = ""
FOR N1 = N2 TO 0 STEP 0
    LET N2 = INT (N1 / 2)
    LET B$ = STR$ (N1 - N2 * 2) + B$
    LET N1 = N2
NEXT N1
PRINT B$
RETURN
```

We cannot oversee all the small steps that are being executed

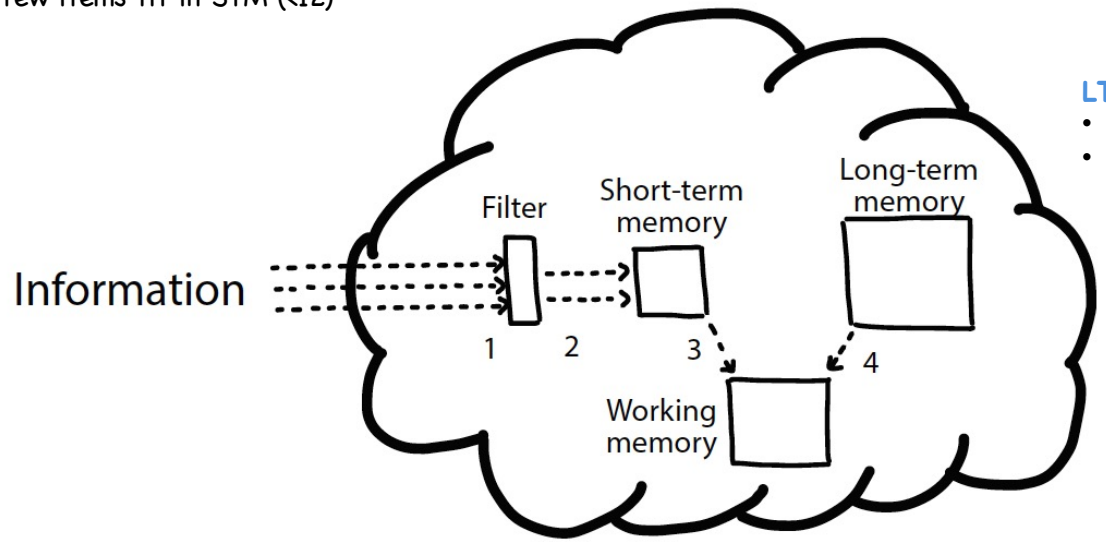


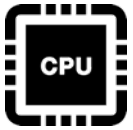
Lack of processing power in the brain

DIFFERENT COGNITIVE PROCESSES THAT AFFECT CODING

STM
- Used to briefly hold incoming information
- **RAM or a cache** that can be used to temporarily store values
- Just a few items fit in STM (<12)

LTM
- Can store your memories for a very long time
- **Hard drive** of your brain

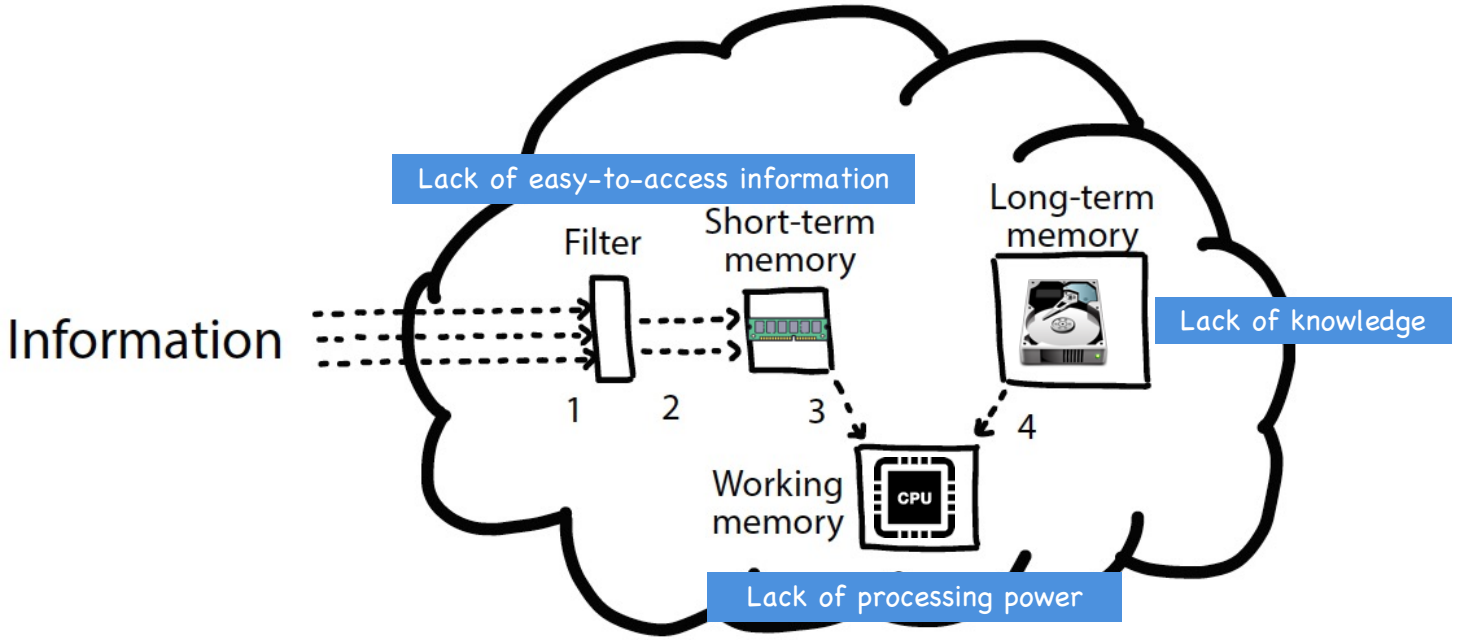




Working Memory
- The actual "thinking", happens in working memory
- **Processor** of the brain

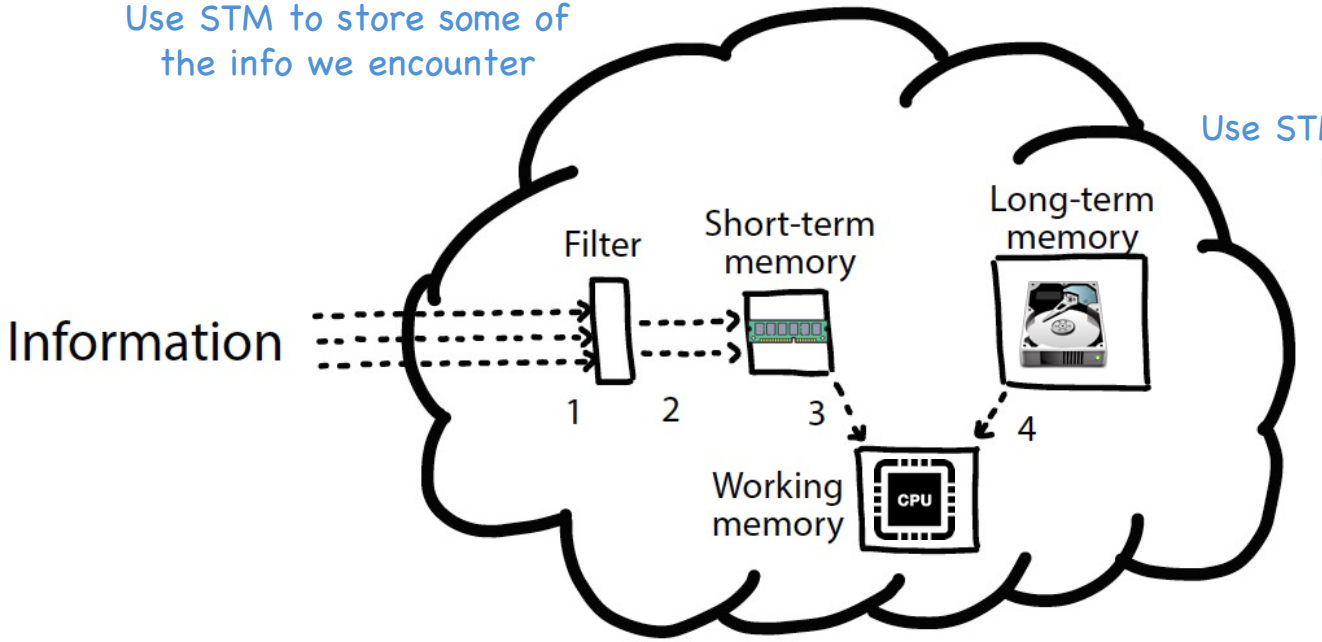
1. Information coming into your brain.
2. Information that proceeds into your STM
3. Information traveling from the STM into the working memory
4. Where it's combined with information from the LTM

CONFUSION AND COGNITIVE PROCESSES



WHAT HAPPENS WHEN WE READ CODE ?

Use STM to store some of the info we encounter



Use STM to retrieve knowledge

Keywords for example

Use working memory : trying to mentally execute the code / understand what is happening

