Algorithmic Thinking and Structured Programming (in Greenfoot)

Teachers:

Renske Smetsers

Sjaak Smetsers

Today’s Lesson plan (3)

- 10 min Looking back
  - What did we learn last week?

- Blocks of:
  - Theory
  - Exercises

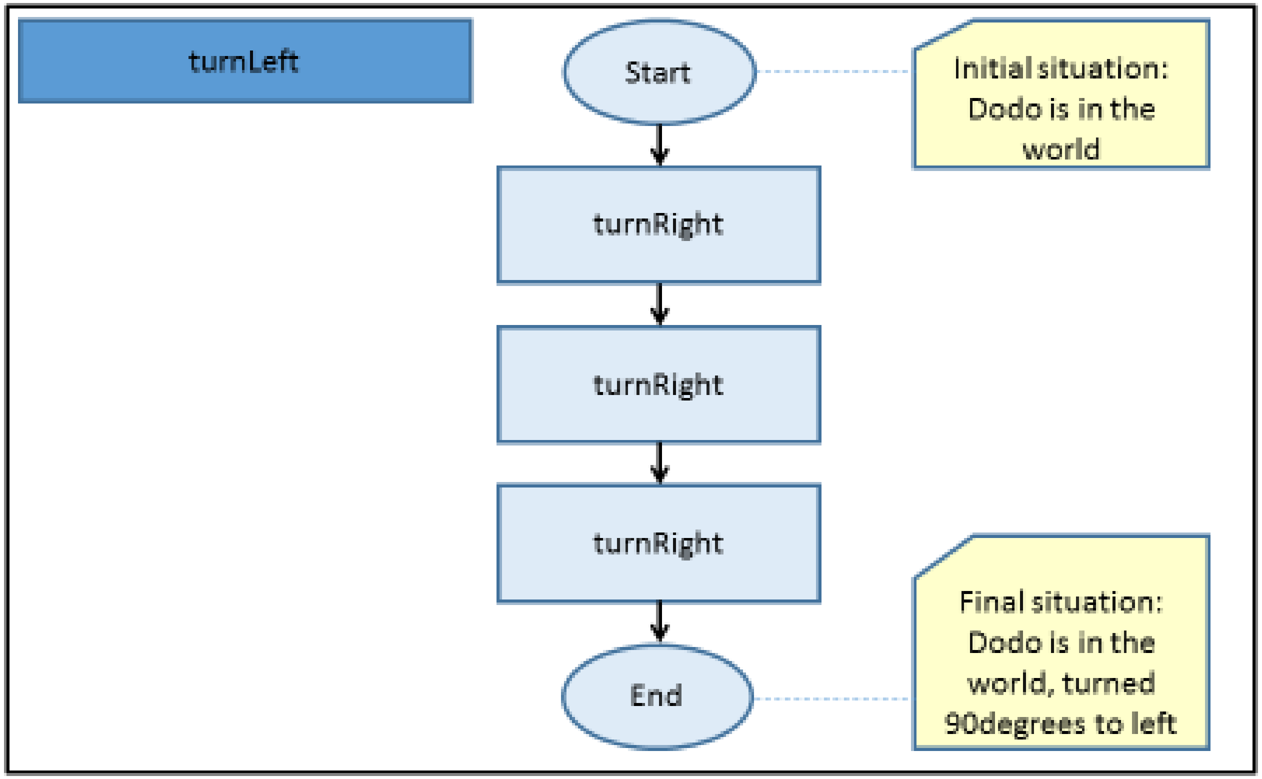
- Course exercises and discuss problems / homework

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

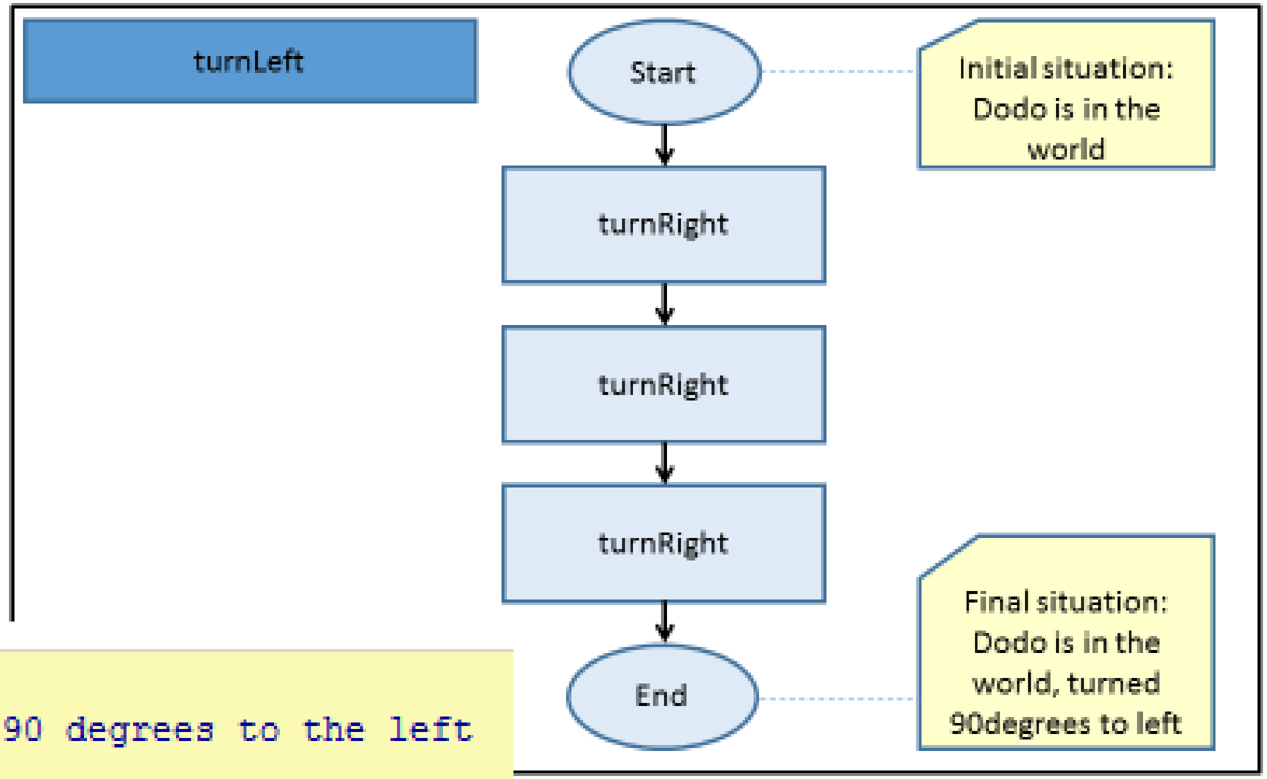
Retrospective

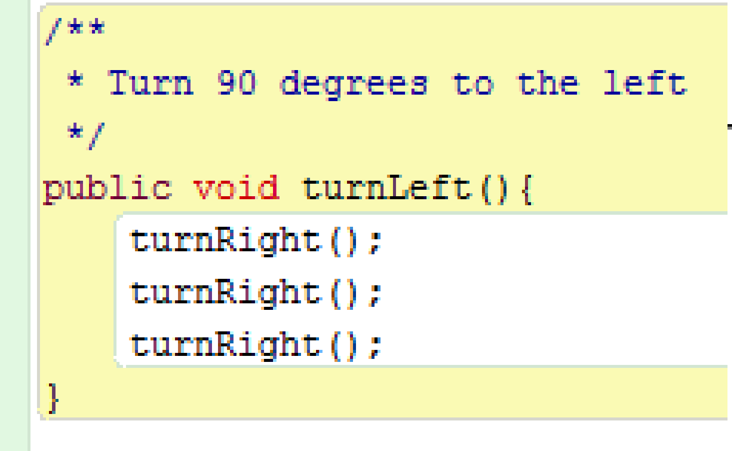
- Parameters, signatures, method calls, results
- Mutator / accessor methods
- Getter / Setter methods
- Flowcharts

Retrospective: sequence

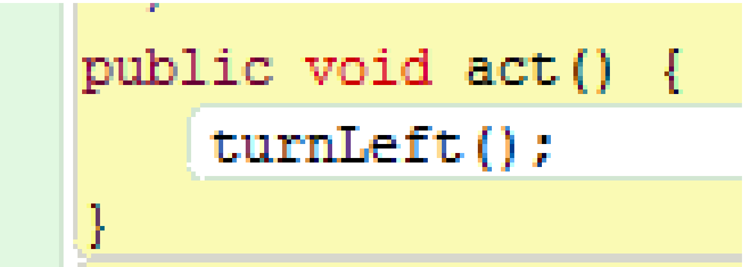

turnLeft
Start
Initial situation: Dodo is in the world
turnRight
turnRight
turnRight
End
Final situation: Dodo is in the world, turned 90degrees to left

Retrospective: sequence


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



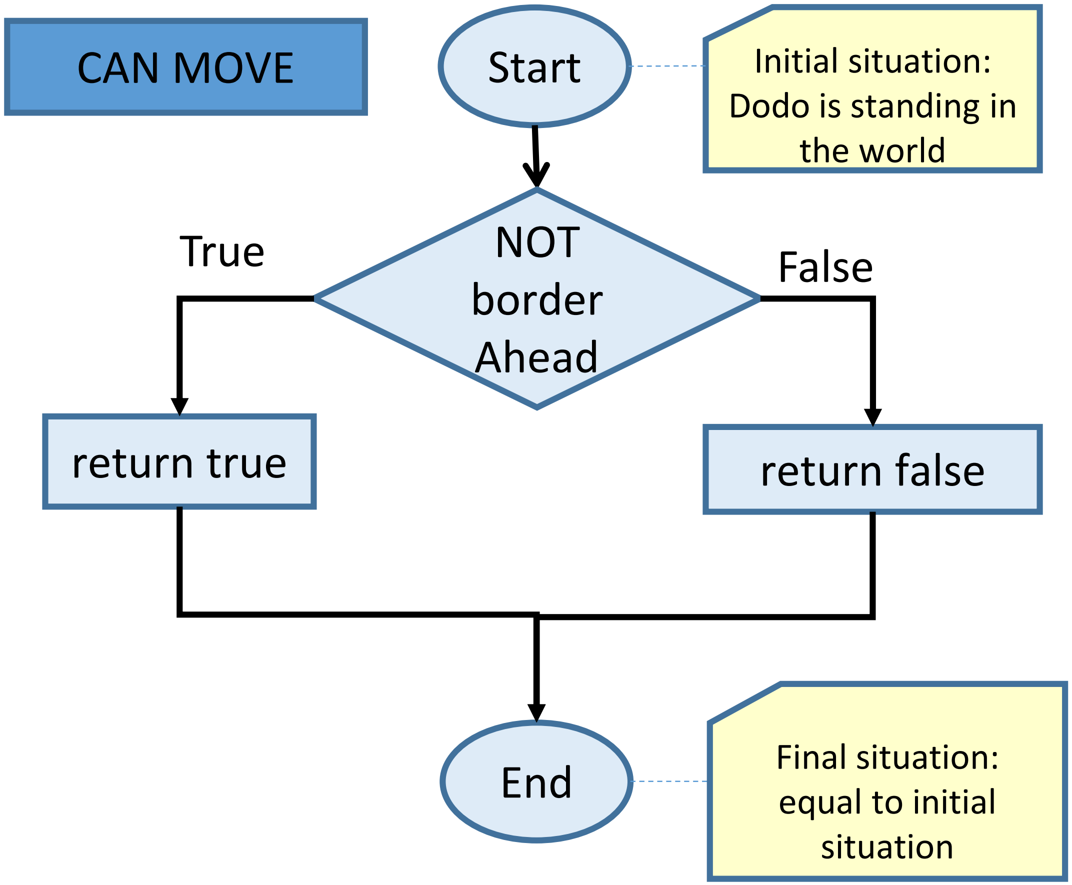
```
/**
 * Turn 90 degrees to the left
 */
public void turnLeft(){
    turnRight();
    turnRight();
    turnRight();
}
```



