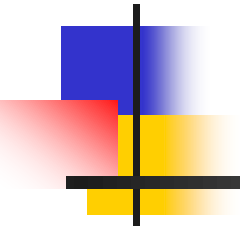
Algorithmic Thinking and Structured Programming (in Greenfoot)



Teachers:

Renske Smetsers-Weeda

Sjaak Smetsers

Ana Tanase

Today's Lesson plan (3)

- 10 min Looking back
  - What did we learn last week?
  - Discuss problems / homework (and handing-in)
  - Answers PRE-task
- Sneak preview
- Blocks of theory/unplugged and exercises

- 10 min Wrapping up
  - Homework
  - Next week: quiz

Discuss problems / homework

- Problems handing-in?
- We’re missing:
  - Assignment 1:
    - Martijn & Jelmer
    - (ex 5.5 until the end) Lieke & Mats
    - Tim
  - Assignment 2:
    - Martijn & Jelmer
    - Bram & Tim
- Please still hand-in!

Discuss problems / homework

- Problems handing-in?
- Where to find assignments and hand-in homework?
  - Assignments posted on magister
    - Class appointment
    - Homework due Wednesday on today's date or next week??
  - Hand-in: email to Renske.weeda@gmail.com
  - Paper: pigeon hole "Renske Smetsers"
  - PAX students: paper during Thursday's INF class

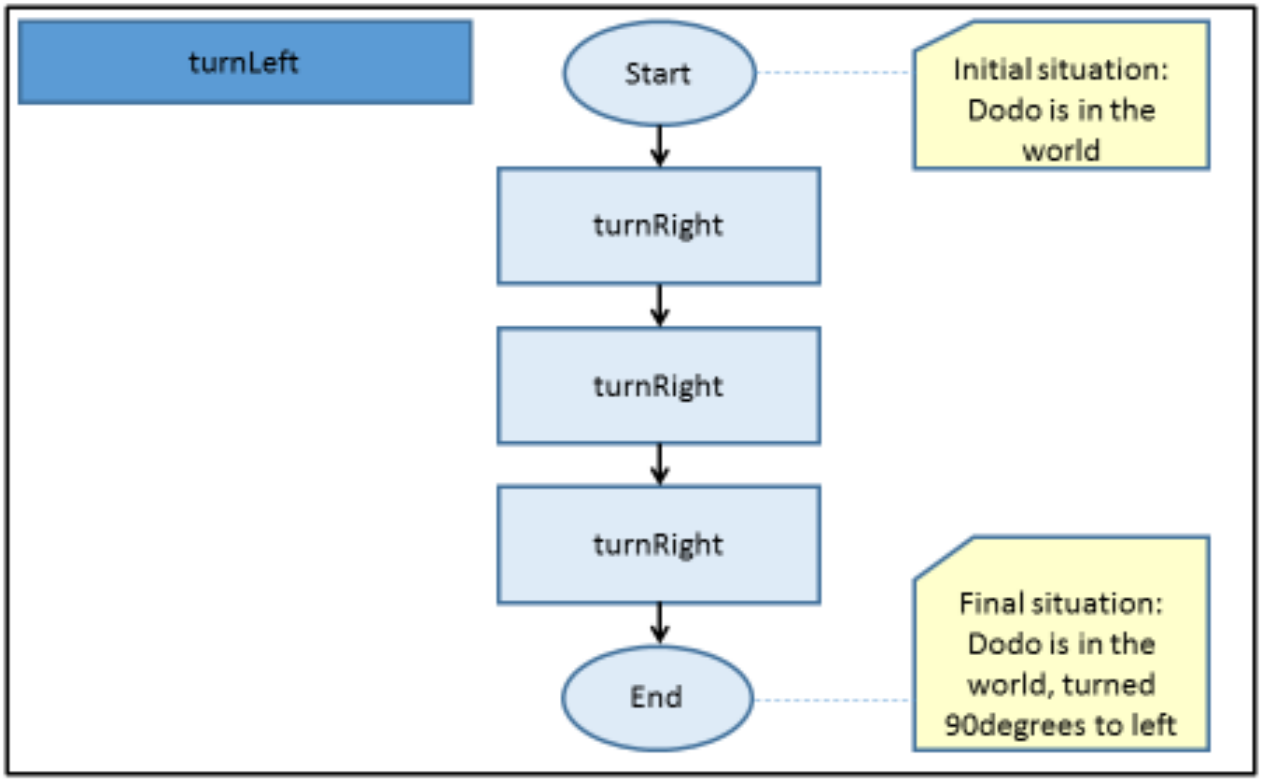
Discuss problems / homework

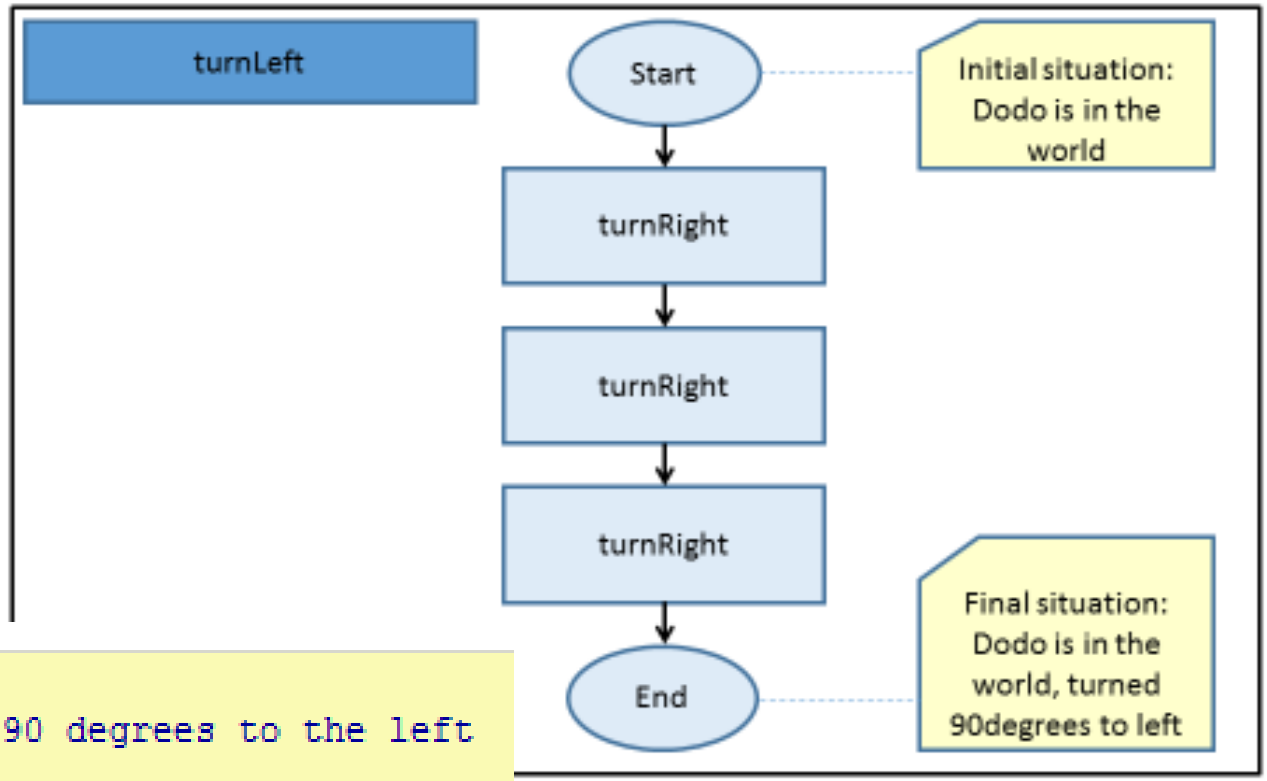
- Give instructions from object's (myDodo's) perspective
- i.e. moveForward (instead of moveRight)