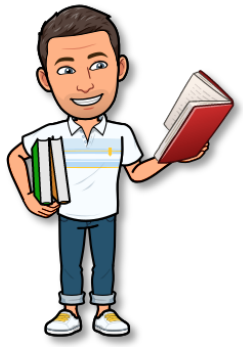
That process is called tracing

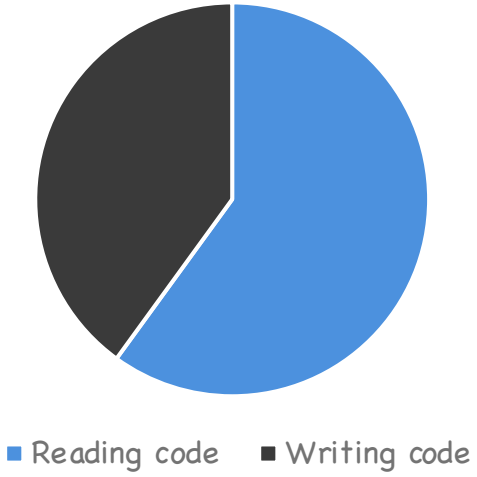
HOW TO READ CODE BETTER ?

SPEED READING FOR CODE

WHY READING SO IMPORTANT ?

Research indicates that almost **60% of programmers' time** is spent understanding code, rather than writing code





Reading code is done for a variety of reasons:

- add a feature
- find a bug
- build an understanding of a larger system

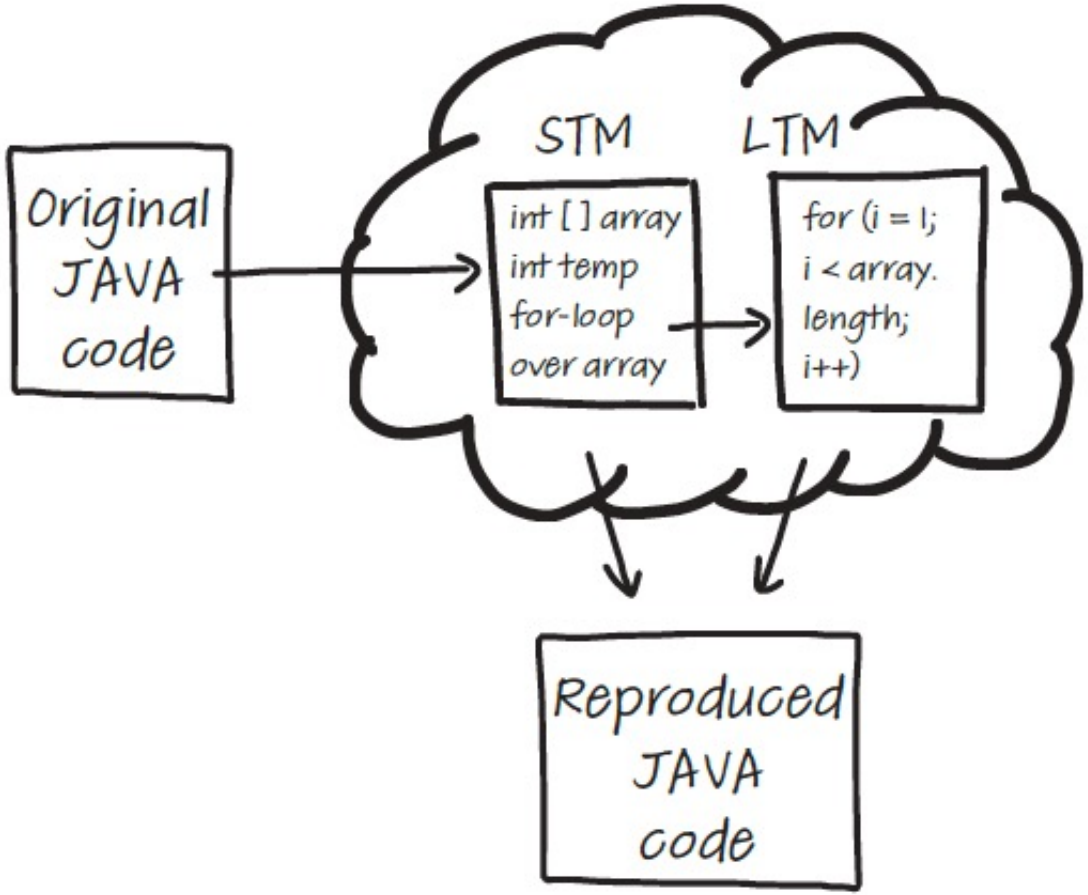
Improving how quickly we can read code (without losing accuracy)
-> **help us improve our programming skills substantially**

QUICKLY READING CODE - PART 1

- Look at the following Java program for 3 minutes
- Try to **reproduce it as best as you can**

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] array = {45, 12, 85, 32, 89, 39, 69, 44, 42, 1, 6, 8};
        int temp;
        for (int i = 1; i < array.length; i++) {
            for (int j = i; j > 0; j--) {
                if (array[j] < array[j - 1]) {
                    temp = array[j];
                    array[j] = array[j - 1];
                    array[j - 1] = temp;
                }
            }
        }
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
    }
}
```