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

Accessor methods (questions)

flowchart

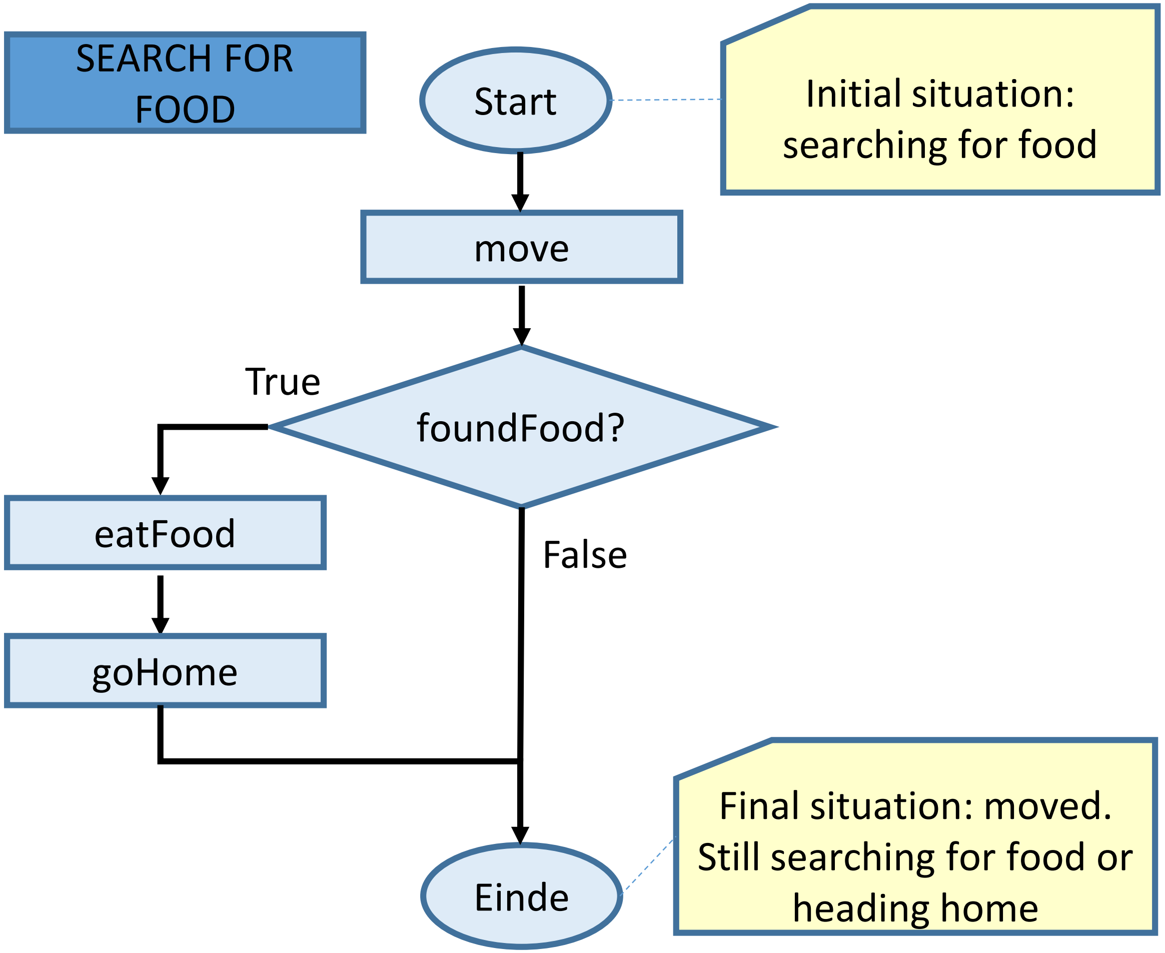

CAN MOVE
Start
Initial situation: Dodo is standing in the world
NOT border Ahead
True
False
return true
return false
End
Final situation: equal to initial situation

code

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

Mutator methods (behavior)

flowchart



SEARCH FOR FOOD

Start

Initial situation: searching for food

move

foundFood?

True

eatFood

goHome

False

Einde

Final situation: moved. Still searching for food or heading home

code

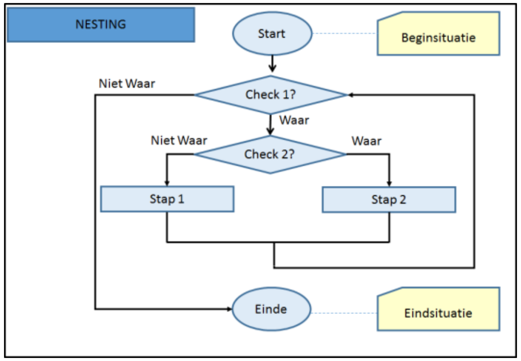
```
public void act() {
    move();
    if ( foundFood() ) {
       eatFood();
       goHome();
    }
}
```

Challenge & problem

You must perform two aspects well:

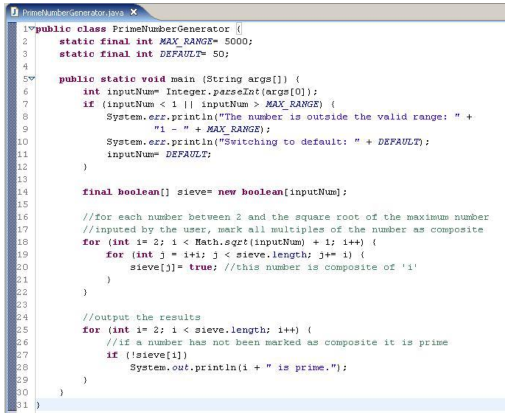
1) Create a *problem-solving algorithm* (a disciplined and creative process)

We use a systematic approach



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

We use Java



```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.");
        }
    }
}
```

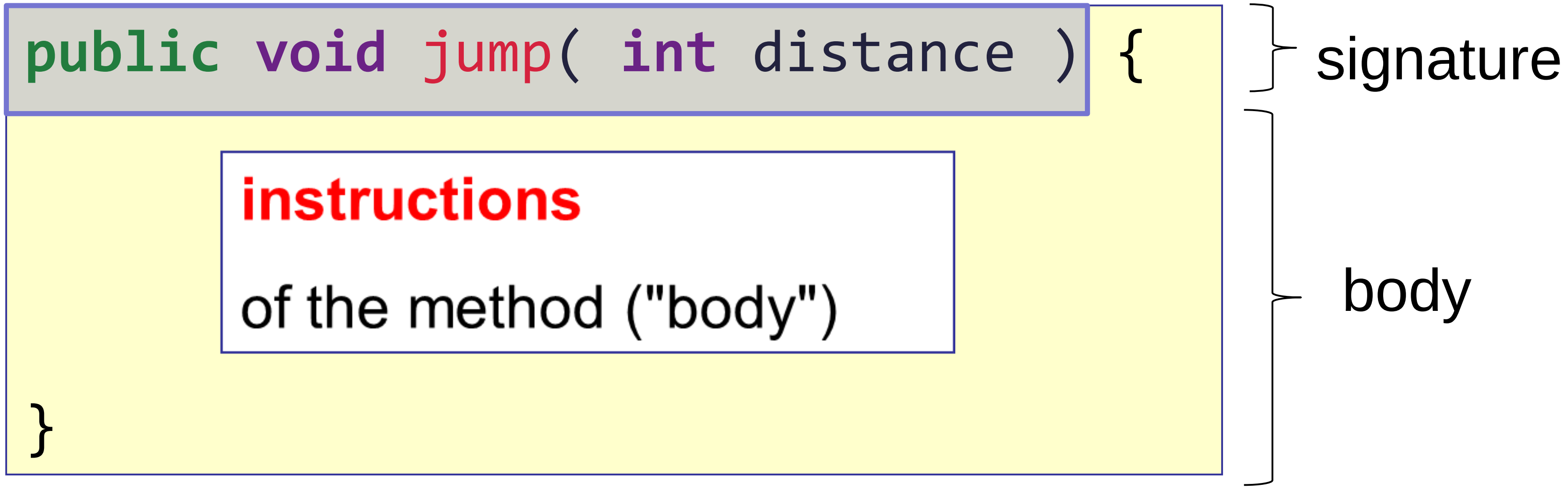
Always check that your algorithm is correct by running/testing the implementation!

Computational thinking

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

Anatomy of a method (1)

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



```
public void jump( int distance ) {

        instructions

        of the method ("body")

}
```

signature

body

Anatomy of a method (2)

Name of this method

```
public void jump( int distance ) {

        instructions

        of the method ("body")

}
```

Anatomy of a method (3)

What type of result (value) is returned?

