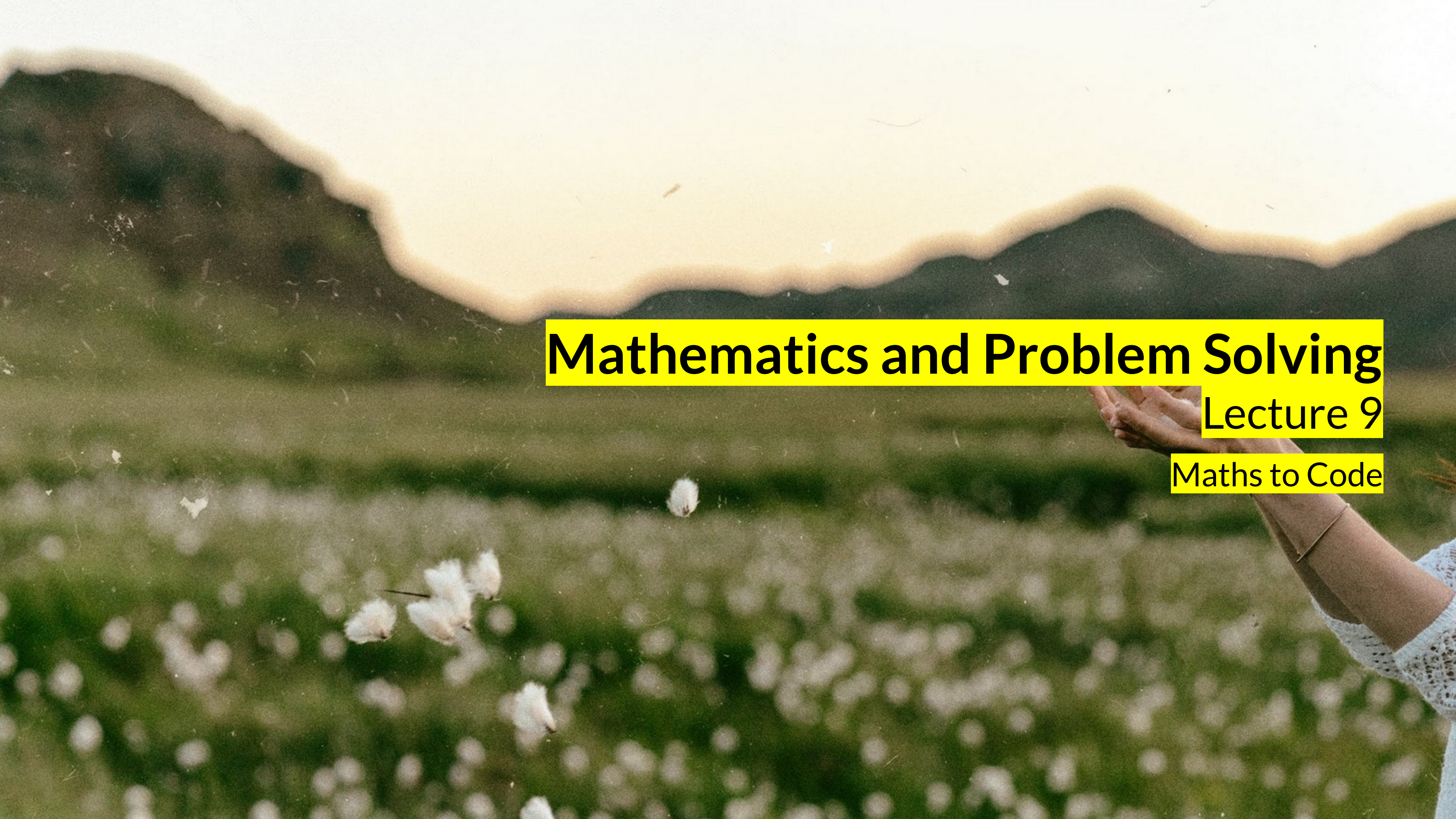


Mathematics and Problem Solving

Lecture 9

Maths to Code

Overview

- Two Paradgims
- Bridging the Gap
- Functions
- Nuts and Bolts

- This might hurt a little
  - There are no easy answers
  - It’s about **thinking in different ways**

Two Paradigms

Programming

- What is programming?