Discuss problems / homework

- Method call vs. Method declaration

- **Demo** of steps to writting new code:
  - Adding a method to turn myDodo to the left
  - Tip: Dodo has a turnRight method


turnLeft
Start
Initial situation: Dodo is in the world
turnRight
turnRight
turnRight
End
Final situation: Dodo is in the world, turned 90degrees to left




```
/**
 * Turn 90 degrees to the left
 */
public void turnLeft(){
    turnRight();
    turnRight();
    turnRight();
}
```

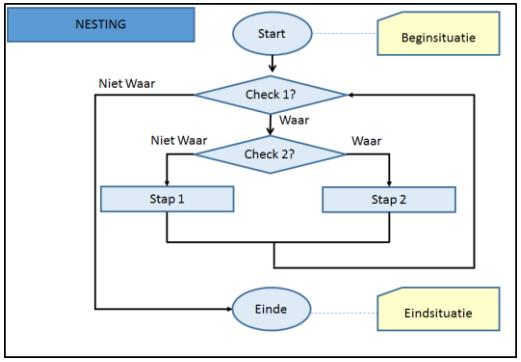
```
public void act() {
    turnLeft();
}
```

Challenge & problem

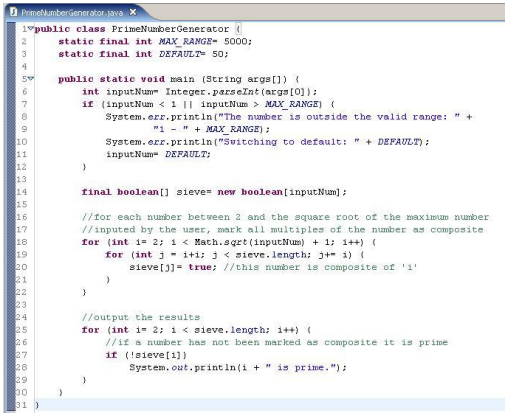
You must perform two aspects well:

1) Create a *problem-solving algorithm* (a disciplined and creative process)

We use a systematic approach



2) *Formulate* that algorithm *in terms of a programming language* (a disciplined and very precise process)



```
PrimeNumberGenerator.java
public class PrimeNumberGenerator {
    static final int MAX_RANGE= 5000;
    static final int DEFAULT= 50;

    public static void main (String args[]) {
        int inputNum= Integer.parseInt(args[0]);
        if (inputNum < 1 || inputNum > MAX_RANGE) {
            System.err.println("The number is outside the valid range: " +
                    "1 - " + MAX_RANGE);
            System.err.println("Switching to default: " + DEFAULT);
            inputNum= DEFAULT;
        }

        final boolean[] sieve= new boolean[inputNum];

        //for each number between 2 and the square root of the maximum number
        //inputed by the user, mark all multiples of the number as composite
        for (int i= 2; i < Math.sqrt(inputNum) + 1; i++) {
            for (int j = i+i; j < sieve.length; j+= i) {
                sieve[j]= true; //this number is composite of 'i'
            }
        }

        //output the results
        for (int i= 2; i < sieve.length; i++) {
            //if a number has not been marked as composite it is prime
            if (!sieve[i])
                System.out.println(i + " is prime.");
        }
    }
}
```

Always check that your algorithm is correct by running/testing the implementation!

Computational thinking

- **Working in a structured manner:**
  - Breaking problems down into subproblems
  - Design, solve and test solutions to subproblems
  - Combing these (sub)solutions to solve problem
- **Analyzing** the quality of a solution
- **Reflecting** about the solution chosen and proces
- **Generalizing** and re-use of existing solutions

Discuss problems / homework

- Getter vs. setter methods

- A class has it's own:
  - Methods
  - Data

- No other object may touch/change this (safe idea!!!)
  - Want info: ask the object with a get method
  - Want to change data: ask the object with a set method
- Object changes/gives data (if you are allowed)

Getter vs. Setter methods

- getNrOfEggs:
  - **Question**
  - Dodo, tell me how many eggs you have laid

- setNrOfEggs;
  - **Statement**
  - Dodo: this is the number of eggs you have to do something with

Types

- Say which types you want to use only:
  - When you declare something
  - Eg: a method declaration

```
int getNrOfEggsNeeded ( ) {
        ….  // this is how eggs Mimi must retrieve
}
```

- Afterwards, when you use a method:
  - DON'T 'declare', just USE it!
  - Also in flowcharts!
  - using a method in code:

```
while (nrEggsFound < getNrOfEggsNeeded ( ) ) {
            findMoreEggs();
}
```

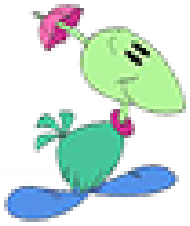
Pre-task (discuss answers)