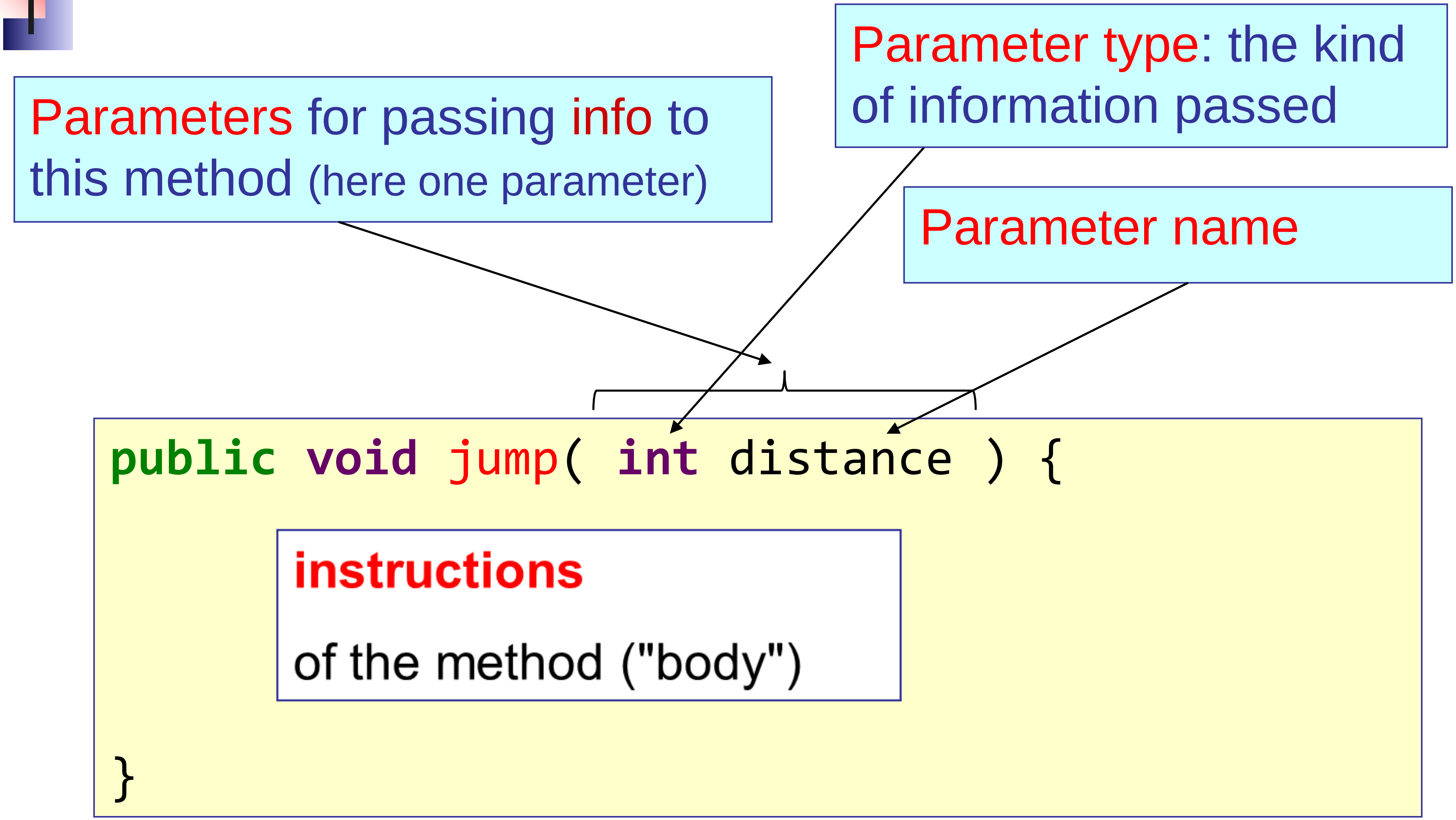
void = nothing returned

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 (4)



Parameters for passing info to this method (here one parameter)

Parameter type: the kind of information passed

Parameter name

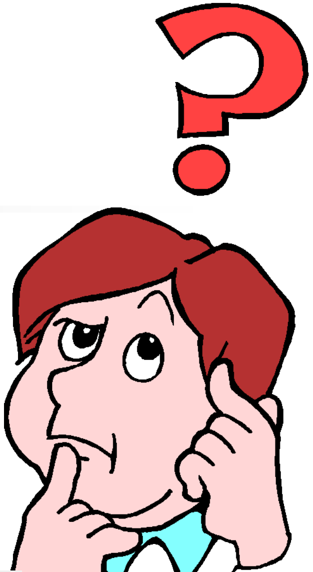
```
public void jump( int distance ) {
```

instructions

of the method ("body")

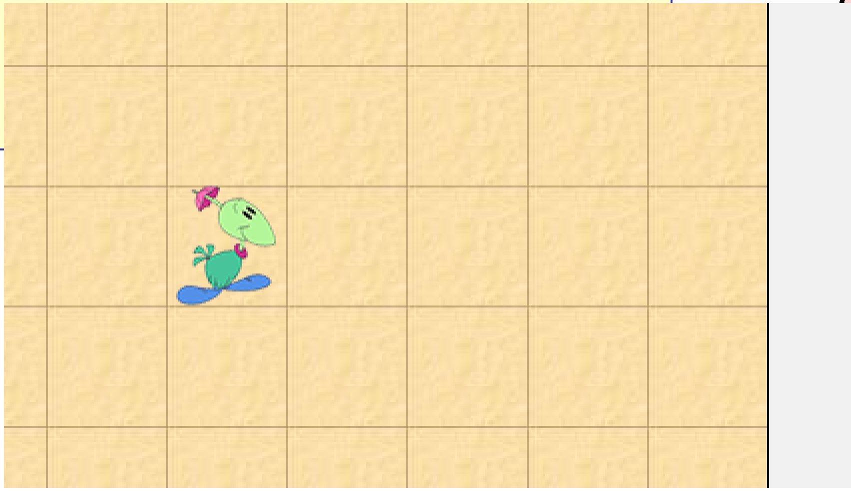
```
}
```

Anatomy of a method (5)



Return a boolean (true or false)

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



Getter and setter methods

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

- Getter vs. setter methods
- 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-Oriented class design

Student
Data: `double moneyInWallet`
Methods: `double countMoneyInWallet ( )`

Class has data and methods

MyDodo

Data:
`int nrOfEggsLaid`
`int nrEggsToHatch`

Methods:
`int getNrOfEggsLaid ( )`
`void setNrOfEggsToHatch ( int nrEggsToHatch )`

Getter vs. Setter methods

- `int getNrOfEggsLaid ( )`
  - **Question:** "*Dodo, please tell me how many eggs you have laid*"
  - **Effect:** Dodo **returns** the number of eggs laid (int)

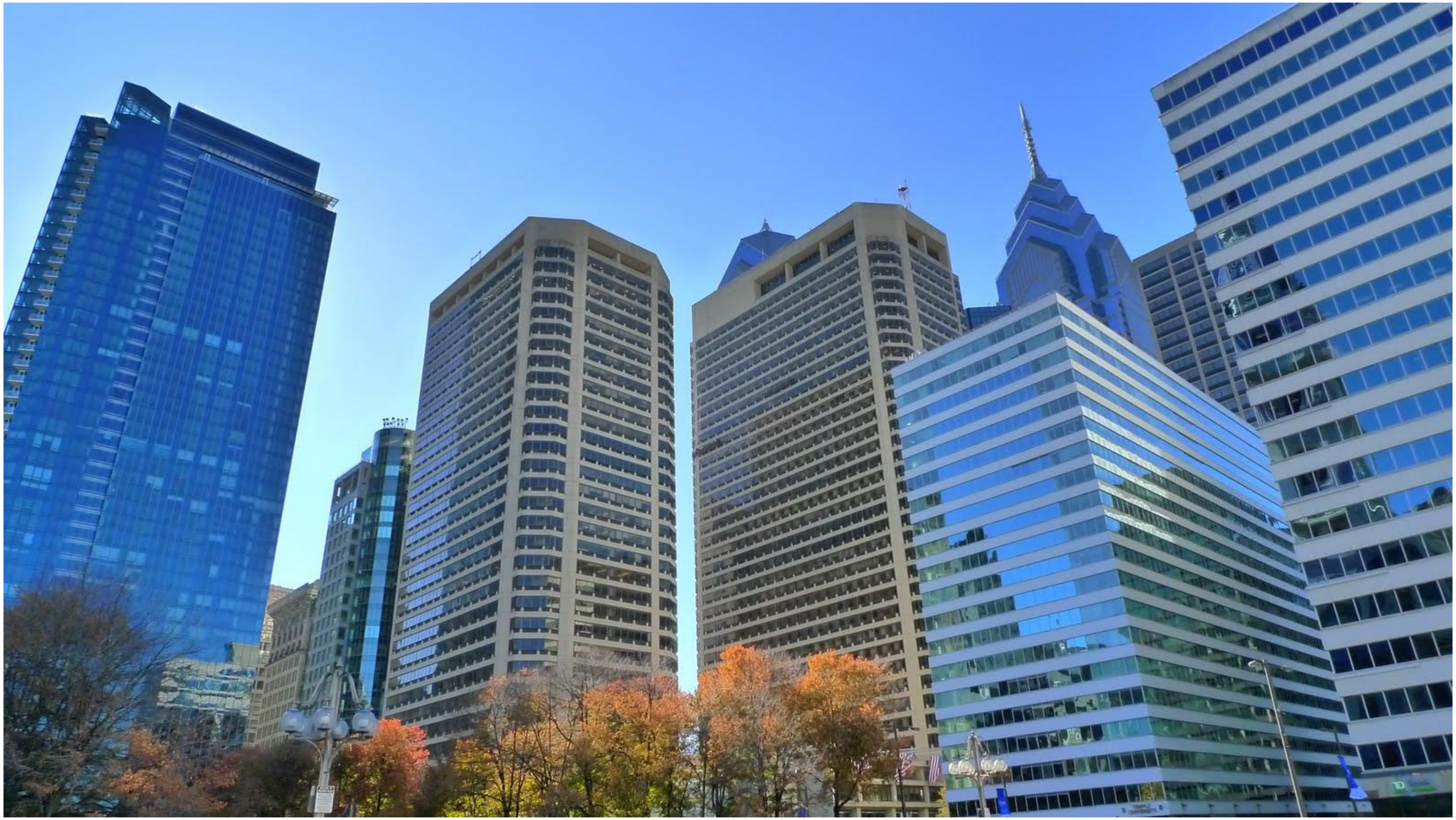
- `void setNrOfEggsToHatch (int nrEggsToHatch)`
  - **Statement:** "*Dodo, this is the number of eggs you have to hatch*"
  - **Effect:** Dodo changes her **data** so that she remembers (or **stores**) this new amount.

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

Any questions so far?

Counting floors



Counting

- Starts at….