WHAT JUST HAPPENED IN YOUR BRAINS



- Some parts of the code are stored in STM
  - Variable names / their values
- Other parts of the code :
  - Syntax of a for-loop for example
  - Use knowledge stored in our LTM

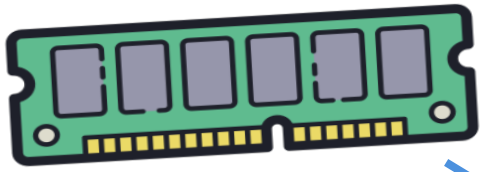
QUICKLY READING CODE - PART 2

- Have a second look at your reproduced code
- **Annotate** which parts of the code you think came from your short-term memory directly and which parts were retrieved from long-term memory
- Compare with someone else

Information retrieved from your LTM depends on what you have stored there :
Less experienced in Java is likely to retrieve a lot less from their LTM

WHY IS READING UNFAMILIAR CODE HARD?

Short Term Memory



Time limitation
Cannot hold information for more than
30 seconds

Size limitation
Just a few slots available for information
7 +/- 2

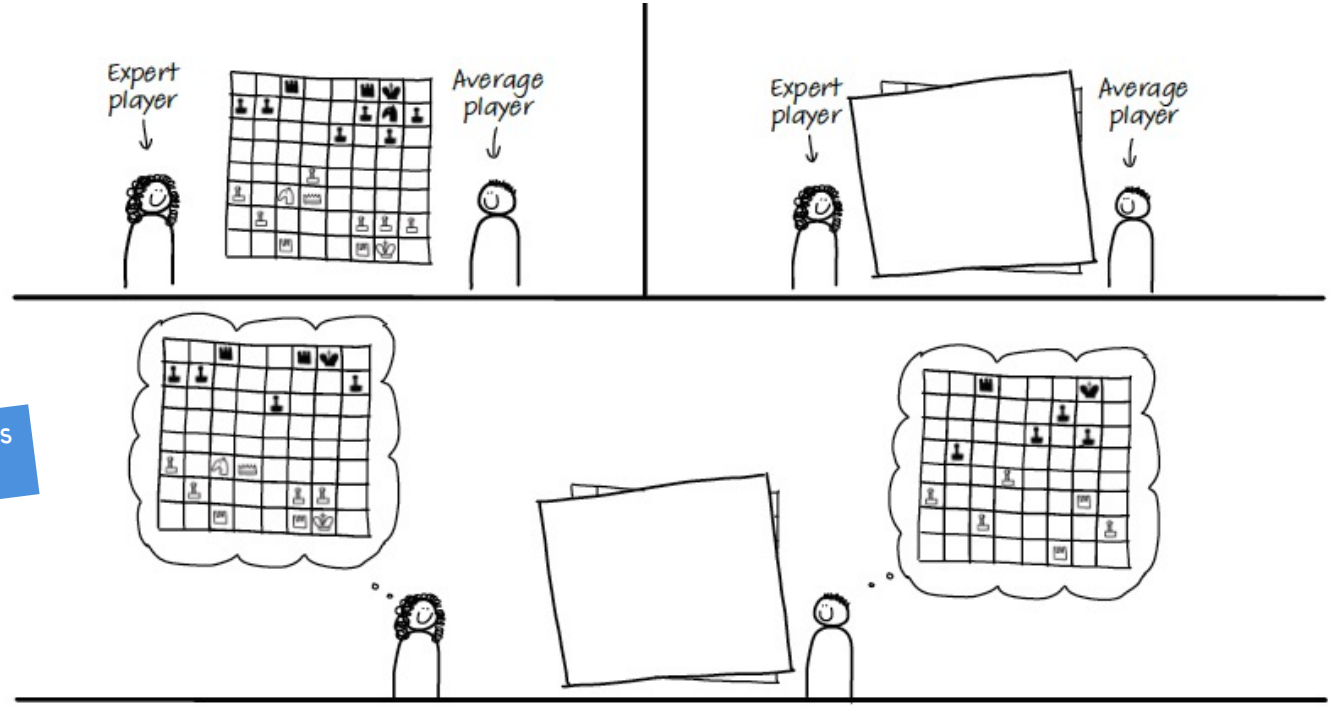
Some research indicates that its capacity could be smaller : just **2 to 6 things**