Programming

- What is programming?
  - Writing a sequence of instructions
  - Designing an algorithm
  - Controlling the flow of execution

Programming

- What is **imperative** programming?
  - Writing a sequence of instructions
  - Designing an algorithm
  - Controlling the flow of execution

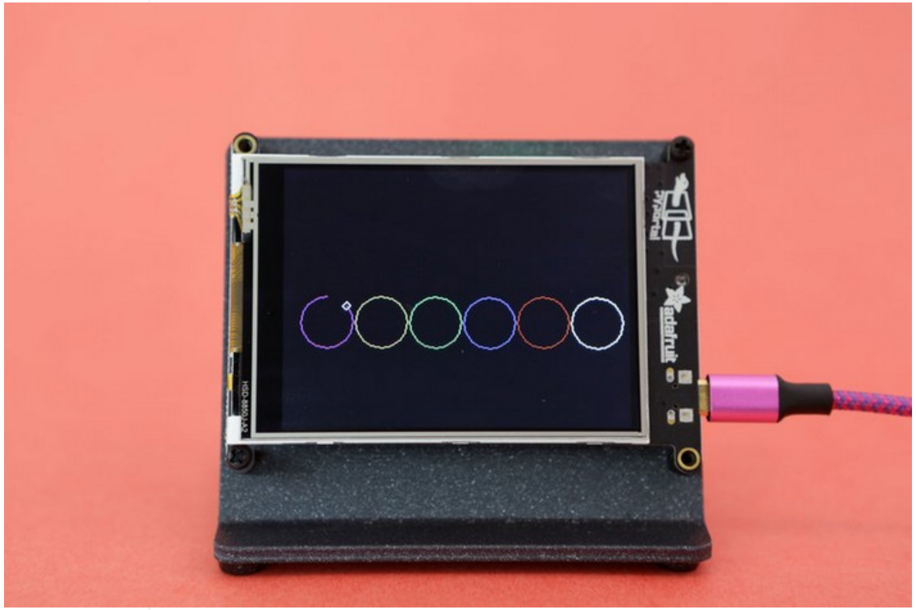
```python
import board
from adafruit_turtle import Color, turtle

turtle = turtle(board.DISPLAY)

mycolors = [Color.WHITE, Color.RED, Color.BLUE, Color.GREEN,
Color.ORANGE, Color.PURPLE]
turtle.penup()
turtle.forward(130)
turtle.right(180)
turtle.pendown()

for i in range(6):
    turtle.pencolor(mycolors[i])
    turtle.circle(25)
    turtle.penup()
    turtle.forward(50)
    turtle.pendown()

while True:
    pass
```



Maths

- What is maths?
  - Declaring what is the case
  - Defining relations between ideas
  - Solving within constraints

$$f(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ F(n-1) + F(n-2) & if \quad n > 1 \end{cases}$$

The Conceptual Divide

- Programming is imperative*
- Maths is declarative

- Imagine trying to write a program to perfrom the process of mathematics
  - Detect when it can apply transformation rules
  - Transform strings
  - Infer patterns
  - Find solutions within constraints
  - Reason over this process to prove that some solutions don’t exist

- Imagine writing maths to describe a program
  - Function: input → output
    - Whole code
    - Each method
  - Sequnce of all variables
    - [(a_1,b_1,c_1), (a_2,b_2,c_2), ... (a_n,b_n,c_n)]
    - Function relating them
  - Recursion