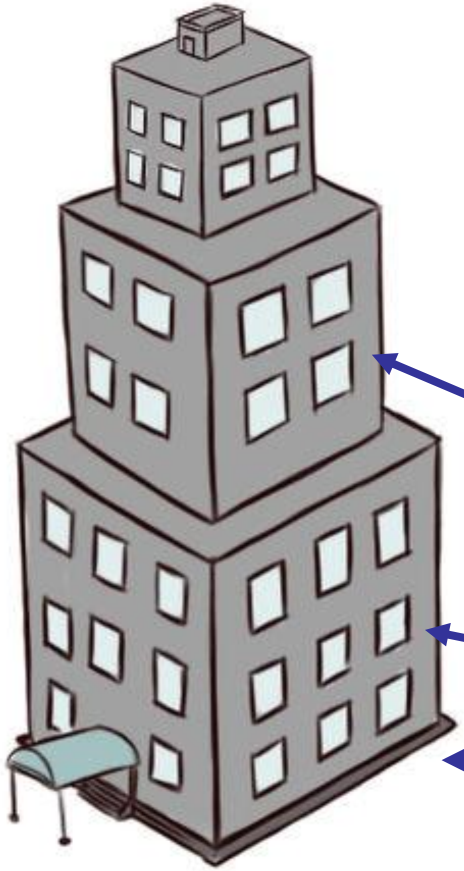
NL

3e verdieping

…

1e verdieping

0: Begane Grond



wikiHow

USA

4th floor

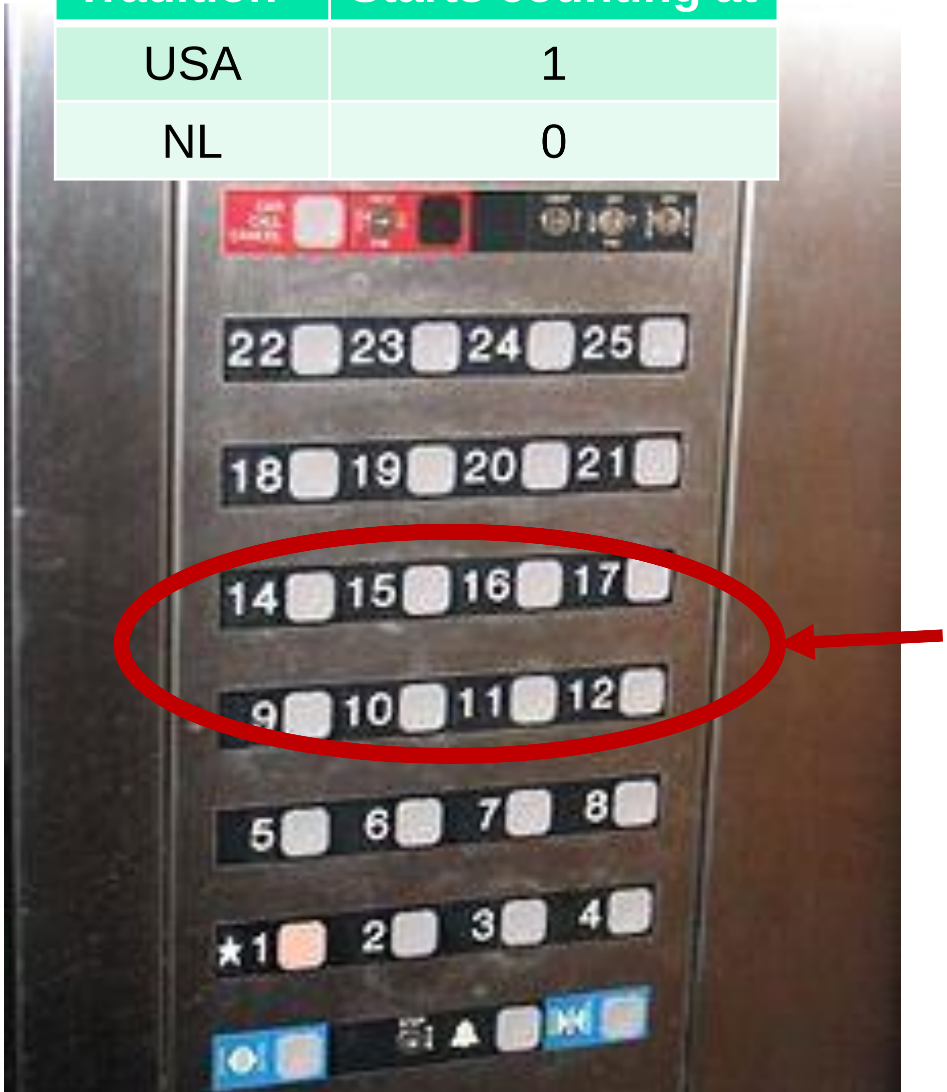
…

2nd floor

First floor

Counting starts at…

Tradition	Starts counting at
USA	1
NL	0



US tradition: skip 13th floor

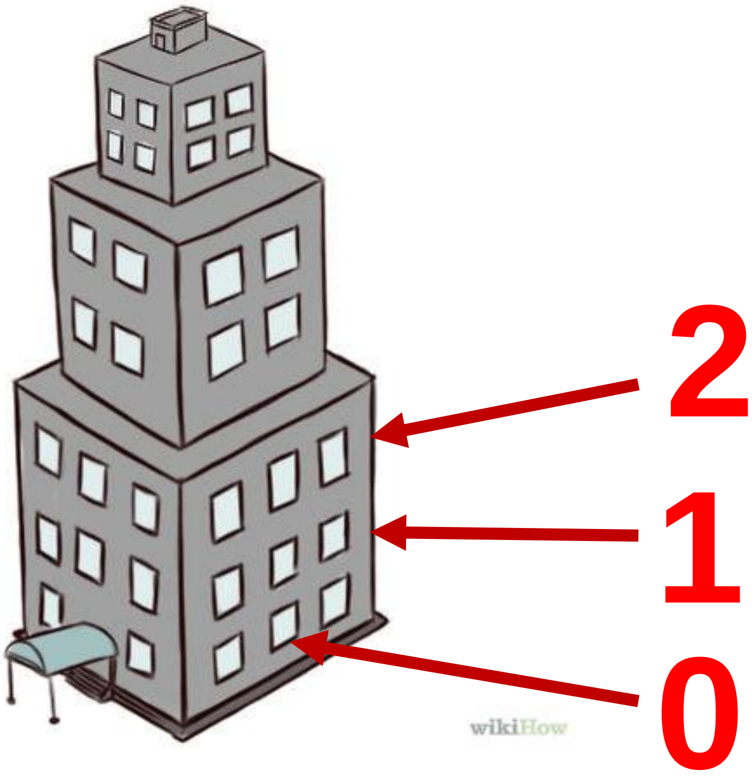
Counting starts at…

Tradition	Starts counting at
USA	~~1~~
NL	0

Tradition	Starts counting at
Maths	~~1~~
Comp. Science	0

~~US tradition: skip 13th floor~~

Start counting at 0!!!



2

1

0

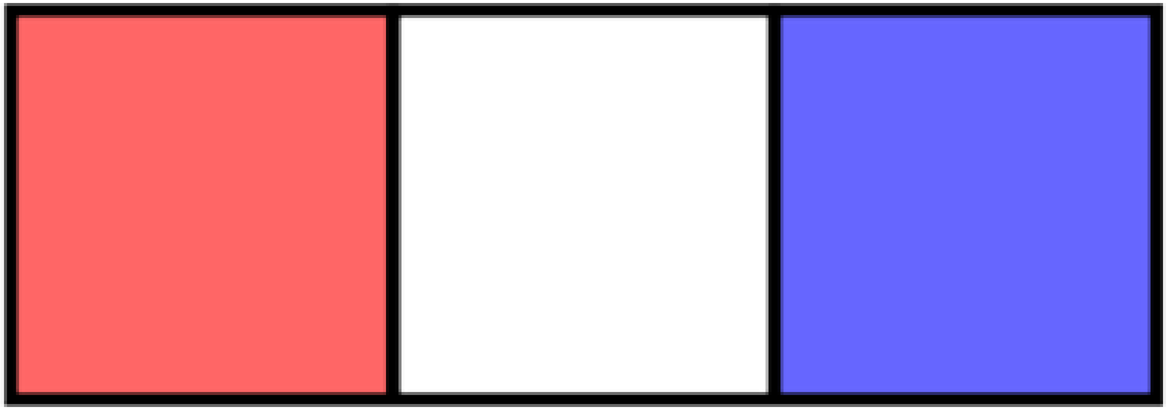
Unplugged: Swap puzzle

What it’s about:

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

Swap Puzzle level 1

Square **0** Square **1** Square **2**



Swap Puzzle

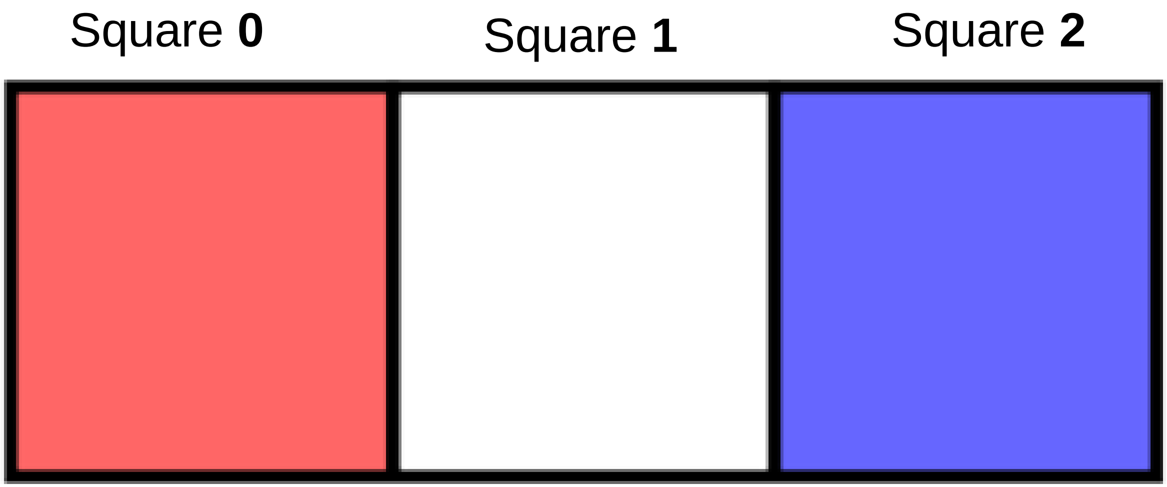
- Pieces start on different (non-white) color
- A piece can move to an empty adjacent square
- Can jump over an adjacent piece of another color onto an empty square