Sneak preview

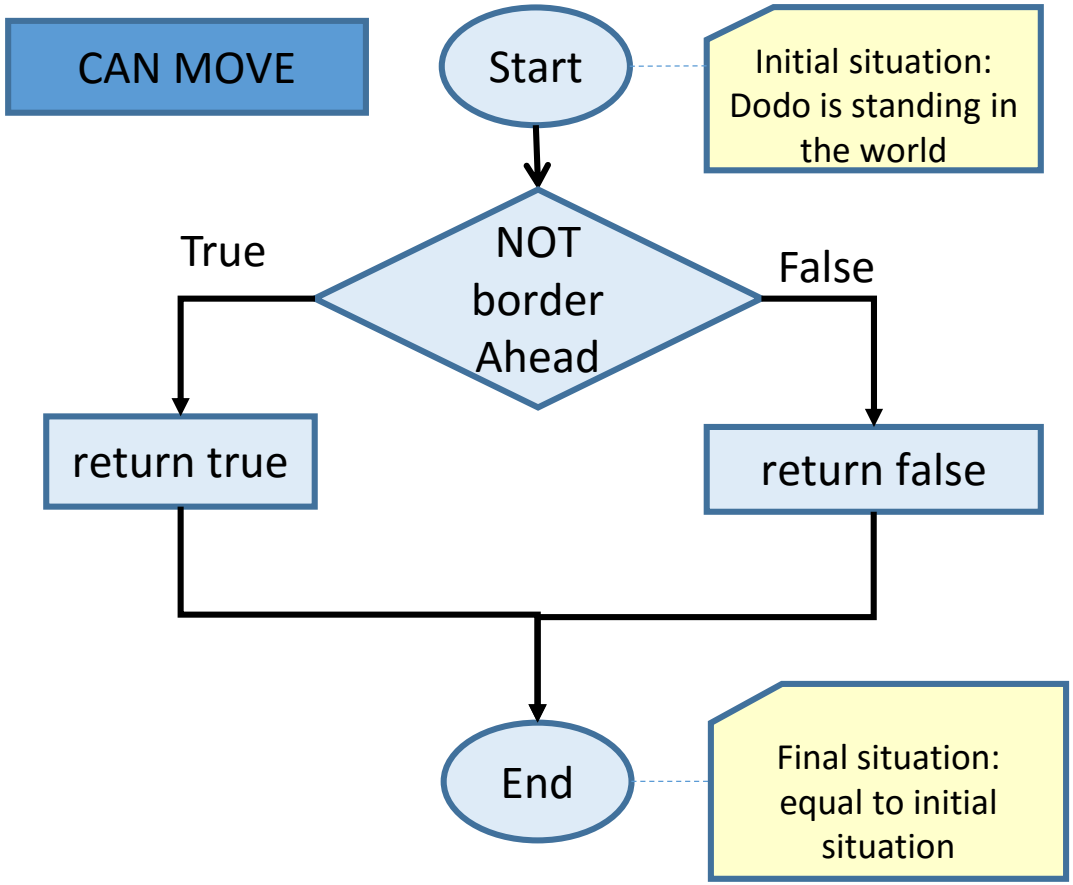
What did we learn last week?

- Parameters, signatures, method calls, results
- Mutator / accessor methods
- Flowcharts

Accessor methods (questions)



flowchart

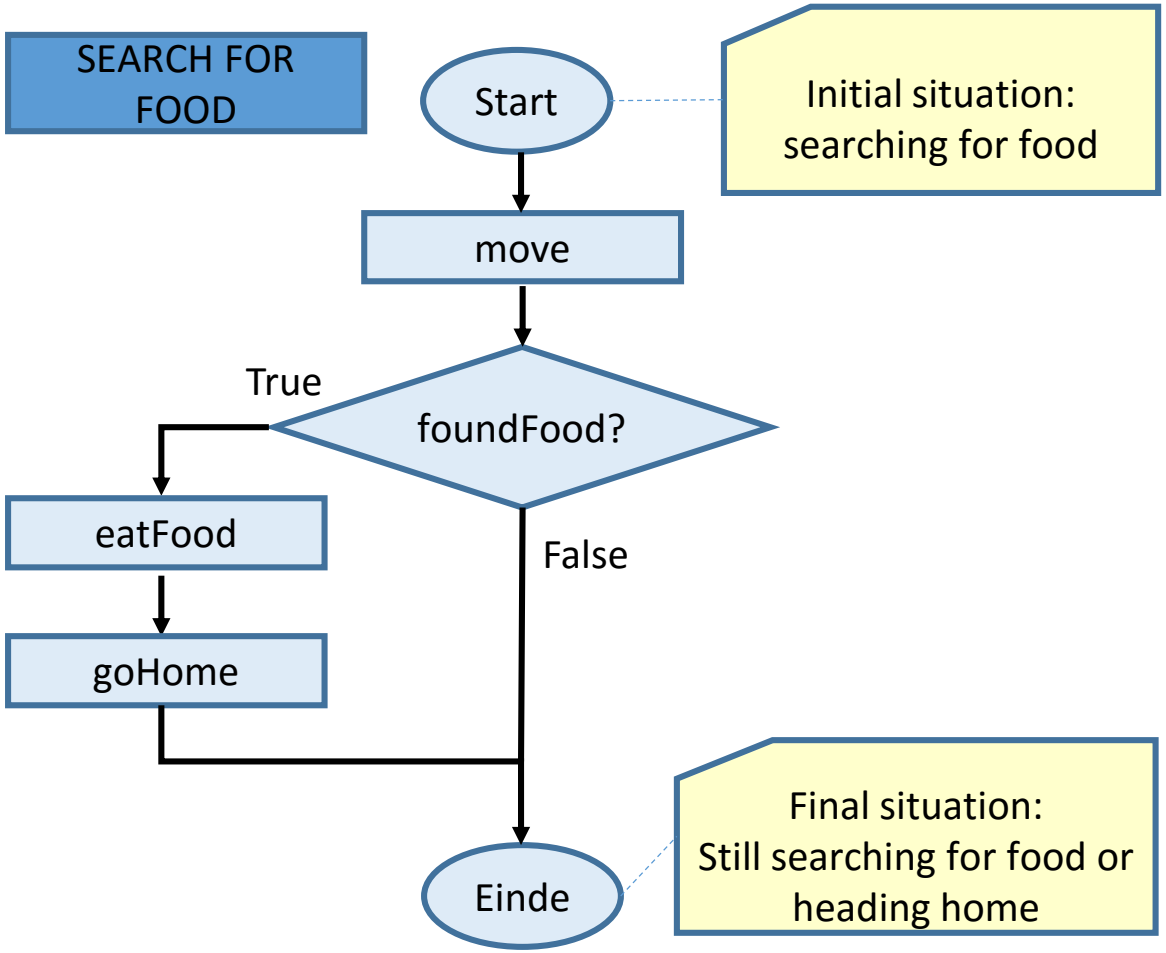


code

```
public boolean canMove() {
if ( ! borderAhead () ) {
      return true;
   } else {
      return false;
   }
}
```

Mutator methods (behavior)

flowchart



code

```
public void act() {
    move();
    if ( foundFood() ) {
       eatFood();
       goHome();
    }
}
```

Anatomy of a method (1)

Signature: first line of a method declaration (up to {)

```
public void jump( int distance ) {
        instructions
        of the method ("body")
}
```

Anatomy of a method (3)

What answer (value) is returned?

void = nothing returned

int = returns an integer (0, 1, 2, ...)