Maths

- Generalities
- Solving
- Proving
- Holistic
- Intuitive

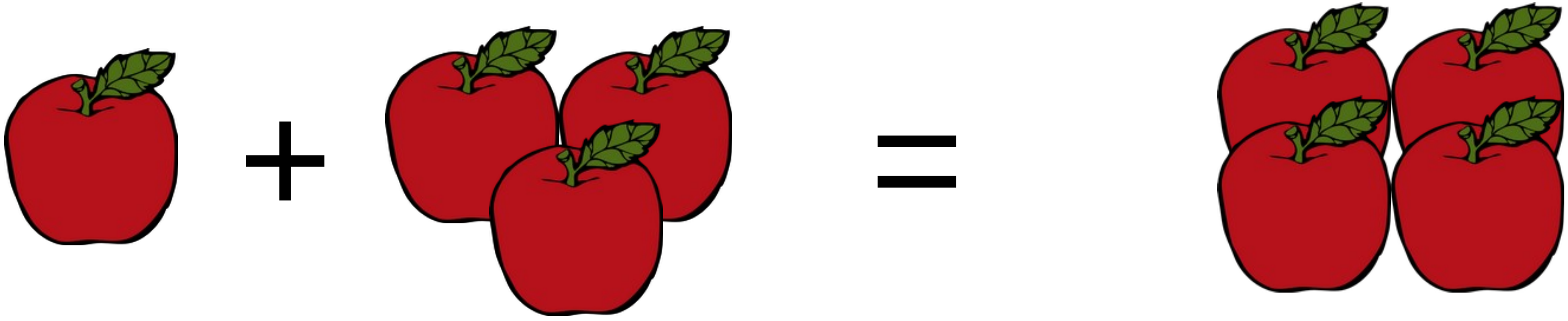
Programming

- Instances
- Evaluating
- Testing
- Local
- Formal


Bridging the gap

Think like a school child

- We reason holistically and intuitively
  - But only **once we are familiar with patterns**
- We are **taught** step-by-step instructions



- Work through the maths yourself with example values
- What steps do you take?
  - Each step is a line of code
  - Create interim variables for clarity
  - Add brackets for clarity

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}}$$

- Mean
  - double meanOfX = mean(x)
- In the sum
  - Difference
    - double diff = x[i] – meanOfX
  - Square
    - double diff2 = diff * diff
- Sum
  - for (int i=0;i<n;i++) { }
- Division
  - double sSquared = Sum / n-1
- Square root
  - double s = Math.sqrt(sSquared)

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

Make code more declarative

- We define things in code all the time
  - double pi = 3.14;
  - int add(int a, int b) { return a + b; }
  - class Set
- Look for declarative ways to write your code

- What does it mean to define?
  - X **is** something (**is a**)
    - Or has a particular **type**
  - X **has** something (**has a**)
    - Or stores particular types of variables

Has a

- In OOP, we design **classes**
- Classes **have** certain properties
  - Variables
  - Methods
- All of this type information is **declarative**

```
export default class InductiveProof1 implements Proof
{
    public readonly SEEDMIN: number = 0;
    public readonly SEEDMAX: number;

    // i(xk + j)(yk + l)(zk + m) + n
    i: number;
    j: number;
    l: number;
    m: number;
    n: number;
    x: number;
    y: number;
    z: number;
    mod: number;
    rem: number;
    valid: boolean;
    r: number;
    seed: number;

    trivial: boolean
```

- Variables, objects and functions have a **name**
  - No operational effect
  - Really important for understanding your code!
  - **Choose good descriptive names!**

Is a

- When we ask what a thing *is* we're asking about it's **type**
  - numerical?
  - boolean?
  - Array?
  - Object? (ClassA, ClassB, ClassC, …)
    - What does the class **have**?
  - **Function**
    - What are it's arguments?
    - What is it's return type?

- OOP gives us powerful inheritence tools
  - A class can **extend** another class
  - A class can **implement** an interface
- Each class in a heirarchy defines some properties
  - When a class *Cat* extends another class *Animal*, the *Cat* **is a** *Animal*
  - *Cat* has the properties of an *Animal*
  - **ArrayList** is a **AbstractList** is a **AbstractCollection** is an **Object;** and **ArrayList** is a **Serializable, Clonable, Iterable, Collection, List, RandomAccess**

```
export default class GameGraph
{
    private _vertices: Vertex[];
    private _edges: Edge[];
    private _changes: Pool<GraphChange>;
    private _outTerminals: Vertex[];
    private _inTerminals: Vertex[];
    private _sourcesEnabled: boolean = true;

    get changes() { return this._changes; }
    get size() { return this._vertices.length; }
    get sourcesEnabled() { return this._sourcesEnabled; }
```

```
export default interface IGameBoard
{
    onTokenCreated: Event1; // token: TokenData
    onTokenRemoved: Event1; // token: TokenData
    onTokenMoved: Event1; // token
    onGroupsSelected: Event2; // groups: ScoringGr
    tokenSpecs: TokenSpec[];
    siblings: number[][];
    descriptions: string[][];

    graph: GridGraph;
    elements: TokenModel[];
    orderedTokens: string[][];
```

```
export default class GridGraph extends GameGraph
{
    private _width: number;
    private _height: number;

    get width() { return this._width; }
    get height() { return this._height; }

    constructor(width: number, height: number)
    {
        let vertices: Vertex[] = new Array(width*height);
        let edges: Edge[] = new Array(width*(height-1));
        let outTerminals: Vertex[] = new Array(width);
        let inTerminals: Vertex[] = new Array(width);
        let nextEdge = 0;
        let nextVertex = 0;

        for (let x=0;x<width;x++)
```

```
export default class GameBoardModel implements IGameBoa
{
    onTokenCreated: Event1 = new Event1(); // token: To
    onTokenRemoved: Event1 = new Event1(); // token: To
    onTokenMoved: Event1 = new Event1(); // token
    onGroupsSelected: Event2 = new Event2(); // groups:
    tokenSpecs: TokenSpec[];
    siblings: number[][];
    descriptions: string[][];

    graph: GridGraph;
    elements: TokenModel[] = [];
    orderedTokens: string[][];

    create() { this.graph.fill(); }
```