A memory issue

OVERCOMING SIZE LIMITS IN YOUR MEMORY

De Groot's first chess experiment

Experts and average chess players were asked to remember a chess setup

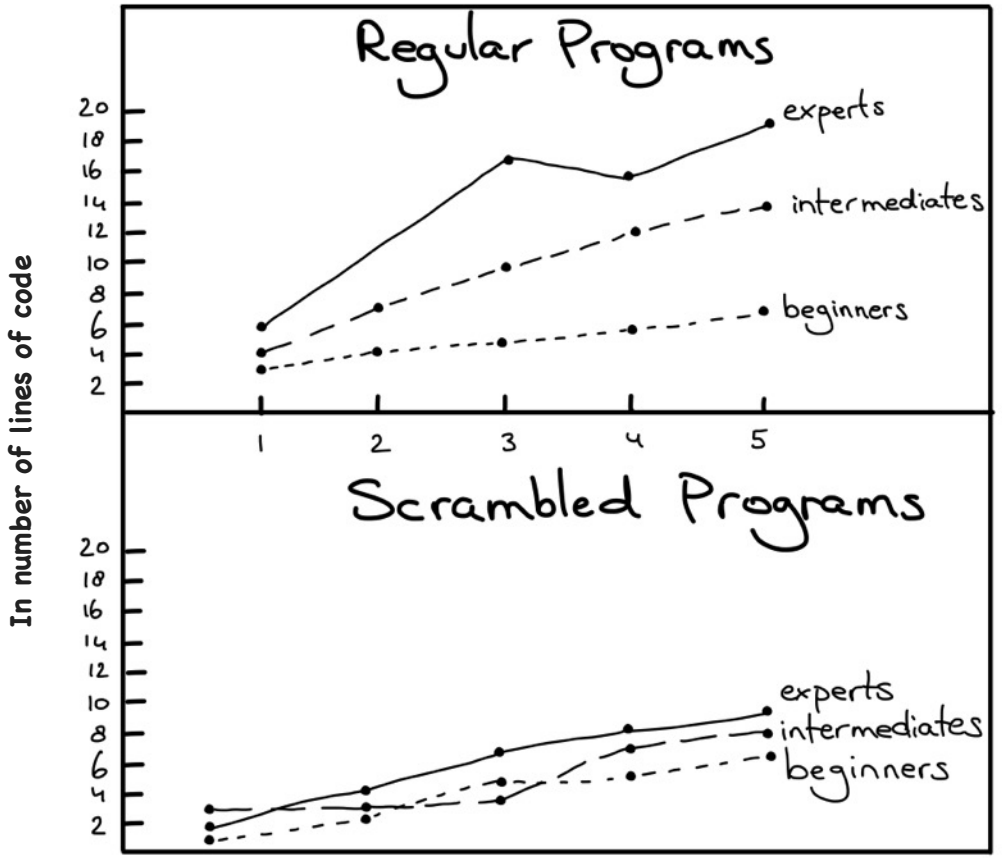


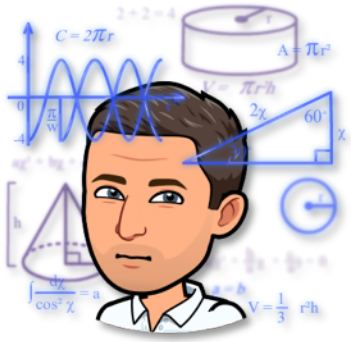
Experts grouped info in logical ways -> chunks
Fit into only 1 slot in STM

Expert players were able to recall more pieces than average players

DE GROOT'S EXPERIMENTS ON PROGRAMMERS

In 1981 Katherine McKeithen, repeat de Groot's experiments on programmers :





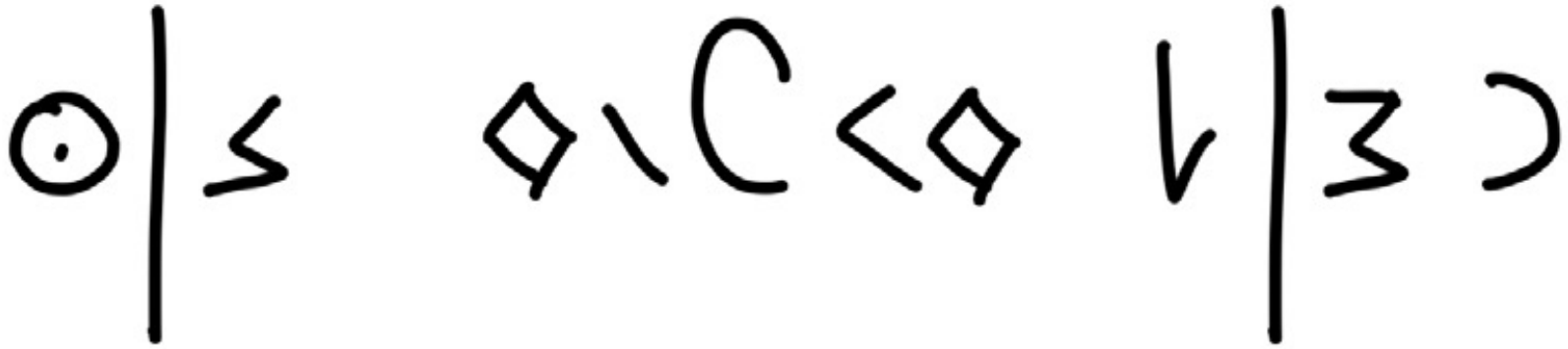
Main takeaway : beginners will be able to process a lot less code than experts

EXPERIENCE CHUNKING

EXPERIENCE CHUNKING 1

Look at this sentence for five seconds and try to remember it as best as you can:

Reproduce it on paper



EXPERIENCE CHUNKING 2

Look at this sentence for five seconds and try to remember it as best as you can:
Reproduce it on paper

abk mrtpi gbar

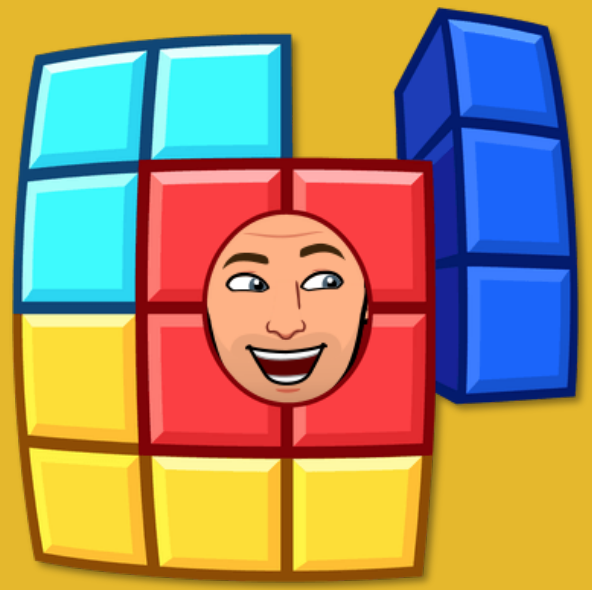
- Easier than the first one!
- Because consists of letters that you recognize
- 2 sentences were the same length:
  3 words, 12 characters, 9 different characters

