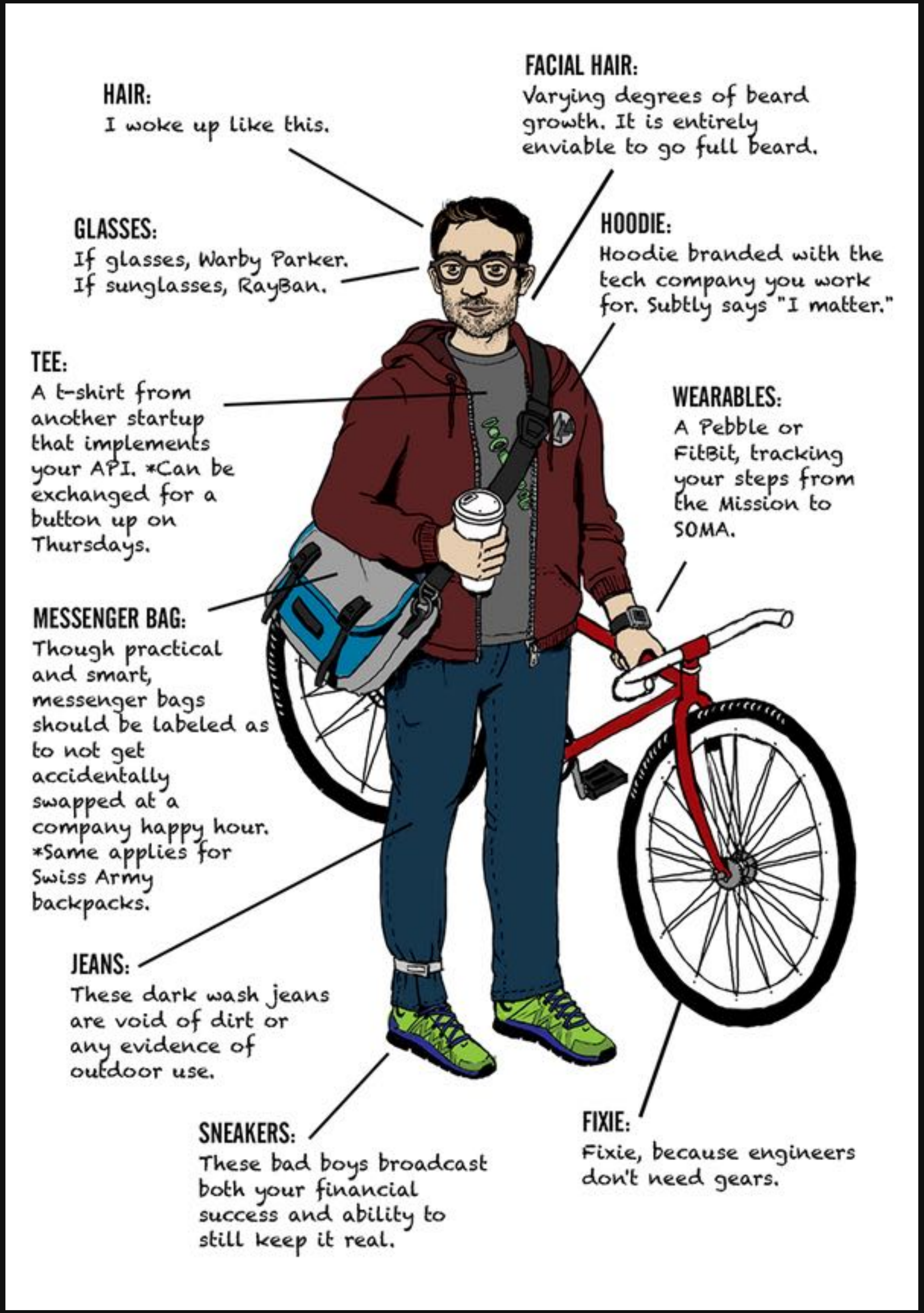
Coding Principles and Style

Things you should do but probably don’t
(and probably won’t)

What is code style?