- Structure your code so that it’s
  - Easy to do things that work
  - Hard to do things that don’t work
- Explicit typing is your friend

Bridging the Gap

- Make your maths more **imperative**
  - Think like a school child
- Make your code more **declarative**
  - What is x? What does x have?
  - Of functions: What arguments does x take? What sort of thing does it return?
  - What shall I call it?

Functions

- Not all programming langages are imperative
  - **Declarative Programming**
- Program defines a problem domain
  - Defines **what** the program should achieve
  - Not **how** it achieves it

- Functional Programming
  - Programs are constructed of functions, combining other functions...
  - Functions are **first-class citizens**
    - You can pass them as arguments and return them from other functions
  - You don’t have a **state**
    - No variables to store and manipulate
    - Lazy evaluation means that functions are evaluted when needed

- Make good use of functions
  - They can be small
  - Do a single well defined task
  - Call other functions
  - Be recursive
  - You don’t need to store the value if you have a function to calculate it
- See if your programming langauge supports
  - Delegates (e.g. C#)
  - Callbacks (e.g. JS)
  - (Not really supported in Java :( )

- A function might be defined imperatively, but once it's written, it's a magic box
  - If it does a clearly defined job, you can treat it like a mathematical function
- Ensure **side-effects** are always expected (and as expected)
  - Maths doesn't have side-effects!

Good **use of functions** is the most important step to making **maths code** more **managable**

Nuts and Bolts

Types

- Integers → Integer Primatives
  - **byte** 8 bits -128 to +127
  - **short** 16 bits -32,768 to +32,767
  - **int** 32 bits -2 billion to +2 billion (approximately)
  - **long** 64 bits -9×10^{18} to $+9 \times 10^{18}$ (approximately)

- Reals → Floating Point Primatives
  - **float** 32 bits $-3.4x10^{38}$ to $+3.4x10^{38}$
  - **double** 64 bits $-1.7x10^{308}$ to $1.7x10^{308}$
- Beware: precision
  - Treat floating point values as non-deterministic

- Arrays → Arrays
  - $X = [X_1, \ldots, X_n] \rightarrow$ int[]
- Watch out for indexes
  - Maths usually indexes from 1 → n
  - Code usually indexes from 0 → n-1

- Mathematical objects such as
  - Sets
  - Tuples
  - Graphs

  you’ll probably need to create your own class

Operators

- $a + b \rightarrow$ a + b
- $a - b \rightarrow$ a - b
- a x b, ab, a.b → a*b
- $a \div b, \frac{a}{b} \rightarrow$ a/b
- a mod b → a % b
- $a^b \rightarrow$ Math.pow(a,b)
- $\log_2 a \rightarrow$ Math.log(a)

- Log to any base
- $log_b \ a = \frac{log \ a}{log \ b}$

```
int log(int a, int base)
{
  return log(a) / log(base);
}
```

- a ∧ b → a && b
- a∨b → a || b
- ¬a → !a

And when this doesn’t work

- Ask: what is this formula achieving?
  - Isolate the formula you can’t translate
  - Think of the formula like a function f(x) → ?
    - What does it depend on? (i.e. the inputs)
    - What is it’s output?
  - How can you write the same function a different way?
    - Perhaps even as a *function*!
  - eg. Summation → for loop

Elegance and Clarity

- Brackets
  - There’s no harm doing ((a) + (b))
- Simplify
  - Do only as many operations in one line that are easy to understand
  - Interim variables
- Descriptive names
  - Reading the name of a function in context should make it clear exactly what it does

Takeaways

- Make **maths** more **imperative**
- Make **code** more **declarative**
- Make **good use** of **functions**
- Make sure you know **what** your **types** are
- **Descriptively** named, **interim variables**
- Translate each operator. When in doubt, add **brackets**