etc. a method can return *anything*

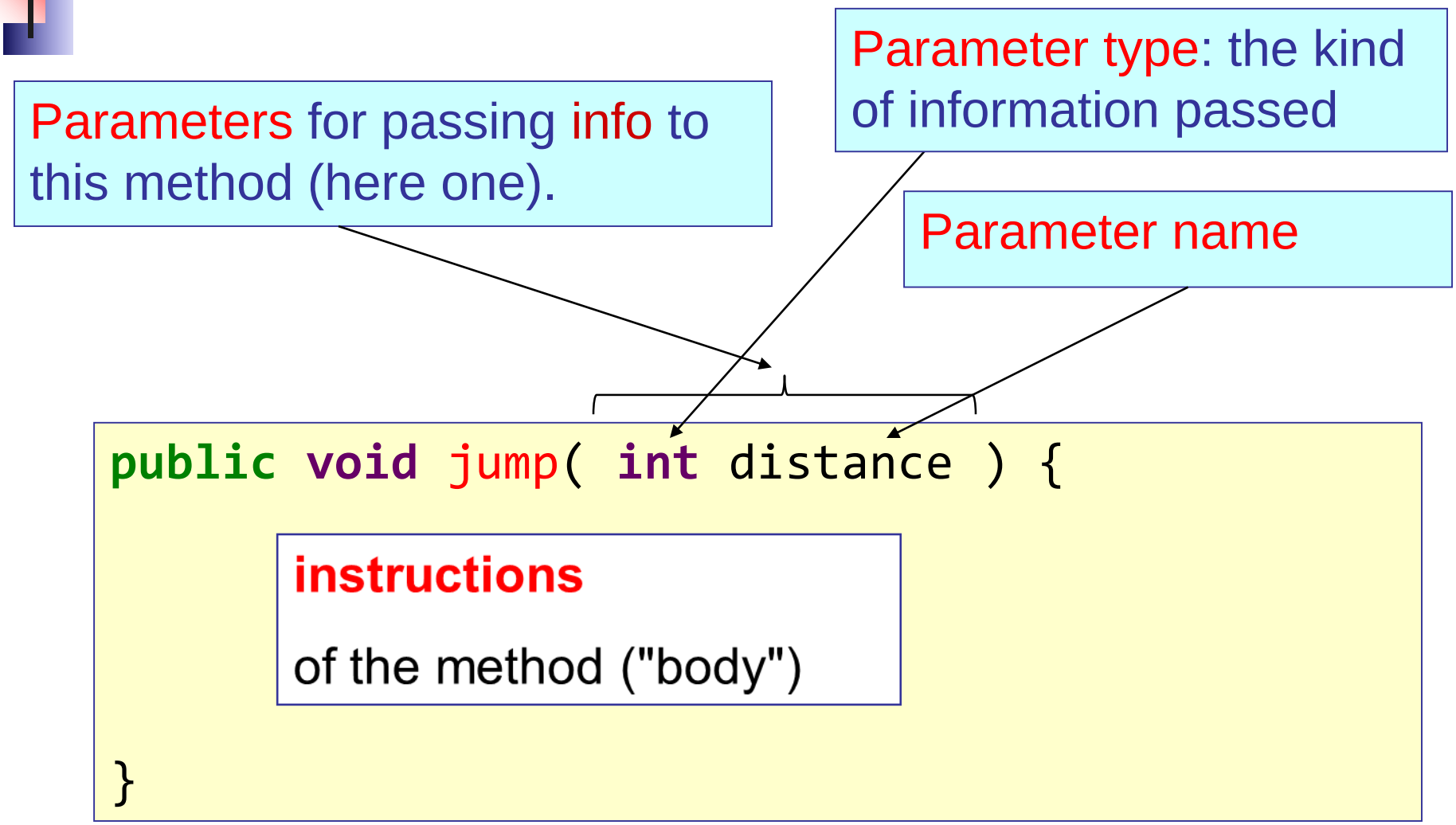
```
public void jump( int distance ) {

        instructions

        of the method ("body")

}
```

Anatomy of a method (3)

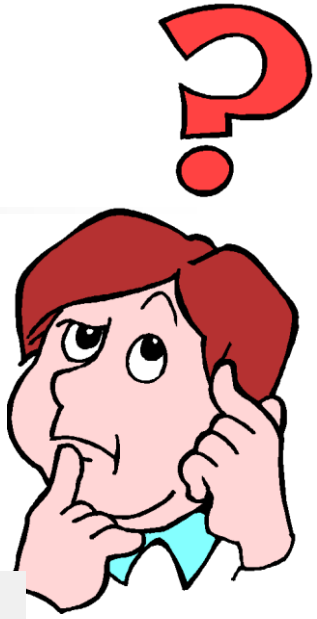
Name of this method

```
public void jump( int distance ) {

        instructions

        of the method ("body")

}
```

Anatomy of a method (4)

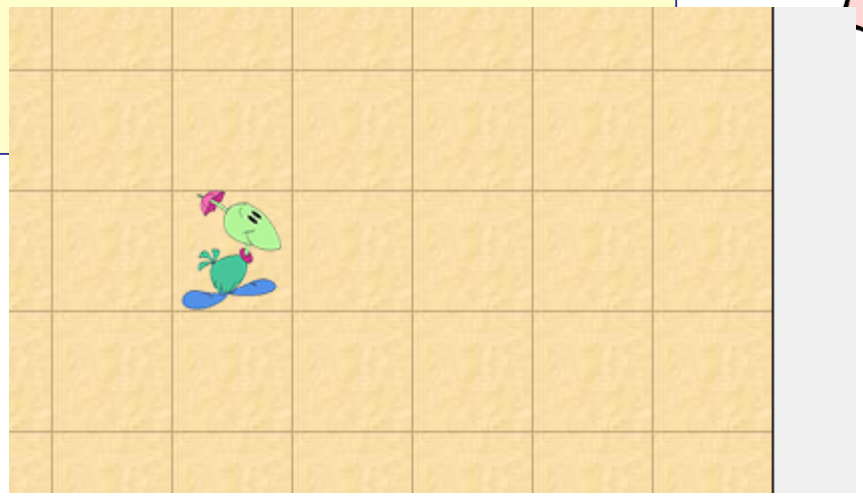


Anatomy of a method (5)



Return a boolean (true or false)

```
public boolean canJump( int distance ) {
   << body >>
}
```



Today's Lesson Goals

- Checking and assigning values
- Algorithms & flowcharts:
  - Sequences
  - Selection (if-then-else)
  - Repetition (While)
- Structured code modification & debugging
- Quality of a solution

Counting

- Starts at….

Unplugged: Swap puzzle

What it's about:

- Coming up with an algorithm
- Looking/planning ahead
- Efficiency
- Testing

Swap Puzzle

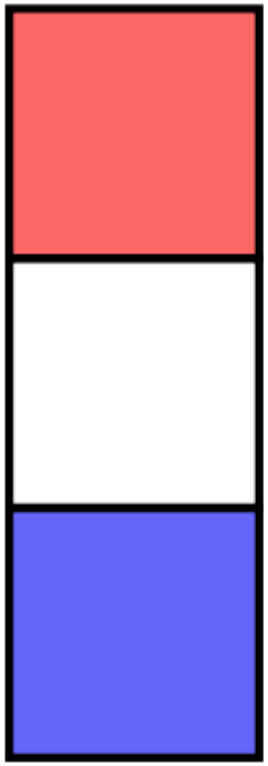
- Pieces start on different (non-white) color
- A piece can move to an empty adjacent square
- Can jump over an adjacent piece of other color onto an empty square