HAIR:
I woke up like this.
FACIAL HAIR:
Varying degrees of beard growth. It is entirely enviable to go full beard.
GLASSES:
If glasses, Warby Parker. If sunglasses, RayBan.
HOODIE:
Hoodie branded with the tech company you work for. Subtly says "I matter."
TEE:
A t-shirt from another startup that implements your API. *Can be exchanged for a button up on Thursdays.
WEARABLES:
A Pebble or FitBit, tracking your steps from the Mission to SOMA.
MESSENGER BAG:
Though practical and smart, messenger bags should be labeled as to not get accidentally swapped at a company happy hour. *Same applies for Swiss Army backpacks.
JEANS:
These dark wash jeans are void of dirt or any evidence of outdoor use.
SNEAKERS:
These bad boys broadcast both your financial success and ability to still keep it real.
FIXIE:
Fixie, because engineers don't need gears.

What is code style?

- The appearance: good, bad, ugly
- Formatting
- Layout
- Organization
- Conventions
- “Grammar”

White Space

- Indentation
- Alignment
- New lines
- Empty lines
- Spaces
- Spaces vs Tabs

```c
while (1) {
  static struct option long_options[] =
    {
      {"verbose",     no_argument,        0, 'v'},
      {"algorithm",   required_argument,  0, 'a'},
      {"help",        no_argument,        0, 'h'},
      {0,             0,                  0, 0}
    };

  c = getopt_long (argc, argv, "vta:b", long_options, &option_index);

  if (c == -1) {
    break;
  }

  switch (c) {
    case 'v':
      verbose_flag = 1;
      break;
    case 'a':
      algorithm = optarg;
      break;
    case 'h':
      PrintHelp();
      break;
    default:
      abort();
  }
}
```

Capitalization

- All caps
- No caps
- First letter
- CamelCase
- camelCase

```
DROP TABLE IF EXISTS books;
CREATE TABLE books(
   number INT,
   title TEXT,
   isbn TEXT,
   publicationDate DATE,
   numPages INT,
   PRIMARY KEY( number )
);
```

Naming Conventions

- Meaningful names
- Short names
- Long names
- Single letters
- Nonsense

```
int Get_A_Random_Number() {

  randy = new Random();

  for( i=0; i<0; i++ ) {
    temp = randy.next();
    sum = temp + sum;
  }

  x = 10;

  randomNumber = sum + x;

  return randomNumber;
}
```

Comments

- Many different ways to write comments
- Block style
- Before line
- Inline
- Afterline

```java
/**
 * Function chooses a random move for the computer player
 * @param
 * @return the board position chosen by the computer
 */
public int computerMove() {

  //make a random generator
    Random r = new Random();

    while(true) {
        int randNum = r.nextInt(9); //get a random number

        if (board[randNum] == 0) {
            board[randNum] = 2;
            return randNum;
            //return the random number if valid
        }
    }
}
```

What is correct?

- Style is personal
- As long as its clear and readable
- If nothing else, be consistent
- Please yourself, please the audience
- Whatever you do is correct and everyone else is wrong

Style Guides

- A guide to proper style
- Nearly every language has one(or multiple)
- Written by language creators, enthusiasts, companies etc
- May be required to adhere if part of larger project or collaboration

C++ Horizontal Whitespace According to Google

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
// Loops and conditions may have spaces inside parentheses, but this
// is rare.  Be consistent.
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// For loops always have a space after the semicolon.  They may have a space
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
  ...