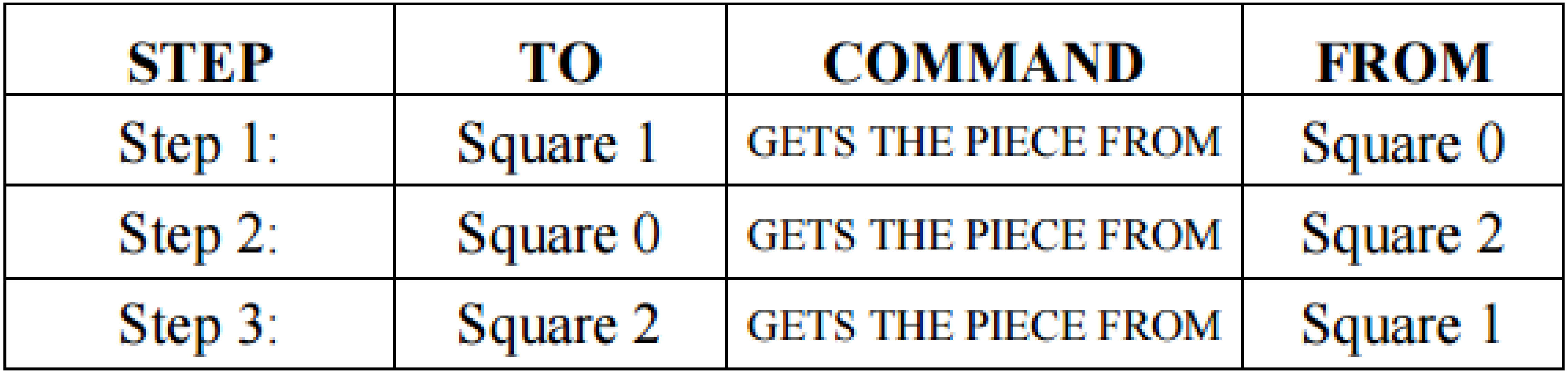
- Method to use: **getsThePieceFrom**

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

- **Goal:** Solve the puzzle in the least amount of steps

Swap Puzzle level 1

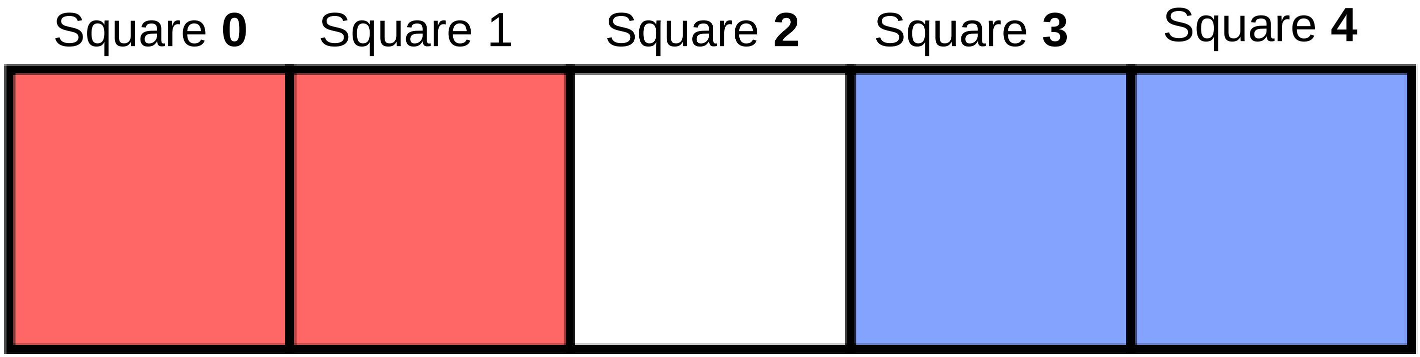




STEP	TO	COMMAND	FROM
Step 1:	Square 1	GETS THE PIECE FROM	Square 0
Step 2:	Square 0	GETS THE PIECE FROM	Square 2
Step 3:	Square 2	GETS THE PIECE FROM	Square 1

Swap Puzzle level 2

- A piece can move to an empty adjacent square
- Can jump over an adjacent piece of **another** color onto an empty square
- **Goal**: Solve the puzzle in the **least** amount of steps
- Write down the steps
- In 5 minutes: compare and share algorithms



Square **0** Square 1 Square **2** Square **3** Square **4**

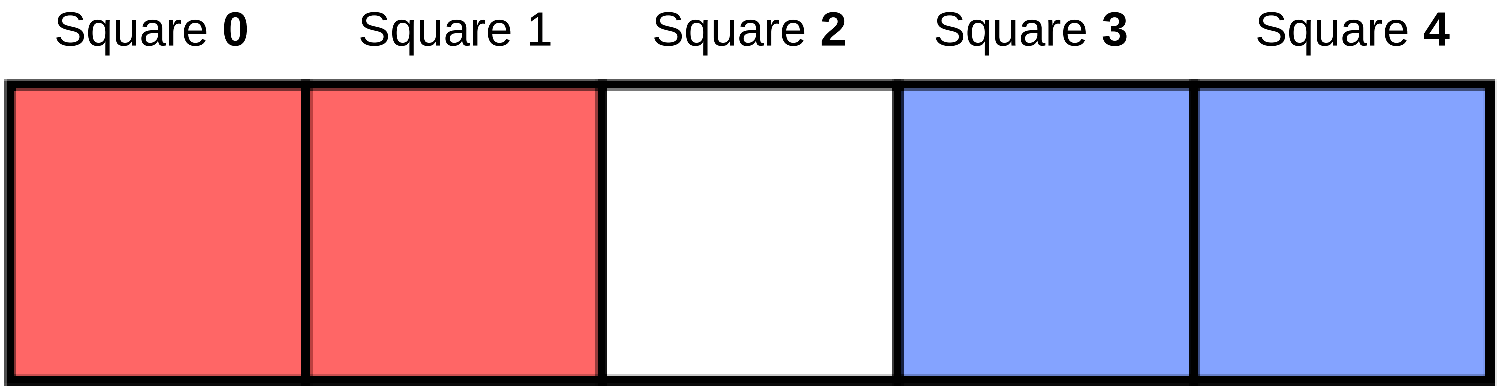
- Method to use: **getsThePieceFrom**

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

Swap Puzzle level 2

Challenge: Most **efficient** algorithm

- What to count / how to compare efficiency?
- How do you know that your algorithm works?



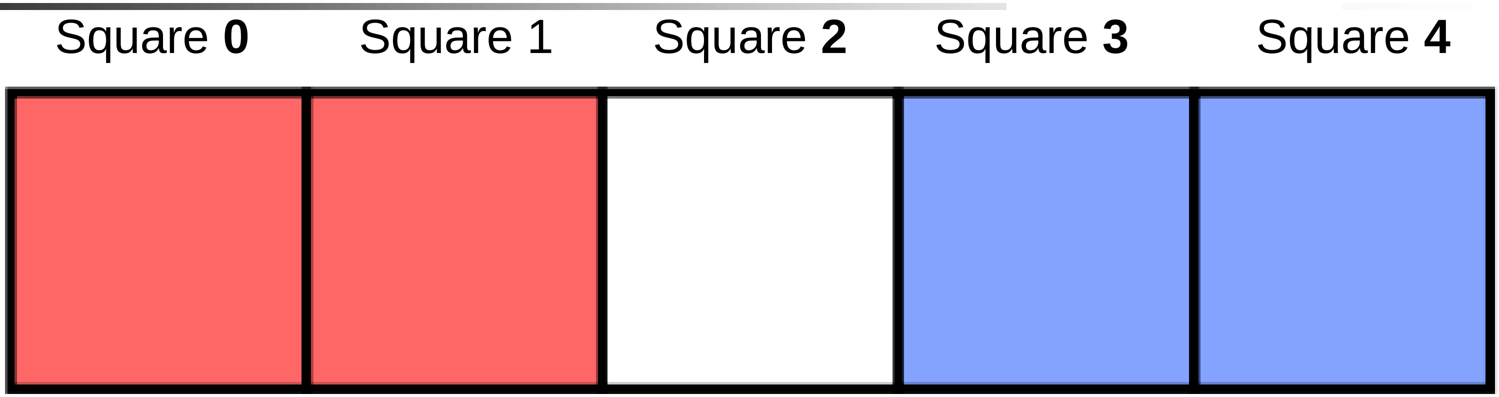
Square **0** Square 1 Square **2** Square **3** Square **4**

SQUARE 2 GETS PIECE FROM SQUARE 1

Can be simplified to (Java code):

```
        square2 = square1;
```

Swap Puzzle level2



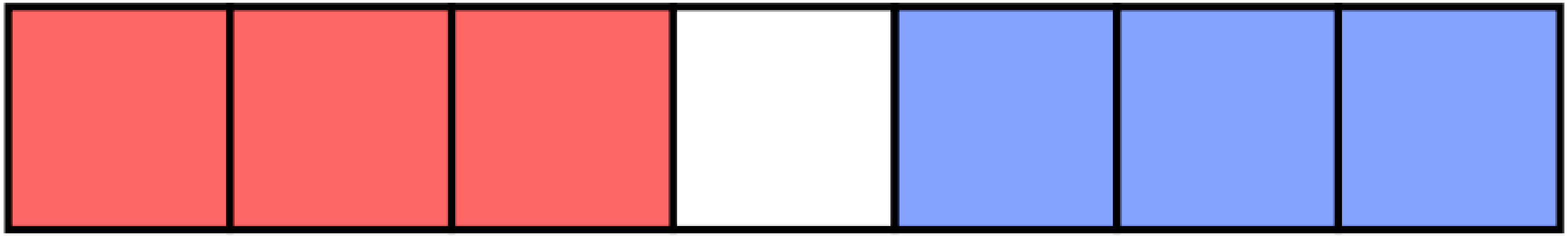
Square **0** Square 1 Square **2** Square **3** Square **4**

Can be solved in 8 moves:

```
Move1: square2 = square1; // sq2 gets piece from sq1
Move2: square1 = square3; // sq1 gets piece from sq3
Move3: square3 = square4; // sq3 gets piece from sq4
Move4: square4 = square2; // sq4 gets piece from sq2
Move5: square2 = square0; // sq2 gets piece from sq0
Move6: square0 = square1; // sq0 gets piece from sq1
Move7: square1 = square3; // sq1 gets piece from sq3
Move8: square3 = square2; // sq3 gets piece from sq2
```

Swap Puzzle level 3

- Can you come up with the most efficient algorithm?



- Answer will be revieled next week!