EXPERIENCE CHUNKING 3

Look at this sentence for five seconds and try to remember it as best as you can:
Reproduce it on paper

cat loves cake

- Here you can chunk the characters into words.
- You can then remember just three chunks: “cat,” “loves,” and “cake.”
  Like the chess experts



CHUNKING IN CODE

How to write "chunkable" code

The more information you have stored about a specific topic the easier it is to effectively divide information into chunks.

- **Use Design Patterns**

  Helpful for performing maintenance tasks
  when the programmers knew that the pattern was present in the code

- **Write comments** (chunk larger pieces of code)

  High-level comments like "this function prints a given binary tree in order"
  Low-level comments such as "increment i (by one)" : create a burden on the chunking process

How to write "chunkable" code

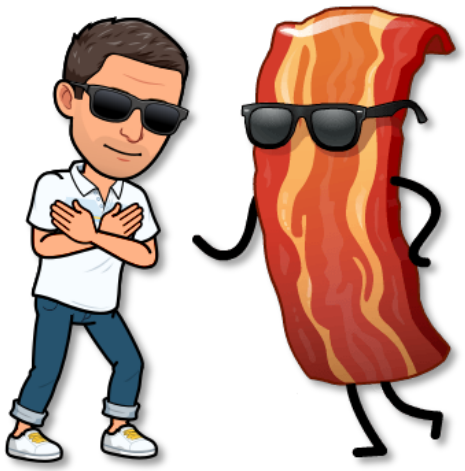
Leave Beacons

- Parts of a program that help a programmer to understand what the code does
  A line of code that your eye falls on which "Now I see"

- Act as a trigger for programmers to confirm or refute hypotheses about the source

- **Simple beacons** : self-explaining syntactic code elements
  Meaningful variable names
  Operators such as +, >,
  && / structural statements if, else, and

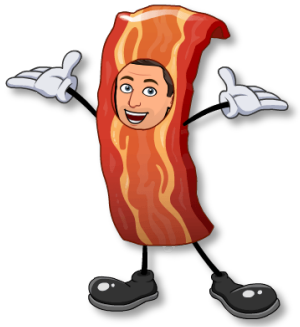
- **Compound beacons** : larger code structures comprised of simple beacons



BEACONS EXAMPLE

```
# A class that represents a node in a tree
class Node:
    def __init__(self, key):
    self.left = None
    self.right = None
    self.val = key

# A function to do in-order tree traversal
def print_in_order(root):
    if root:
    # First recur on left child
    print_in_order(root.left)
    # then print the data of node
    print(root.val),
    # now recur on right child
    print_in_order(root.right)
    print("Contents of the tree are")
    print_in_order(tree)
```

- Comments using the word "tree"
- Variables called root and tree
- Fields called left and right
- String contents in the code that concern trees



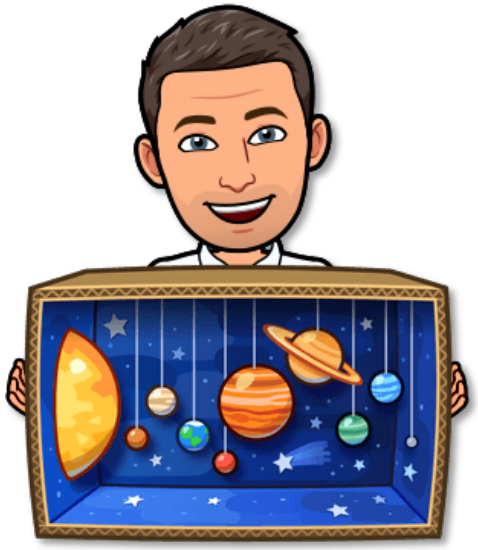
HOW TO READ CODE BETTER ?

LEARN PROGRAMMING SYNTAX MORE QUICKLY

WHAT WE LEARNED

What you already know impacts to a large extent how efficiently you can read and understand code

- The more concepts, data structures, and syntax you know
- The more code you can easily chunk, and thus remember and process



FLASHCARDS

A great way to learn anything, including syntax, fast
Simply paper cards or Post-its of which you use both sides

One side has a prompt on it

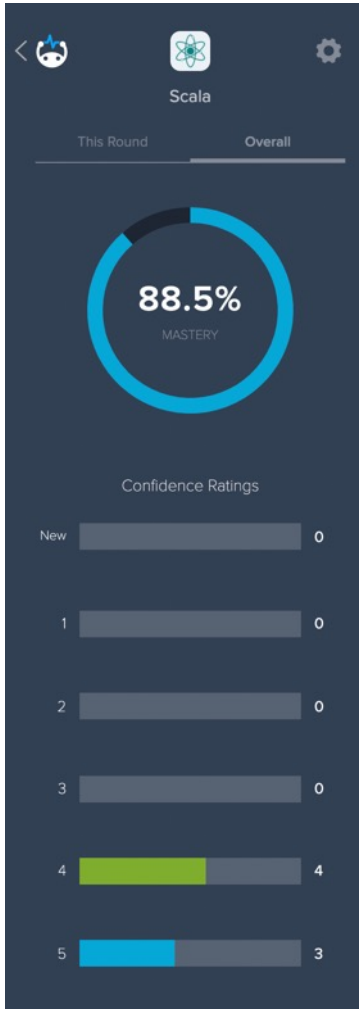
The thing that you want to learn

The other side has the corresponding knowledge on it

FLASHCARDS

When to use them ?