- Method to use: **getsThePieceFrom**

Step 1: Square 1 **GETS THE PIECE FROM** Square 0

- Solve the puzzle in the least amount of steps
- Write down the steps

Swap Puzzle level 1



Square 0

Square 1

Square 2

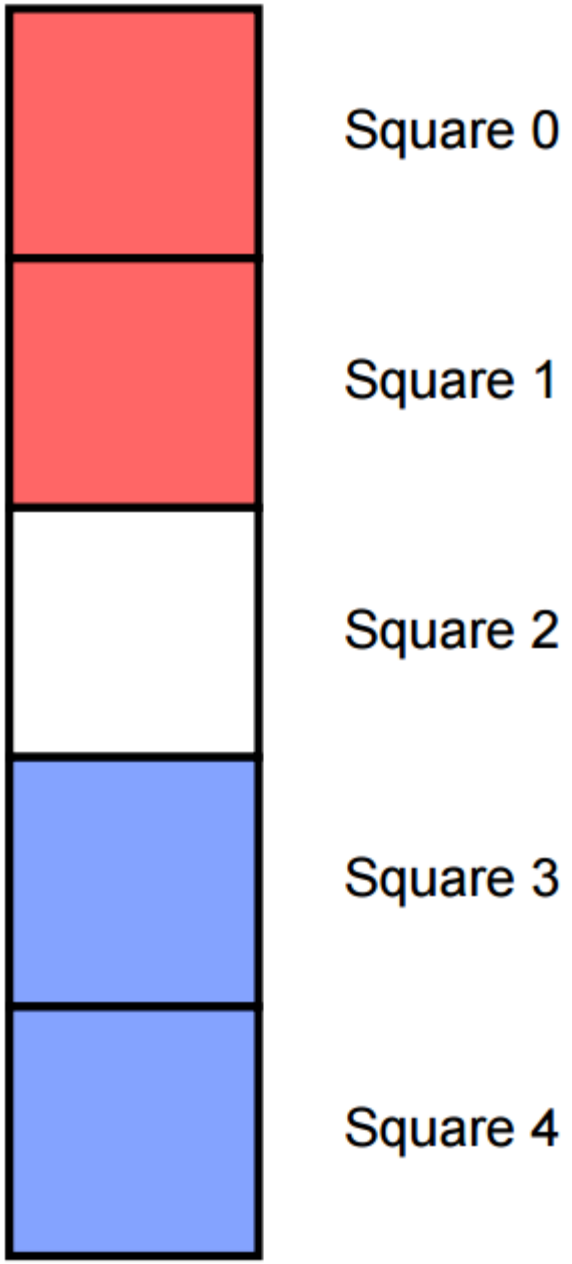
STEP	TO	COMMAND	FROM
Step 1:	Square 1	GETS THE PIECE FROM	Square 0
Step 2:	Square 0	GETS THE PIECE FROM	Square 2
Step 3:	Square 2	GETS THE PIECE FROM	Square 1

Swap Puzzle level 2

Challenge:

Most efficient algorithm?

- What to count/compare with?
- How to test?



Swap Puzzle level 2

The level 2 puzzle can be solved in 8 moves as follows:

Step 1: **Square 2** GETS THE PIECE FROM **Square 1**

Step 2: **Square 1** GETS THE PIECE FROM **Square 3**

Step 3: **Square 3** GETS THE PIECE FROM **Square 4**

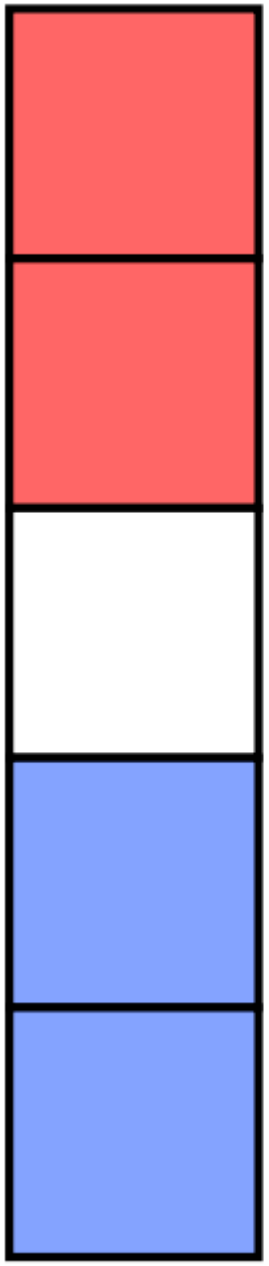
Step 4: **Square 4** GETS THE PIECE FROM **Square 2**

Step 5: **Square 2** GETS THE PIECE FROM **Square 0**

Step 6: **Square 0** GETS THE PIECE FROM **Square 1**

Step 7: **Square 1** GETS THE PIECE FROM **Square 3**

Step 8: **Square 3** GETS THE PIECE FROM **Square 2**



Assigning values

- SQUARE 1 GETS THE PIECE FROM SQUARE 2
- Means:

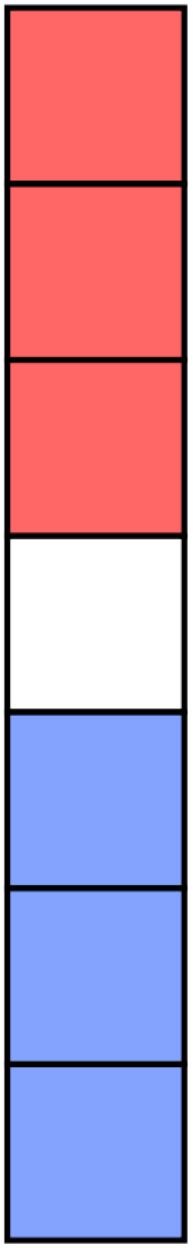
  Set SQUARE 1 to (value of) SQUARE 2

- In Java code:

```
square1 = square2;
```

Swap Puzzle level 3

- Which pair can find the algorithm with min. # steps?
- Less writing?
  - Use: sq1 = sq2
  - Which means: square 1 getsPieceFrom square2
- When you're sure you know (and tested):
  - Check that it can't be done faster (count #steps)
  - Check your steps
  - Hold op your hand
  - Challenge: class will execute your steps



Swap Puzzle level 3

The level 3 puzzle can be solved in 15 moves as follows:

Step 1: **Square 3** GETS THE PIECE FROM **Square 2**

Step 2: **Square 2** GETS THE PIECE FROM **Square 4**

Step 3: **Square 4** GETS THE PIECE FROM **Square 5**

Step 4: **Square 5** GETS THE PIECE FROM **Square 3**

Step 5: **Square 3** GETS THE PIECE FROM **Square 1**

Step 6: **Square 1** GETS THE PIECE FROM **Square 0**

Step 7: **Square 0** GETS THE PIECE FROM **Square 2**

Step 8: **Square 2** GETS THE PIECE FROM **Square 4**

Step 9: **Square 4** GETS THE PIECE FROM **Square 6**

Step 10: **Square 6** GETS THE PIECE FROM **Square 5**

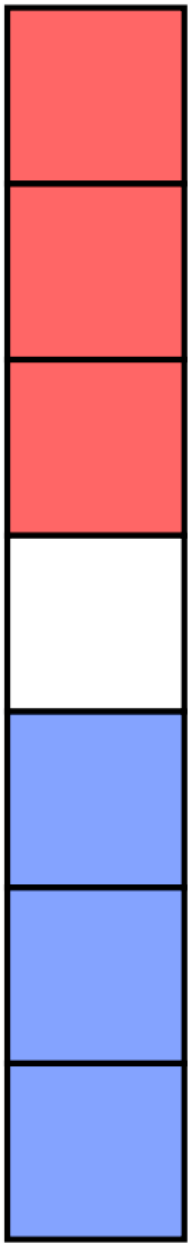
Step 11: **Square 5** GETS THE PIECE FROM **Square 3**