// Range-based for loops always have a space before and after the colon.
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:         // No space before colon in a switch case.
    ...
  case 2: break;  // Use a space after a colon if there's code after it.
```

Coding Principles

Coding Principles

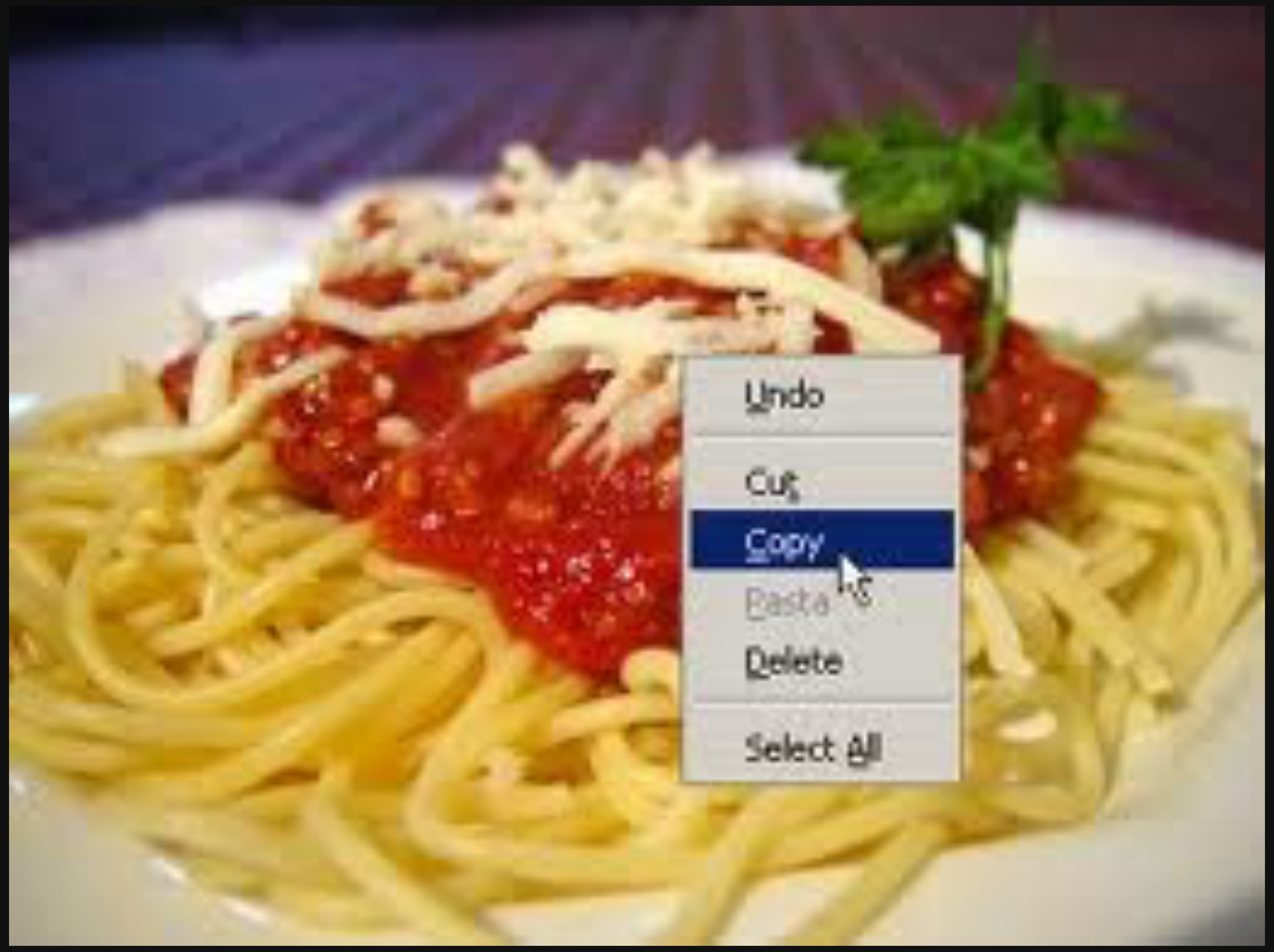
- Ideas
- Guidelines
- Rules
- Ethics

Code Reuse and Libraries

- Never write new code if you don’t have to
- Chances are somebody has already done it, Google it