- Use them often to practice
- Use Apps like **Brainscape** : allow you to create your own digital flashcards
  They will remind you when to practice again
- After a few weeks, your syntactic vocabulary will have increased

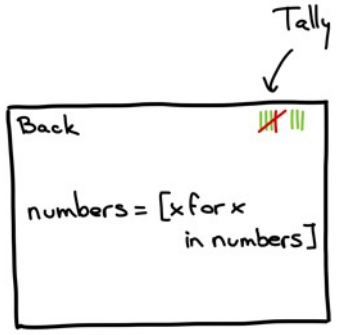


Expanding the set of flashcards

- When you're learning a new programming language, framework, or library
- When you're about to Google a certain concept

Thinning the set of Flashcards

- Keep a little tally on each card of your right and wrong answers
- Demonstrate what knowledge is already reliably stored in your long-term memory



HOW TO REMEMBER SYNTAX LONGER ?

2 techniques **to strengthen memories** :

- **Retrieval practice** : actively trying to remember something
  Retrieval is a more formal word for remembering
- **Elaboration** : actively thinking / connecting new knowledge to existing memories

→ **Read / hide code exercises**

Thinking about what you want to remember :

Relating it to existing memories

- Making the new memories fit into schemata already stored in your LTM

Using elaboration to learn new programming concepts

For example, you might try thinking of related concepts in other programming languages

- Thinking of alternative concepts in Python or other programming languages
- Or thinking of how this concept relates to other paradigms

CREATE YOUR OWN FLASHCARDS

FLASHCARDS

- Think of the **top 10 programming concepts** you always have trouble remembering
- Make a set of flashcards for each of the concepts and try using them

You can also do this collaboratively in a group or team, where *you might discover that you are not the only one that struggles with certain concepts.*

FLASHCARDS - EXAMPLE

Pattern matching	val x: Int = Random.nextInt(10)
	
	x match {
	case 0 => "zero"
	case 1 => "one"
	case 2 => "two"
	case _ => "other"
	}

Extension method	case class Circle(x: Double, y: Double, radius: Double) extension (c: Circle) def circumference: Double = c.radius * math.Pi * 2

Becomes a routine

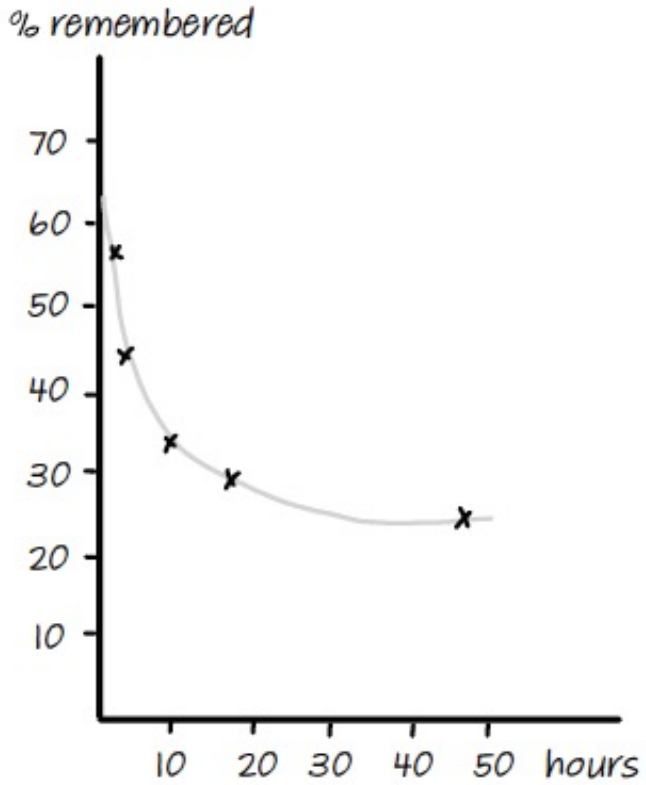
How to read code better ?

How to not forget things

HOW TO NOT FORGET THINGS

The big problem with our **Long-Term Memory** :

We cannot remember things for a long time without extra practice



After 2 days, just 25% of the knowledge remains in our LTM

HOW TO NOT FORGET THINGS - SPACED REPETITION

The best way that science knows to prevent forgetting is to practice regularly. Each repetition strengthens your memory.

You remember the longest if you study over a longer period

- In contrast with formal education, where we try to cram all the knowledge into one semester
- **Revisiting your set of flashcards once a month**

  Enough to help your memory in the long run / relatively doable!

HOW TO READ CODE BETTER ?

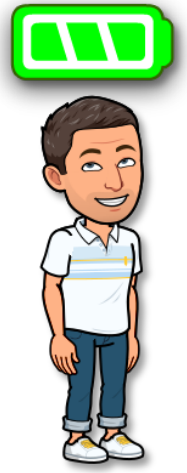
READ COMPLEX CODE

WORKING MEMORY

Working memory : short-term memory applied to a problem

What happens in the working memory when we read code?

- Like STM the working memory is only capable of processing between 2 and 6 things at a time
- In the context of working memory, this capacity is known as the ***cognitive load***.



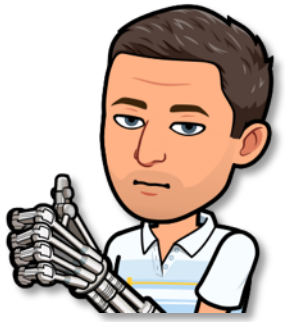
COGNITIVE LOAD

Load Type	Description	
Intrinsic load	How complex the problem is in itself	
Extraneous load	What outside distractions add to the problem	Accidental complexity
Germane load	Cognitive load created by having to store your thought to LTM	

TECHNIQUE 1 - REFACTORING CODE TO REDUCE COGNITIVE LOAD

In many cases these refactoring are temporary

- Only meant to allow you understand the code
- Can be rolled back once your understanding is solidified



Refactoring example - replace unfamiliar language constructs

Simply rewrite the code to use a regular structure temporarily

```
Optional<Product> product = productList.stream()
        .filter(p -> p.getId() == id)
        .findFirst();
```

If new to lambdas

Remove extraneous cognitive load

```
public static class Toetsie implements Predicate <Product>{
    private int id;
    Toetsie(int id){
        this.id = id
    }
    boolean test(Product p){
        return p.getID() == this.id;
    }
}

Optional<Product> product = productList.stream().
        filter(new Toetsie(id)).
        findFirst();
```