Step 12: **Square 3** GETS THE PIECE FROM **Square 1**

Step 13: **Square 1** GETS THE PIECE FROM **Square 2**

Step 14: **Square 2** GETS THE PIECE FROM **Square 4**

Step 15: **Square 4** GETS THE PIECE FROM **Square 3**



Swap puzzle: what its about

- Describing your steps => algorithm!
  - Series of actions to get the job done
  - Algorithm? Then you'll still have solution next week

- Importance of testing:
  - **before**: step through your answer (like processor)
  - **after**: don't assume it works, check it!
- Efficiency
  - Think of a solution that works, then check efficiency
- Looking ahead vs. trail and error
  - Consider all possible moves
  - Necessary when the puzzle gets harder

Swap-puzzle and assigning values

Assigning values using =

- SQUARE 1 GETS THE PIECE FROM SQUARE 2
- Means:

  Set SQUARE 1 to (value of) SQUARE 2
- In Java code:

  square1 = square2;

Check value using ==

- In Java, to check if square1 is red:

```
if ( square1 == red ) {
    …
}
```

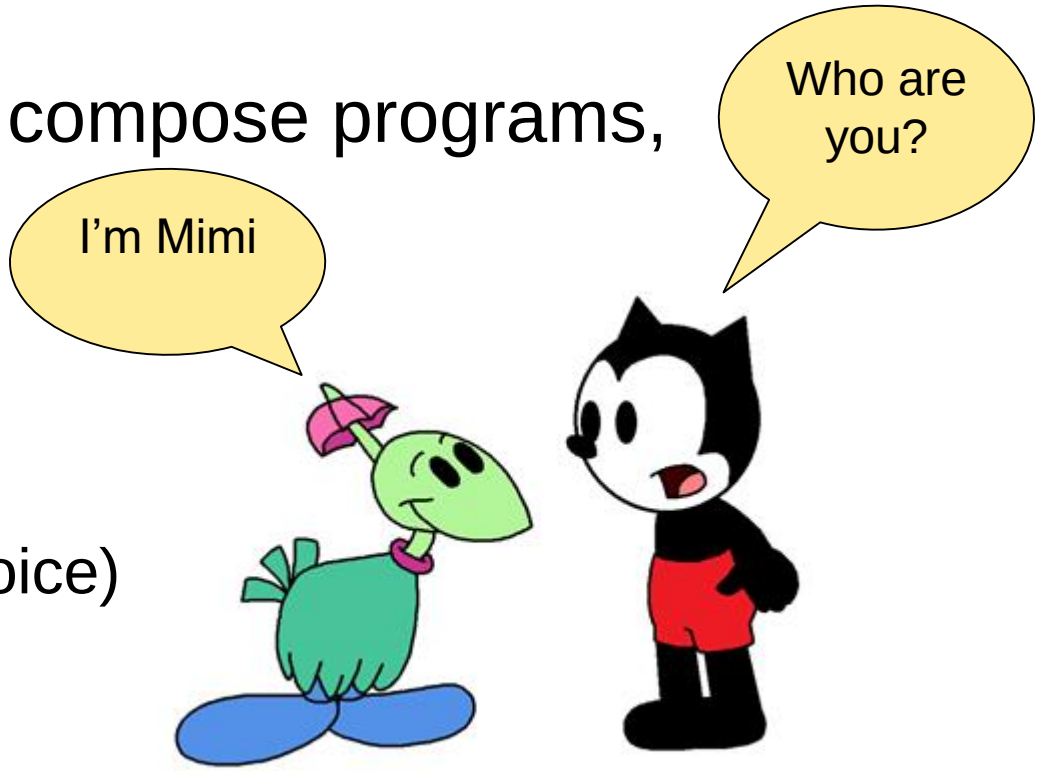
Checking values

- ! Means NOT
- && Means AND
- || Means OR

Java building blocks (for specifying behaviour)

Control structures:
constructions to compose programs,

- Sequence
- Selection (Choice)
- Repetition



Specifying behavior

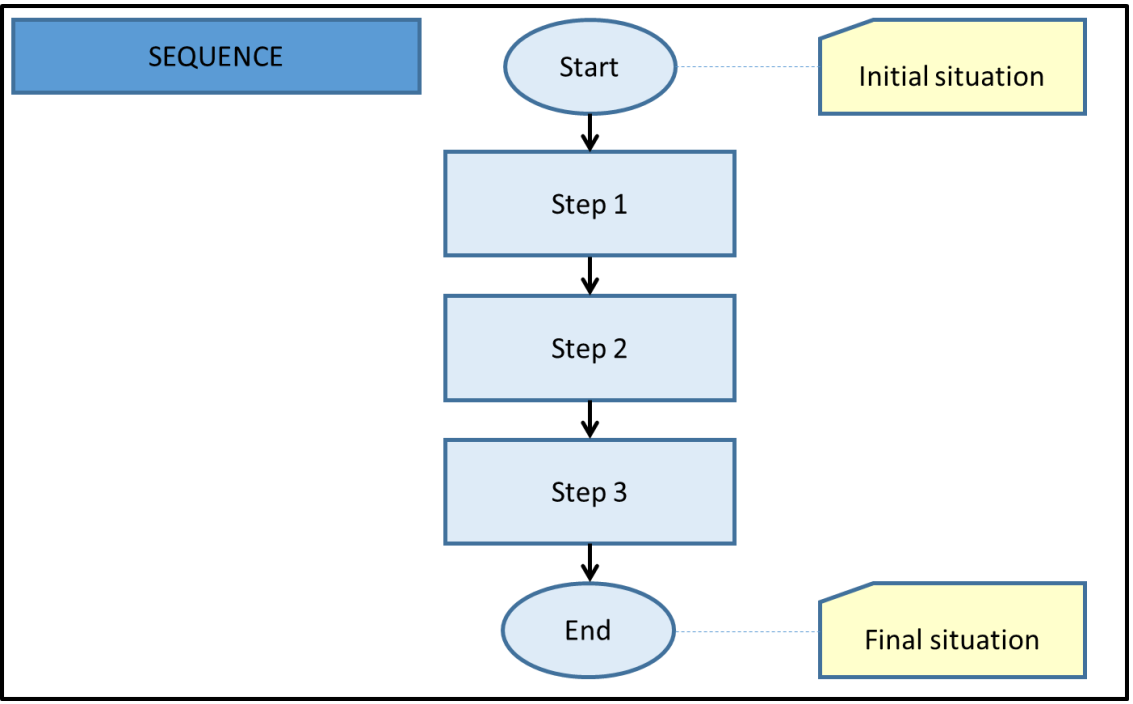
Control structures:
constructions to compose programs,

like:

- Sequence: `stepA; stepB; …`
- Selection: `if ( check() ) then stepsThen else stepsElse`
- Repetition: `while ( check() ) stepsWhile`

Sequence

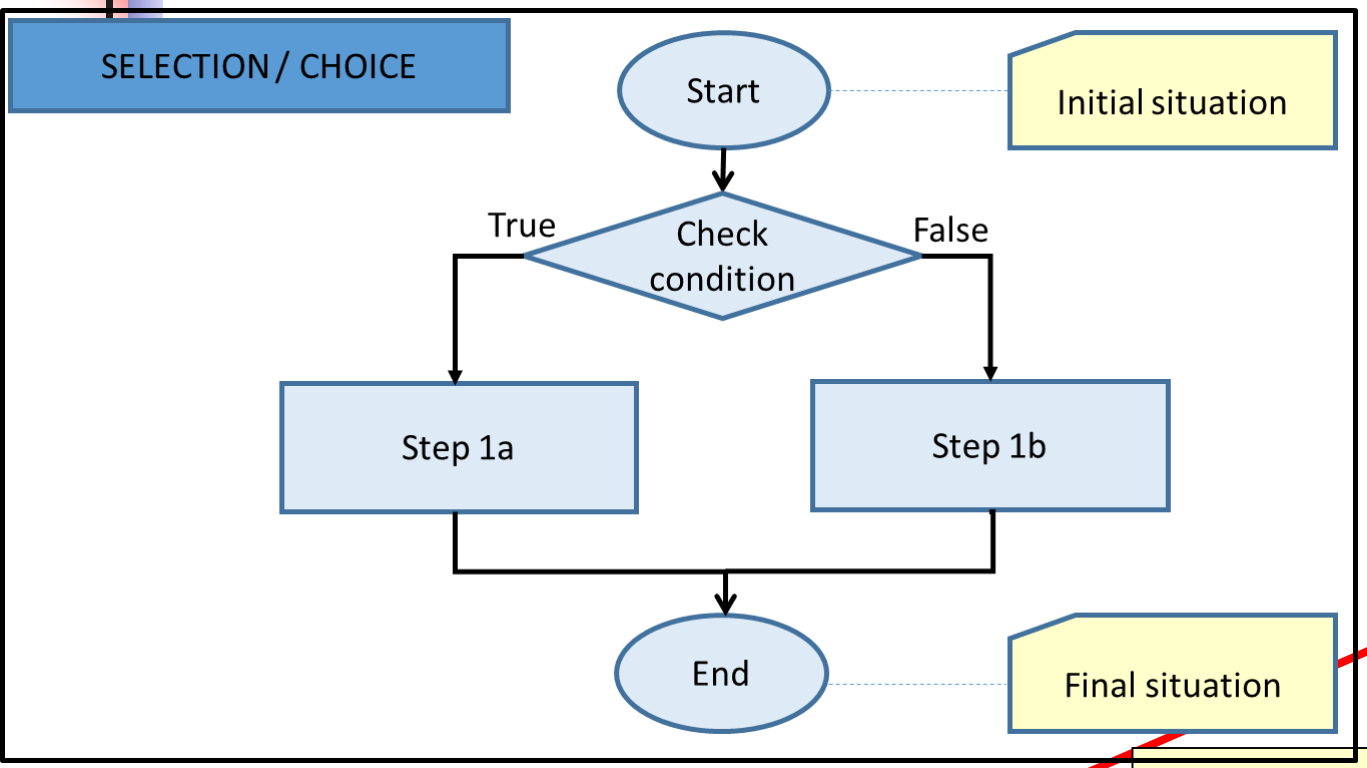
flowchart



code

```
public … methodName( … ) {
   step1();
   step2();
   step3();
}
```

Selection (choice, if..then..else)

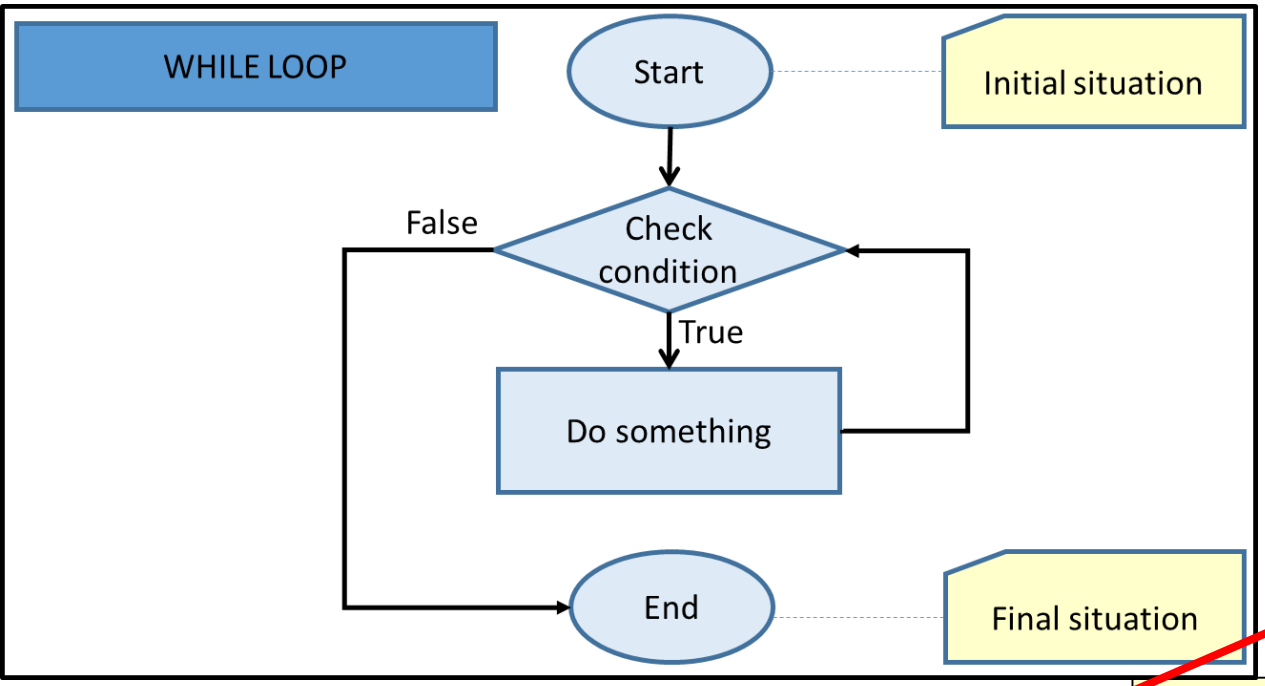


flowchart

code

```
public … methodName( … ) {
   while ( check () ) {
      doSomething ();
   }
}
```

Repetition (iteration, loop)



flowchart

code

```
public … methodName( … ) {
   while ( check () ) {
      doSomething ();
   }
}
```