- Use libraries, its better than anything you’ll write

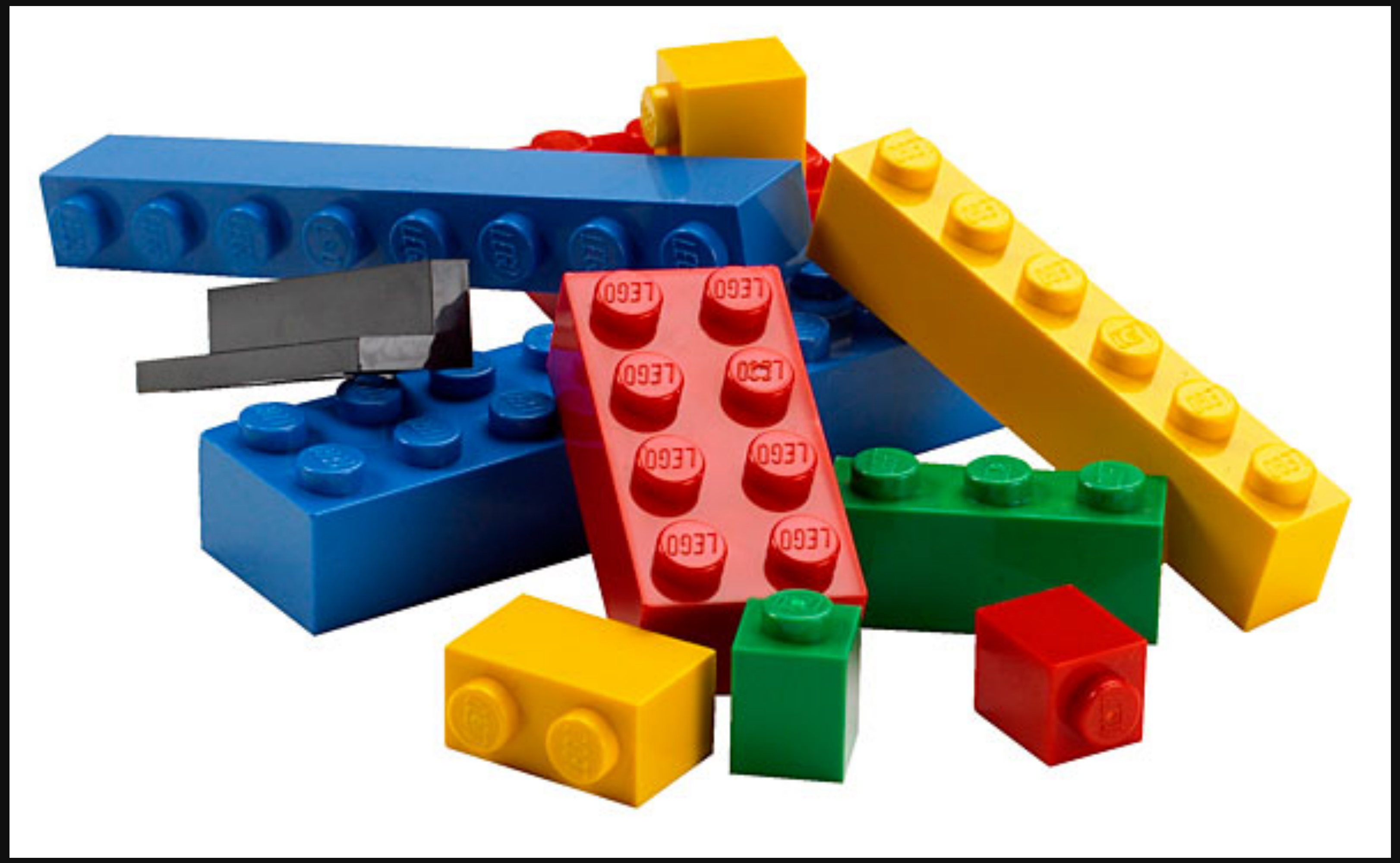
DRY

- Don’t Repeat Yourself
- No more copypasta
- Modularize your code
- Rule of Three: If you used it three times, put it in a module



Modularity

- Organize code into independent, interchangeable modules
- Functions, structures, objects etc
- Building blocks to make something bigger
- Universal and extendable



YAGNI

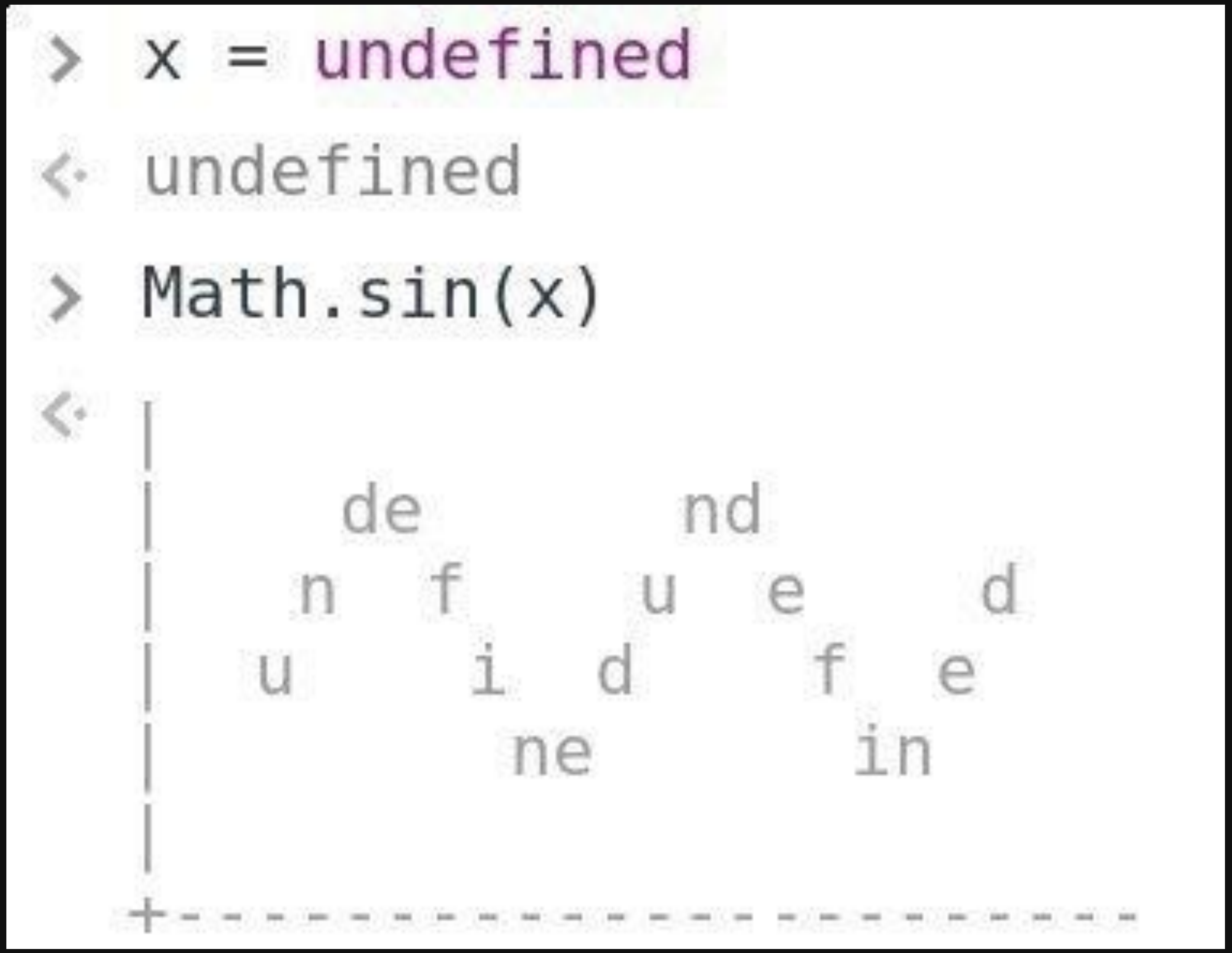
- You Aren’t Going To Need It
- “Always implement things when you actually need them, never when you just foresee that you need them” - Ron Jeffries
- Don’t waste time writing code that you may not need and will only complicate things

KISS

- Keep It Simple Stupid
- Start with the simplest thing that could possibly work

Generic Progamming & Polymorphism

- Write code to work no matter what
- Account for all possible uses
- Weak typing and abstraction
- Envision each function as a Black Box