Swap puzzle: what its about

- **Describing** your steps => **algorithm** !!
  - Specific series of actions to get the job done
  - Write down 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, then check for **smarter** solution
- **Looking ahead** vs. trail and error
  - Look ahead and consider all possible moves
  - Necessary for efficient result with complex problem

Swap-puzzle and assigning values

Assigning values using =

- square 1 **gets** the value of square 2
- **Set** square1 **to** (value of) square 2
- In Java code: **square1 = square2;**

Check value using ==

- **Does** square 1 **have** the value of … ?
- In Java, to **check** if square1 **is / has** redPiece:

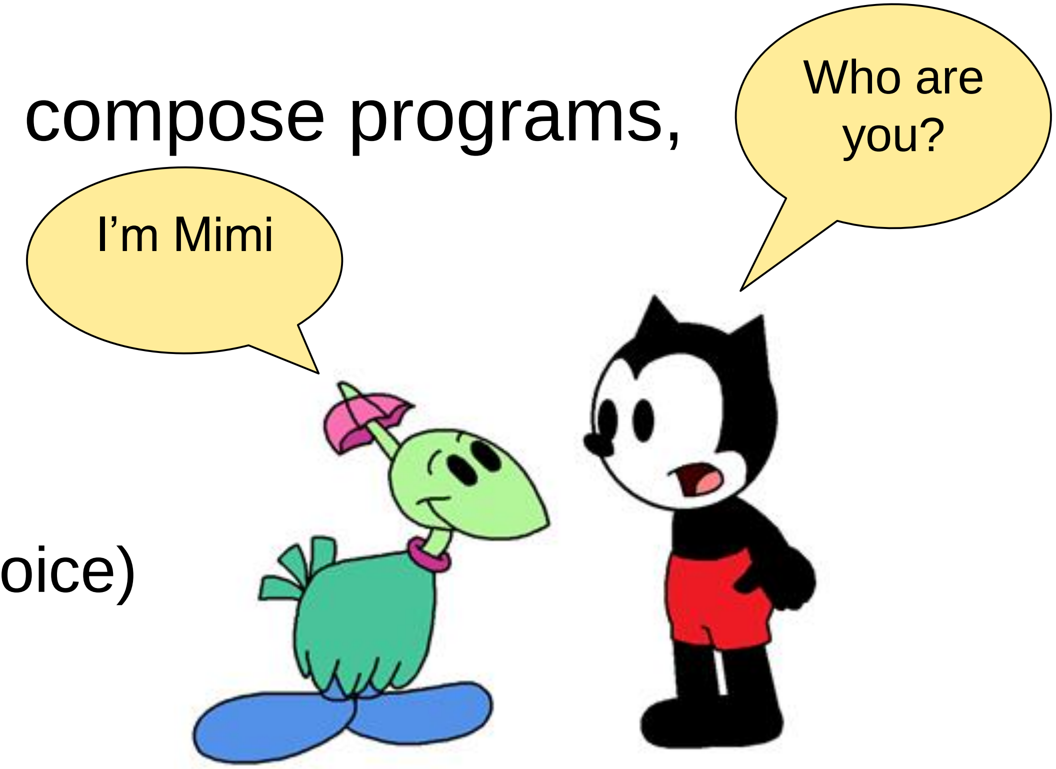
```
if ( square1 == redPiece ) {
        …
}
```

Checking values

- **==** means EQUALS TO
  - recall: ‘=’ means ‘gets value’ or ‘becomes’
- **!** Means NOT
- **&&** Means AND
- **||** Means OR

Java building blocks (for specifying behaviour)

Control structures:
constructions to compose programs,



- □ Sequence
- □ Selection (Choice)
- □ Repetition

Specifying behavior

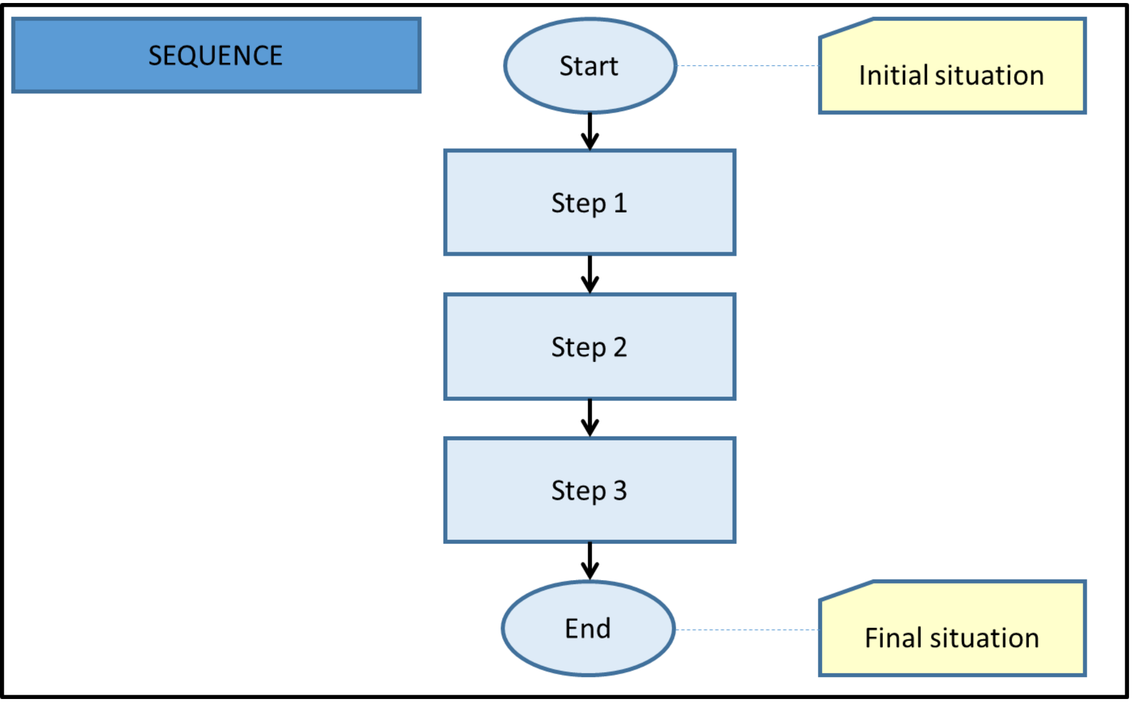
Control structures:
constructions to compose programs

like:

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

Sequence

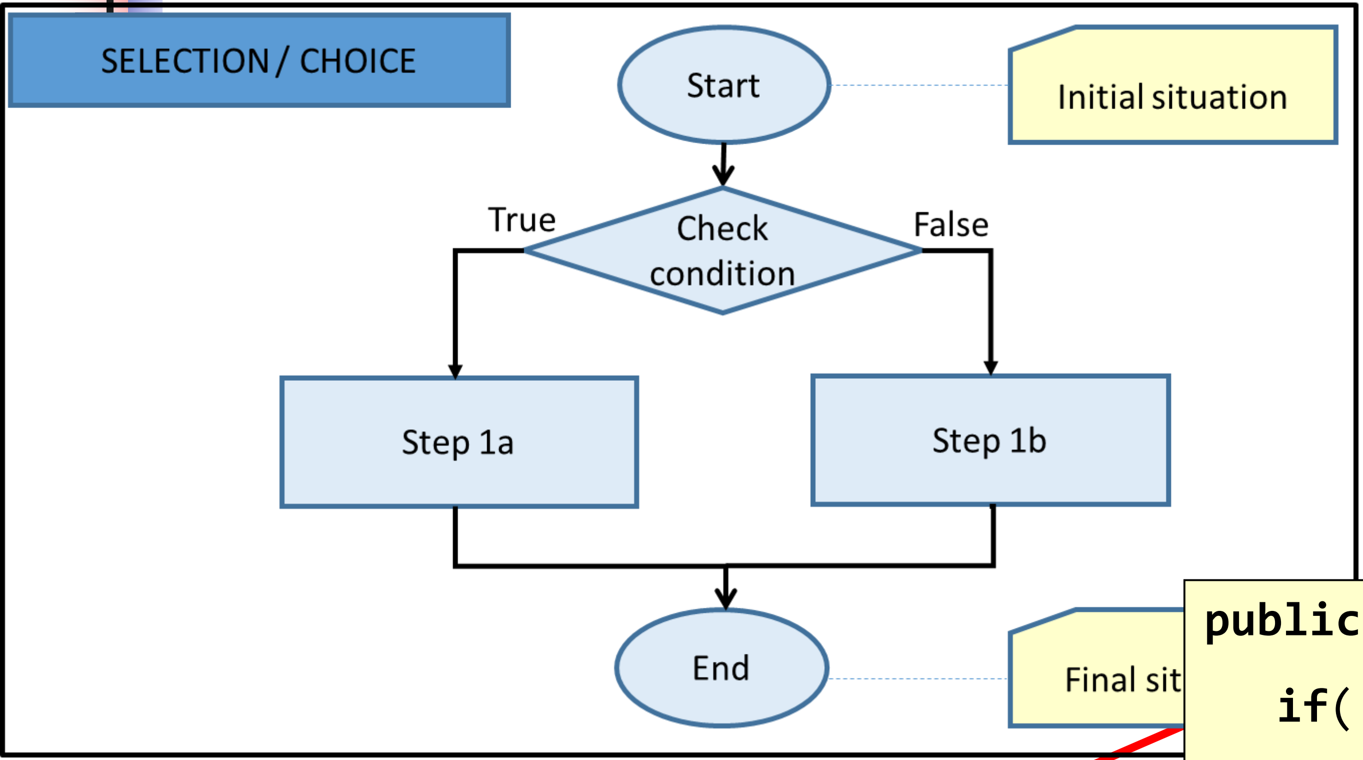
flowchart



code

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

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

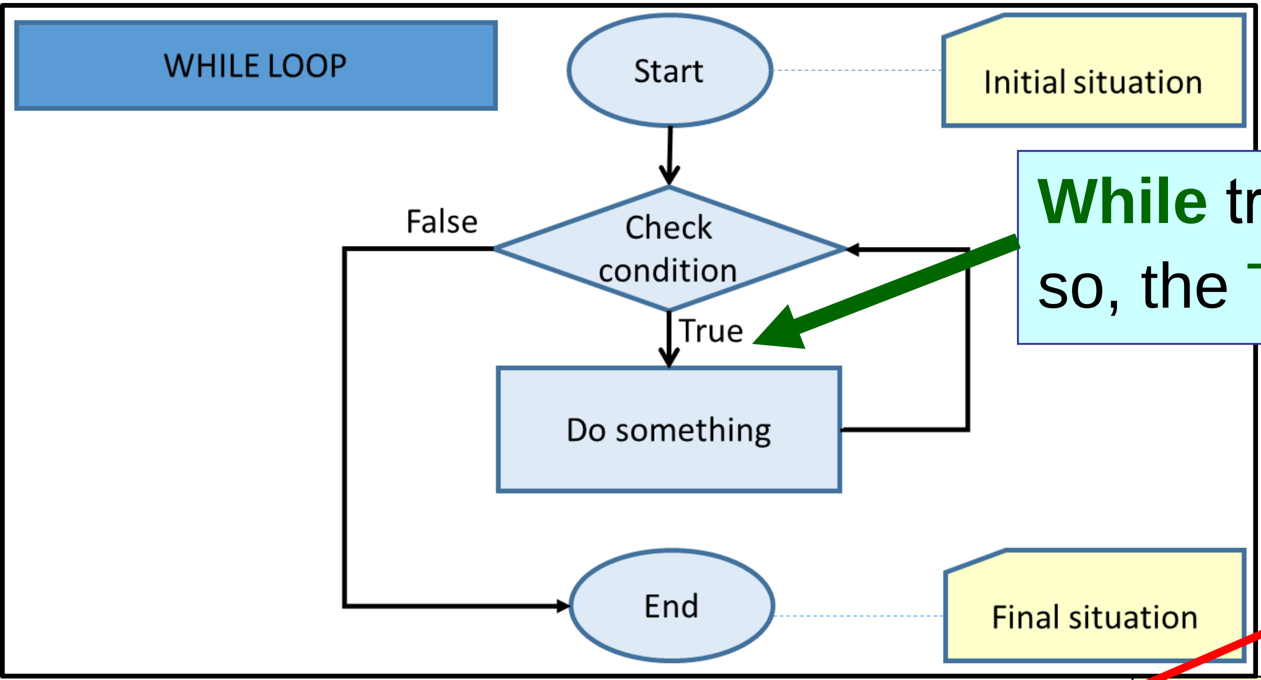


flowchart

code

```
public … methodName( … ) {
   if( check () ) {
      step1a();
   }else{
      step1b();
   }
}
```

Repetition (iteration, loop) – WHILE



While true, then repeat…
so, the TRUE-part is **repeated**

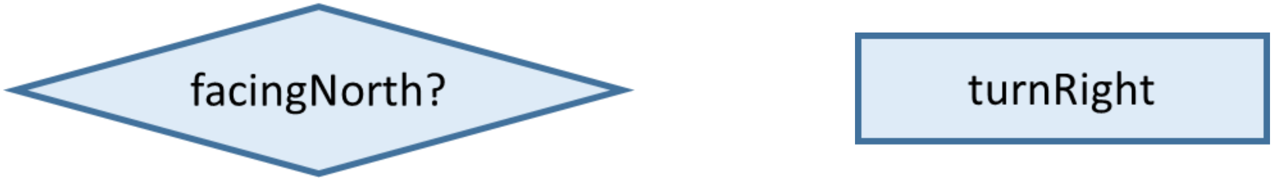
flowchart

code

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

Turn facing North- using if

Assume you may use the methods:



facingNorth?

turnRight

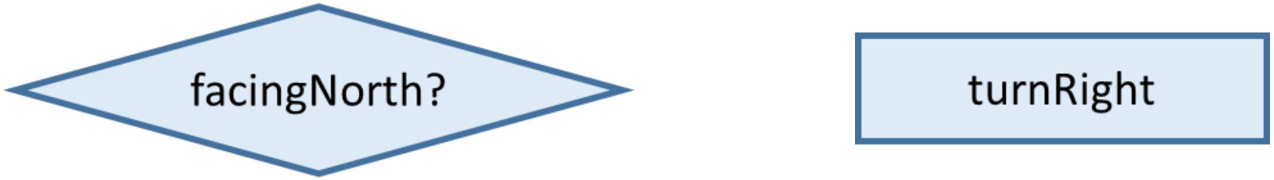
How to turn so that facing North?

1. Algorithm (in words)
2. Flowchart
3. Code

Draw a flowchart using (only) if statements

Turn facing North – using while

Assume you may use the methods:



facingNorth?

turnRight

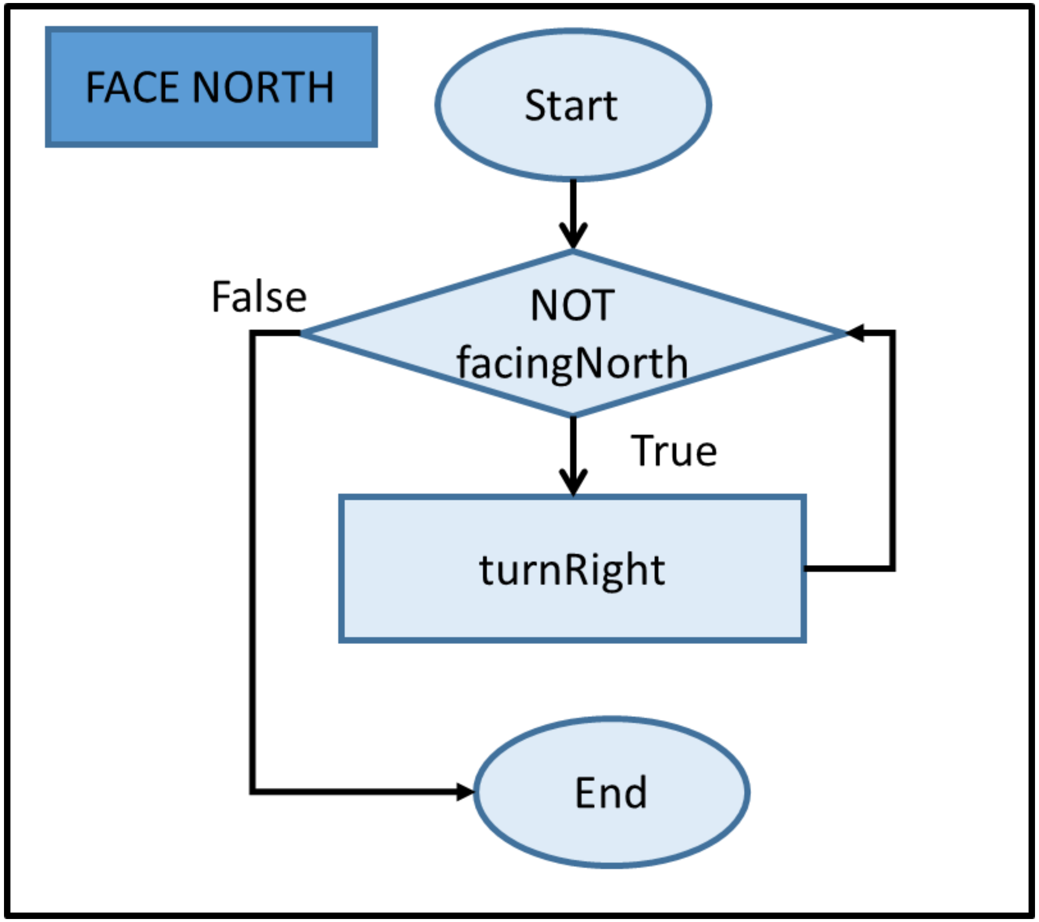
How to turn so that facing North?

1. Algorithm (in words)
2. Flowchart
3. Code

Draw a flowchart using a while

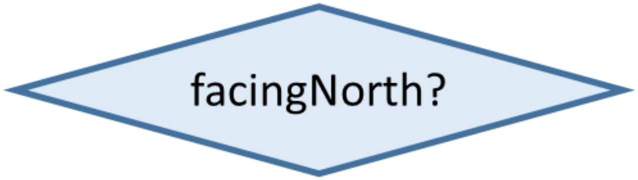
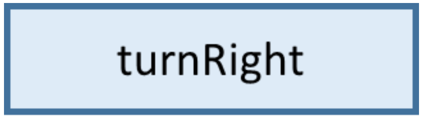
Turn facing North – using while

- Why is this solution more elegant / preferable?



Turn facing North – alg into code

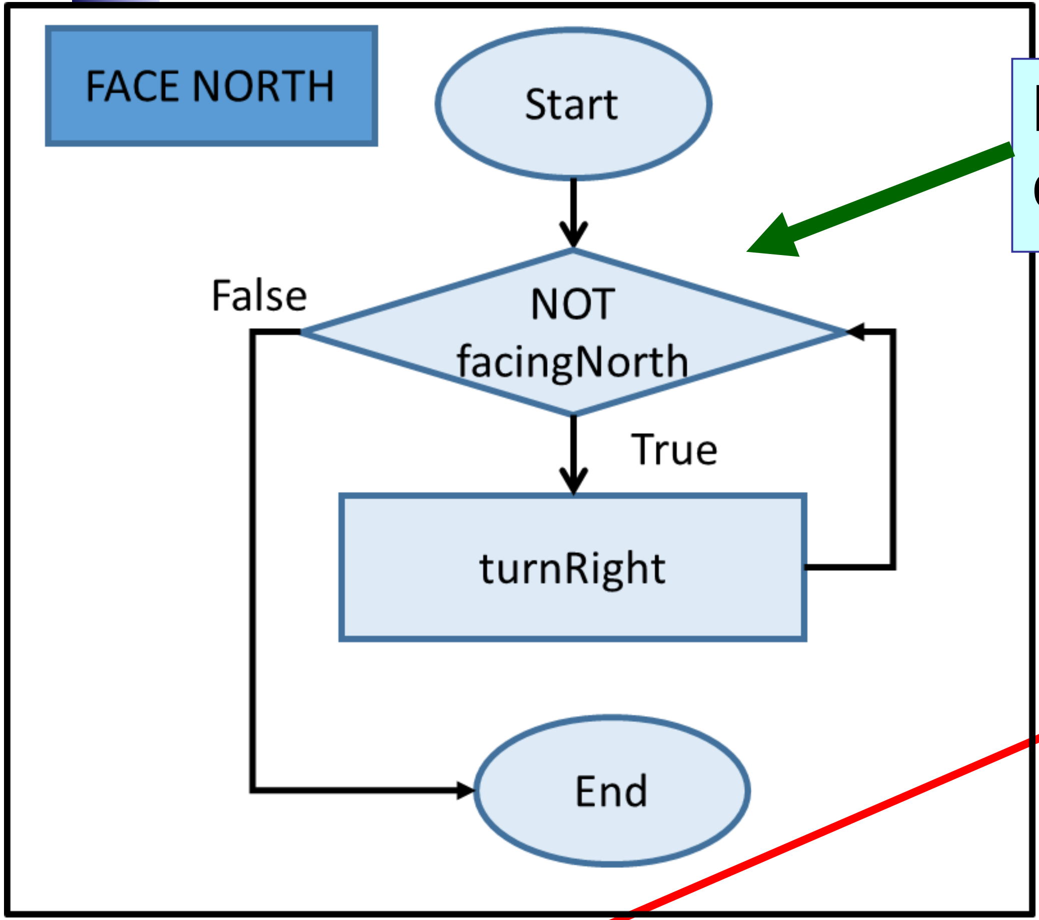
Assume you may use the methods:

Flowchart	Code
 facingNorth?	`boolean facingNorth ( )`
 turnRight	`void turnRight ( )`

How to turn so that facing North?