What is easy to read depends on your prior knowledge

No shame in helping yourself understand code by translating it to a more familiar form

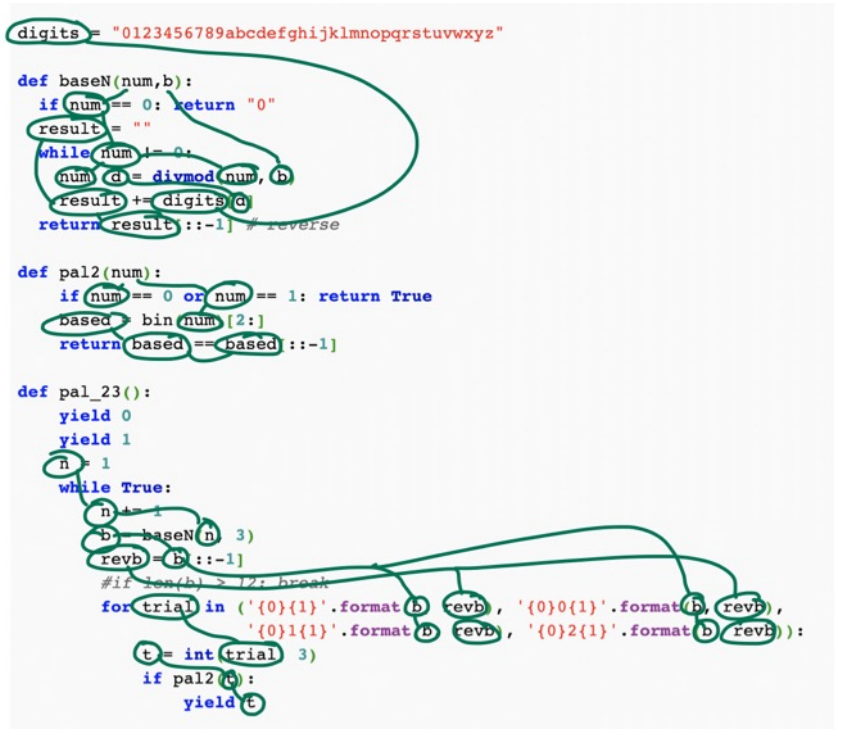
TECHNIQUE 2- MARKING DEPENDENCIES / DEPENDENCY GRAPH

- Print out the code, or convert it to a PDF
- Open it on a tablet so you can make annotations digitally
- Circle all the variables

```
digits = "0123456789abcdefghijklmnopqrstuvwxyz"

def baseN(num,b):
  if num == 0: return "0"
  result = ""
  while num != 0:
    num, d = divmod(num, b)
    result += digits[d]
  return result[::-1] # reverse

def pal2(num):
    if num == 0 or num == 1: return True
    based = bin(num)[2:]
    return based == based[::-1]

def pal_23():
    yield 0
    yield 1
    n = 1
    while True:
        n += 1
        b = baseN(n, 3)
        revb = b[::-1]
        #if len(b) > 12: break
        for trial in ('{0}{1}'.format(b, revb), '{0}0{1}'.format(b, revb),
                      '{0}1{1}'.format(b, revb), '{0}2{1}'.format(b, revb)):
            t = int(trial, 3)
            if pal2(t):
                yield t
```

Link similar variables

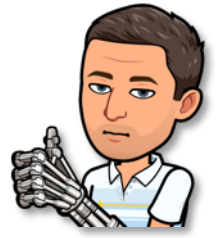
- Draw lines between occurrences of the same variable
- Helps you to understand where data is used in the program



Now have a reference that you can refer to for information about the code's structure

TECHNIQUE 3- MARKING DEPENDENCIES / STATE TABLE

A state table focuses on the values of variables rather than the structure of the code.
It has columns for each variable and lines for each step in the code.



	N	N2	B$	N1
init	7	7	—	7
loop 1		3	0	3
loop2				

How to ?

1. Make a list of all the variables
2. Create a table : give each variable its own column
3. Add 1 row for each distinct part of the execution of the code
4. Execute each part of the code : write down the value each variable has afterward in the correct row and column.

Combining State Tables and Dependency Graphs

COMBINING STATE TABLES AND DEPENDENCY GRAPHS

```java
public class Calculations {
    public static void main(String[] args) {
        char[] chars = {'a', 'b', 'c', 'd'};
        // looking for bba
        calculate(chars, 3, i -> i[0] == 1 && i[1] == 1 && i[2] == 0);
    }

    static void calculate(char[] a, int k, Predicate<int[]> decider) {
        int n = a.length;
        if (k < 1 || k > n)
            throw new IllegalArgumentException("Forbidden");
        int[] indexes = new int[n];
        int total = (int) Math.pow(n, k);
        while (total-- > 0) {
            for (int i = 0; i < n - (n - k); i++)
                System.out.print(a[indexes[i]]);
            System.out.println();
            if (decider.test(indexes))
                break;
            for (int i = 0; i < n; i++) {
                if (indexes[i] >= n - 1) {
                    indexes[i] = 0;
                } else {
                    indexes[i]++;
                    break;
                }
            }
        }
    }
```

Following the steps outlined in the previous sections :

create both a dependency graph and a state table for this Java program

WHAT DO WE DO NOW ?

I learned a lot, but it doesn't apply to me

I learned a lot and it applies to me

I didn't learn much and it doesn't apply to me

I can apply what little I have learned

Level of knowledge acquired

Applicability of this knowledge

THE PROGRAMMER'S BRAIN

by Felienne Hermans

Code

Short Term Memory (STM)
- RAM or a cache of our brain
- Just a few items fit in STM (<12)
- Store some of the info we encounter

Info traveling from the STM into the working memory

Long Term Memory (LTM)
- Hard drive of our brain
- Can store memories for a very long time
- Retrieve related knowledge (ex : keywords)

Combined with information from the LTM

Working Memory (WM)
- Processor of our brain
- The actual "thinking" happens here
- Mentally execute the code aka TRACING

When we read code

60% of our time

Why is reading unfamiliar code hard?

LIMITED

Short term memory
- Time : 30 seconds
- Size : 7 +/-2 things

How to read code better ?