```
> x = undefined
<· undefined
> Math.sin(x)
<· |
   |    de      nd
   |   n  f    u  e    d
   |  u    i  d    f  e
   |        ne      in
   |
   +-----------------------
```

SOLID for OO

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- Each module should only have a single functionality
- “A class should only have one reason to change” - Robert C Martin

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Open/Closed Principle

- “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” - Bertrand Meyer
- Write code that doesn’t have to be changed when the requirements change

Open/Closed Example

- Have a function that calculates area of rectangle

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

Open/Closed Example

- Write a function which computes total area of a bunch of rectangles

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Open/Closed Example

- Now expand it to do circles too, then for trapezoids ad infinitum

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

Open/Closed Example

- Or write it better from the start

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

Liskov Substitution Principle

- If object S is a subtype of object T, then objects of type T can be replaced by objects of type S without breaking anything
- New subtypes must extend behavior without modifying original

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Example

- Imagine a class Rectangle and subclass Square
- Square breaks the functionality of Rectangle without extending it

```java
class Rectangle {
  int width;
  int height;

  public void setWidth(int w){
    width = w;
  }

  public void setHeight(int h){
    height = h;
  }

  public int getArea(){
    return width * height;
  }
}
```

```java
class Square extends Rectangle {
  public void setWidth(int w){
    width = w;
    height = w;
  }

  public void setHeight(int h){
    width = h;
    height = h;
  }
}
```

Interface Segregation Principle

- No client should be forced to depend on methods it doesn’t use
- Split large interfaces into smaller ones
- If you only want to eat food, you shouldn’t have to set the table first
- Xerox example

*An interface is a list of methods that a given class must implement

Dependency Inversion Principle

- "High-level modules should not depend on low-level modules. Both should depend on abstractions."
- "Abstractions should not depend on details. Details should depend on abstractions."
- Make code modules depend on concepts(interfaces) instead of each other
- e.g. an outlet has some connections, we can connect them however we please

DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Exception Handling

- Programmatically resolve errors instead of crashing
- Resolve error and continue execution
- Print meaningful error messages
- Even define and throw your own errors
- Most languages have built-in exception handling, you just have to use it

```
public void initialize() {
  try {
    loadRoomConfig();
    loadBoardConfig();
    calcAdjacencies();
    loadConfigFiles();
  } catch (BadConfigFormatException e) {
    e.getMessage();
  } catch (FileNotFoundException e){
    e.getMessage();
  } catch (Exception e){
    e.getMessage();
  }

  dealCards();
}
```

Test-Driven Development

- Define parameters and write failing tests
- Write code to pass tests
- Periodically run tests during development to ensure no regression
- Use testing libraries such as JUnit(Java), googletest(C++), PyUnit(Python)

```java
//Tests adjacency list for cell in the top left corner of board
@Test
public void testAdjacencyTopLeft(){
  BoardCell cell = board.getCell(0,0);
  LinkedList<BoardCell> testList = board.getAdjList(cell);
  assertTrue(testList.contains(board.getCell(1, 0)));
  assertTrue(testList.contains(board.getCell(0, 1)));
  assertEquals(2, testList.size());
}
```

Documentation

- Document your code
- Comment throughout your code
- Explain what it does and how it works
- Write README’s and describe the Black Box functionality

Refactor

- Periodically refactor your code
- You’ll understand the project better after you write it
- Clean up garbage code
- Rename things
- Reorganize and streamline

Take Home Message

- Keep these in mind when coding
- Write better code now and forget about it later
- Don’t write fragile code
- Write code that is easy to use, understand and extend
- Goal: once a piece of code has been finished, you should have to touch it again