Turn to North

- Algorithm
- Flowchart
- Code

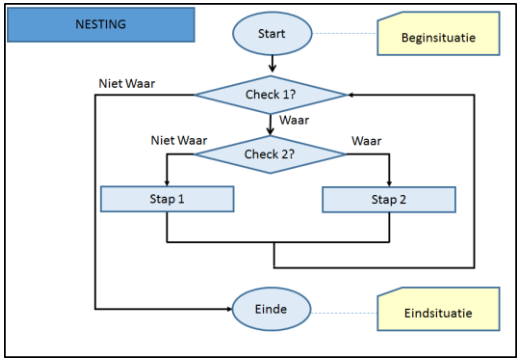
- Given the method: boolean facingNorth( )

Challenge & problem

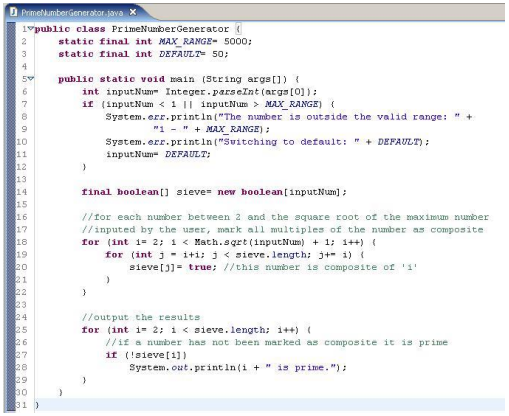
You must perform two aspects well:

1) Create a *problem-solving algorithm* (a disciplined and creative process)

We use a systematic approach



2) *Formulate* that algorithm *in terms of a programming language* (a disciplined and very precise process)

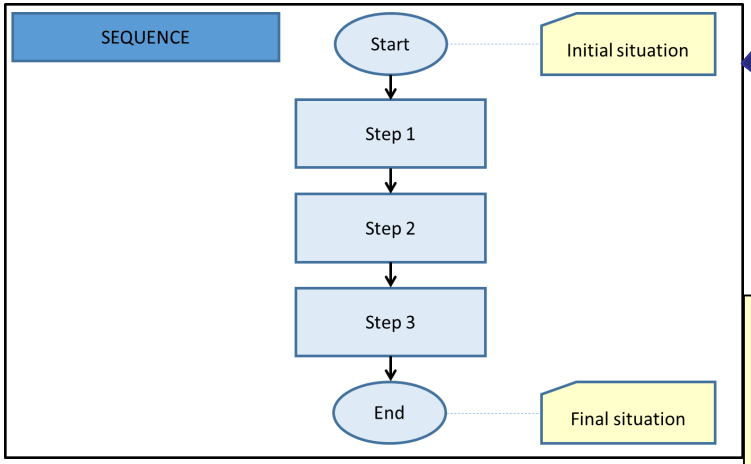


```
PrimeNumberGenerator.java
public class PrimeNumberGenerator {
    static final int MAX_RANGE= 5000;
    static final int DEFAULT= 50;

    public static void main (String args[]) {
        int inputNum= Integer.parseInt(args[0]);
        if (inputNum < 1 || inputNum > MAX_RANGE) {
            System.err.println("The number is outside the valid range: " +
                    "1 - " + MAX_RANGE);
            System.err.println("Switching to default: " + DEFAULT);
            inputNum= DEFAULT;
        }

        final boolean[] sieve= new boolean[inputNum];

        //for each number between 2 and the square root of the maximum number
        //inputed by the user, mark all multiples of the number as composite
        for (int i= 2; i < Math.sqrt(inputNum) + 1; i++) {
            for (int j = i+i; j < sieve.length; j+= i) {
                sieve[j]= true; //this number is composite of 'i'
            }
        }

        //output the results
        for (int i= 2; i < sieve.length; i++) {
            //if a number has not been marked as composite it is prime
            if (!sieve[i])
                System.out.println(i + " is prime.");
        }
    }
}
```

Always check that your algorithm is correct by running/testing the implementation!

Steps in making a solution

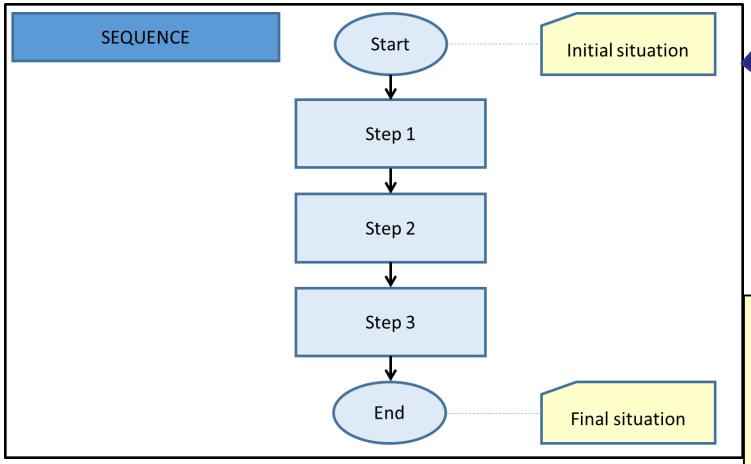
1. Think -> Algorithm
2. Flowchart
3. Code




```
public .. methodName( … ) {
   step1();
   step2();
   step3();
}
```

Debugging (fixing mistakes)

1. Remove compile errors
2. Check if code represents flowchart
3. Check if flowchart represents algorithm
4. Check for thinking-errors in your algorithm




```
public .. methodName( … ) {
   step1();
   step2();
   step3();
}
```

Modifying code

- After each MINOR adjustment
  - Compile
  - Test if it still works
- If you do too much at once, and then get an error…
  - … you're doomed to get frustrated!
- Remember, from our first lesson:
  - Expect to make mistakes!

Questions?

Computational thinking

- **Working in a structured manner:**
  - Breaking problems down into subproblems
  - Design, solve and test solutions to subproblems
  - Combing these (sub)solutions to solve problem
- **Analyzing** the quality of a solution
- **Reflecting** about the solution chosen and proces
- **Generalizing** and re-use of existing solutions

Wrapping up

Homework for Wednesday 8:30 December 9th:

- Assignment 2:
  - All ex. incl diagnostic test 6.1 and 6.2
  - **Renske.weeda@gmail.com**
- Assignment 3: up to and including 5.1

- **ZIP code** and 'IN' and **email** to
- Flowcharts: on paper in **pigeonhole** or photo/scan and paste into document
- **All course downloads on: http://www.cs.ru.nl/~sjakie/Greenfoot/Kandinsky/**
- Next week: Quiz
- Reflection/Evaluation

Wrapping up

- Quiz: what to expect?
  - Assignment 1 & 2
  - Difference between accessor/mutator methods
  - Signature of a method (incl parameters, results)
  - Types (such as int, boolean, String, void)
  - Inheritance (class diagram)
  - Explain flowcharts: sequence, selection, repetition
  - Transform an algorithm into flowchart
- Reflection/evaluation: tips/tops