"More concepts, data structures and syntax you know the more code you can easily chunk, and thus remember and process"

Learn programming syntax

Read / Hide / Write code exercises

Use Flashcards
- Front : prompt
- Back : corresponding knowledge

Remember syntax longer
- Retrieval : trying to remember something
- Elaboration : connecting new knowledge to existing memories

How to not forget things ?

"We cannot remember things for a long time without extra practice"

DON'T FORGET

After 2 days, just 25% of the knowledge remains in our LTM

Spaced repetition
- Practice regularly
- Best way to prevent forgetting

Revisit your Flashcards
- Once a month
- Each repetition strengthens your memory

Write CHUNKABLE Code

"Experts group info in logical ways a.k.a chunks"

- Write COMMENTS — High-level comments help to chunk larger pieces of code
- Use Design PATTERNS — Help to process code faster
- Leave BEACONS — var names, operators (+, >), if, else, comments,...

Read complex code easier

Reduce cognitive load

Refactoring code
Ex : replace unfamiliar language constructs

"Our ability to learn a natural language can be a predictor of your ability to learn to program."

Dependency graph
- Circle variables
- Draw lines between occurrences

State table
- Focuses on the values of variables
- 1 column / variable
- 1 line / step in the code

Cognitive load
- Capacity of our Working Memory
- Capacity : 2 to 6 "things"

Roles of variables (Sajaniemi's framework)

- Fixed value : value does not change after initialization
- Stepper : variable stepping through a list of values
- Flag : has happened or is the case
- Walker : traverses a data structure (search index)
- Most-recent holder : holds the latest value encountered
- Most-wanted holder : holds the best value found so far
- Gatherer : collects data and aggregates it
- Container : holds multiple elements
- Follower : keep track of a previous value
- Organizer : transformed variable
- Temporary : used only briefly

"Understanding what types of information variables hold is key to being able to reason about and make changes to code."

"Many similarities between reading code and reading natural language"

Text comprehension strategies applied to code

Activating
Actively thinking about code elements help our WM to find relevant information stored in the LTM

Monitoring
- Keep track of what we are reading and our understanding
- ex : ticking the lines

Inferring
Inferring the meaning of variable names

Questioning
- Asking ourself questions while reading code
- Help us understand the code's goals and functionality
- ex : What are the 5 most central concepts of the code?

Determining importance
Identify which parts of the code are likely to have the most influence on the program's execution

Visualizing
List all operations in which variables are involved (dependency graph, state table,...)

Summarizing
- Write a summary of code in natural language
- Help us gain a deeper understanding of what's happening in that code

Goal of the code: what is the code trying to achieve?
Most important lines of code
Most relevant domain concepts
Most relevant programming constructs
...

Write better code

Search....

Avoid

Clear names help our LTM
LTM searches for related informations

Abbreviation
Check Hofmeister research

Snake Case -> use camel Case
camelCase leads to higher accuracy

Avoid Arnaoudova's linguistic anti-patterns

- Methods that do more than they say
- Methods that do the opposite than they say
- Identifiers whose name says that they contain more than what the entity contains
- Methods that say more than they do
- Identifiers whose name says that they contain less than what the entity contains
- Identifiers whose name says the opposite than the entity contains

LTM can store different types of memory

Memories

Their occurrence in 7 open-source projects :
- 11% of setters also return a value
- 2.5% of methods : method name + comment = opposite descriptions
- 64% of identifiers starting with "is" turned out not to be Boolean

Procedural / Implicit
- How to do something
- ex : How to run a bike

Declarative / Explicit
- Memories we are explicitly aware of
- Facts we can remember

"Experts heavily rely on episodic memory when solving problems / rely on solutions that have previously worked for similar problems."

Episodic
- Memories of experience
- ex : meeting our wife / husband

Semantic
- Memories for meanings / concepts / facts
- ex : 10 x 10 = 100

Getting better at solving complex problems

Automatization
create implicit memories

"Set some time aside every day to practice and continue until you can consistently perform the tasks without any effort"

Deliberate practice to improve skills
- Repeat a lot
- It frees up cognitive load for larger problems
- ex : deliberately type 100 for loops when struggling with it

Deliberate practice : requires focused attention and is conducted with the specific goal of improving performance.

Study worked examples
create episodic memories

Worked examples : something like a recipe which describes in detail the steps that are needed to solve the problem.

Code reading club
- Exchange code / explanation
- Learn from each other

Read books / blog post
- About source code

Explore github
- Choose repositories (domain knowledge)
- Focus on the programming itself

Better handle interruptions

20 % of developers time on interrupts

Store mental model
- Apart from the code
- Comments : excellent location to leave it
- Warm-up period in comprehension activities

Prepare for it

Help your "Prospective memory"
- Put TODO comments in the part of the code
- Remind you to complete / improve part of the code

Prospective memory : memory of remembering to do something in the future. (related to planning / problem solving)

15' to start editing code
after an interruption

Label subgoals
- Write down small steps of a problem
- Use mind maps for example

On-boarding process

Typical

High cognitive load

Senior dev — dev throws information → Newcomer

Explain only relevant informations

Separate → Domain learning / Exploring code

Support the LTM of the newcomer

Exploration
- Browse the codebase
- Get a general sense of the codebase

Searching
ex : find a class that implements a certain interface

Transcription
- Give the newcomer a clear plan
- Implement it

Limit tasks to ONE programming activity

Comprehension
Understand aspects of the code
ex : summarize a specific method in natural language

Incrementation
- Add a feature to an existing class
- Creation of the plan for the feature.

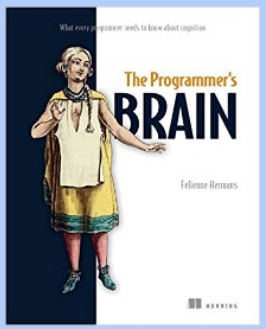
Start with it : read code together

#sharingiscaring

by Yoan THIRION @yot88

https://bit.ly/2WMtn02

How to Read Complex Code



Dr Felienne Hermans

- ✓ Associate professor at the Leiden Institute of Advanced Computer Science at Leiden University
- ✓ Author of "The Programmer's Brain"
- ✓ @felienne

Live 5 October 12:30 (UTC+2)

> Software Craftsmanship Luxembourg

Free registration on meetup.com