1. Algorithm (in words)
2. Flowchart
3. Code

Flowchart -> code



Note: often a ‘**NOT**’ is used in condition (just as in words)

flowchart

code

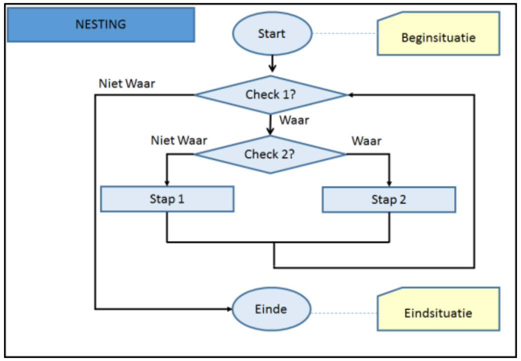
```
public void faceNorth( ) {
   while ( ! facingNorth ) {
      turnRight ();
   }
}
```

Challenge & problem

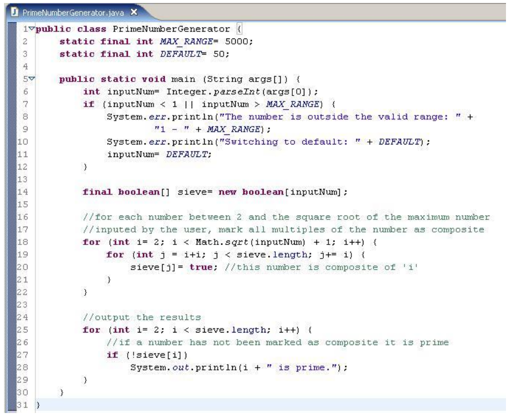
You must perform two aspects well:

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)

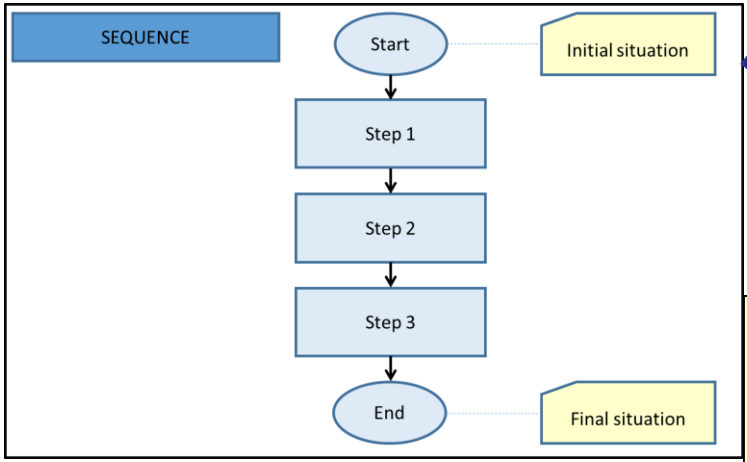


```java
public class PrimeNumberGenerator {
    static final int MAX_RANGE= 5000;
    static final int DEFAULT= 50;

    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 creating a solution

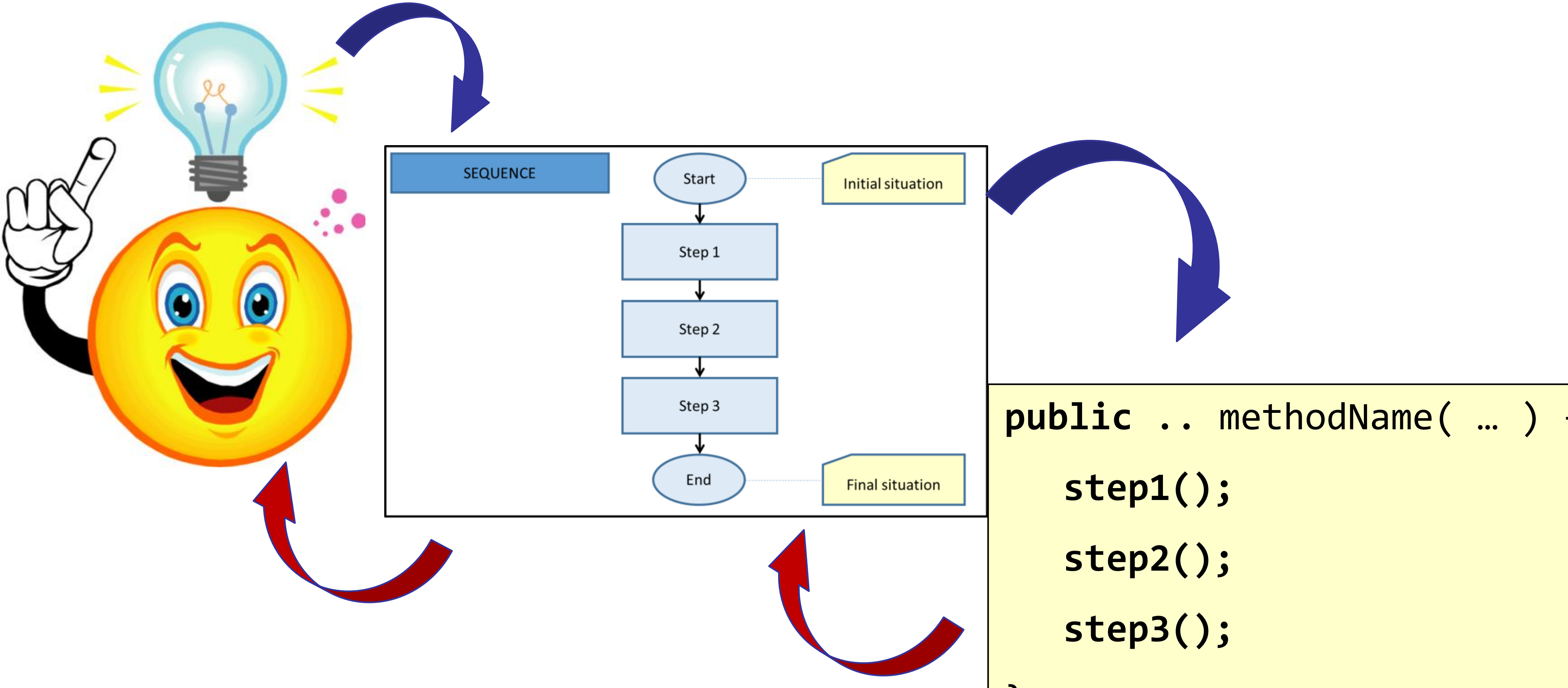
1. Think ➡ Algorithm
2. Flowchart
3. Code



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

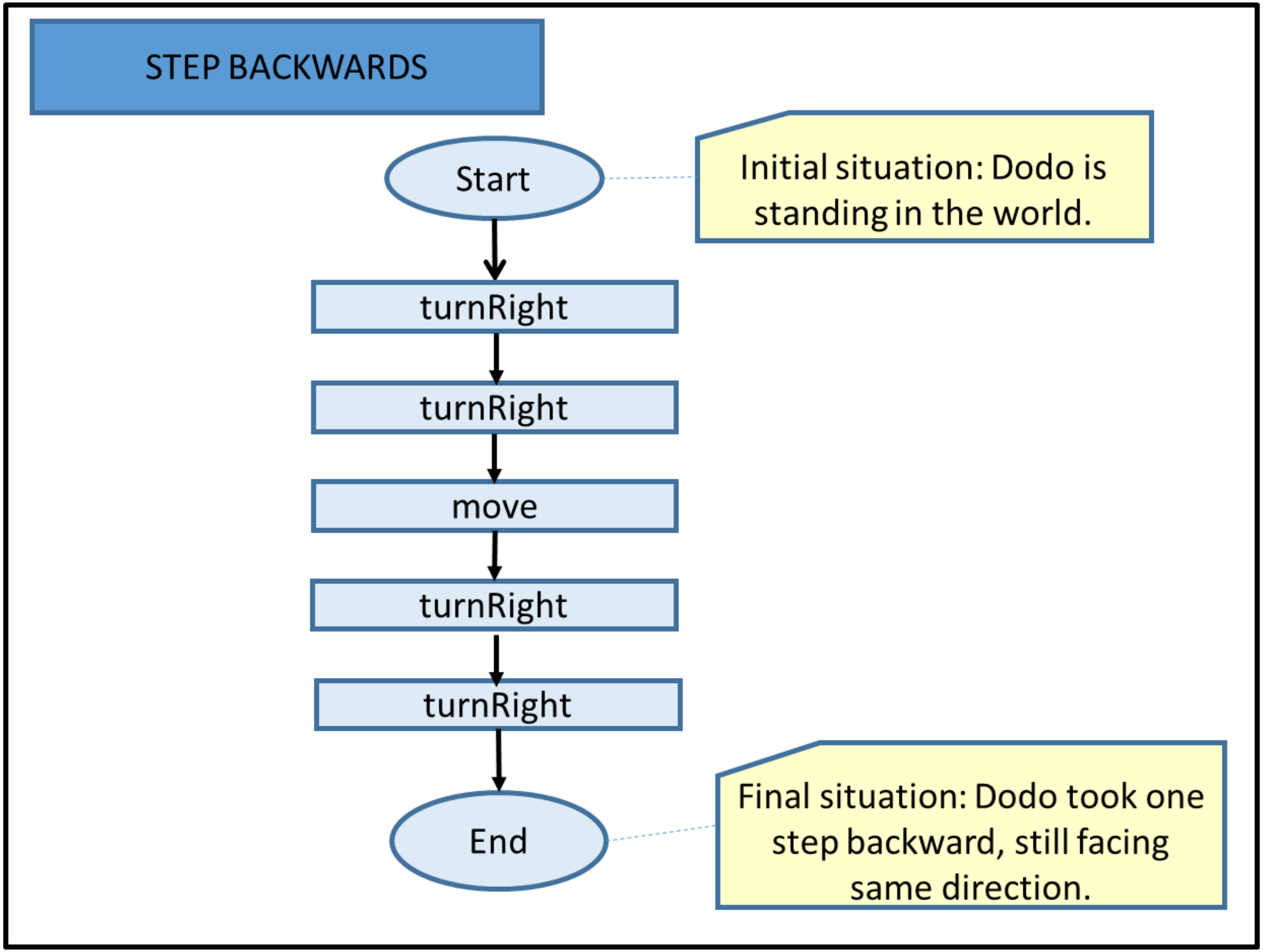
Debugging (fixing mistakes)

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();
```

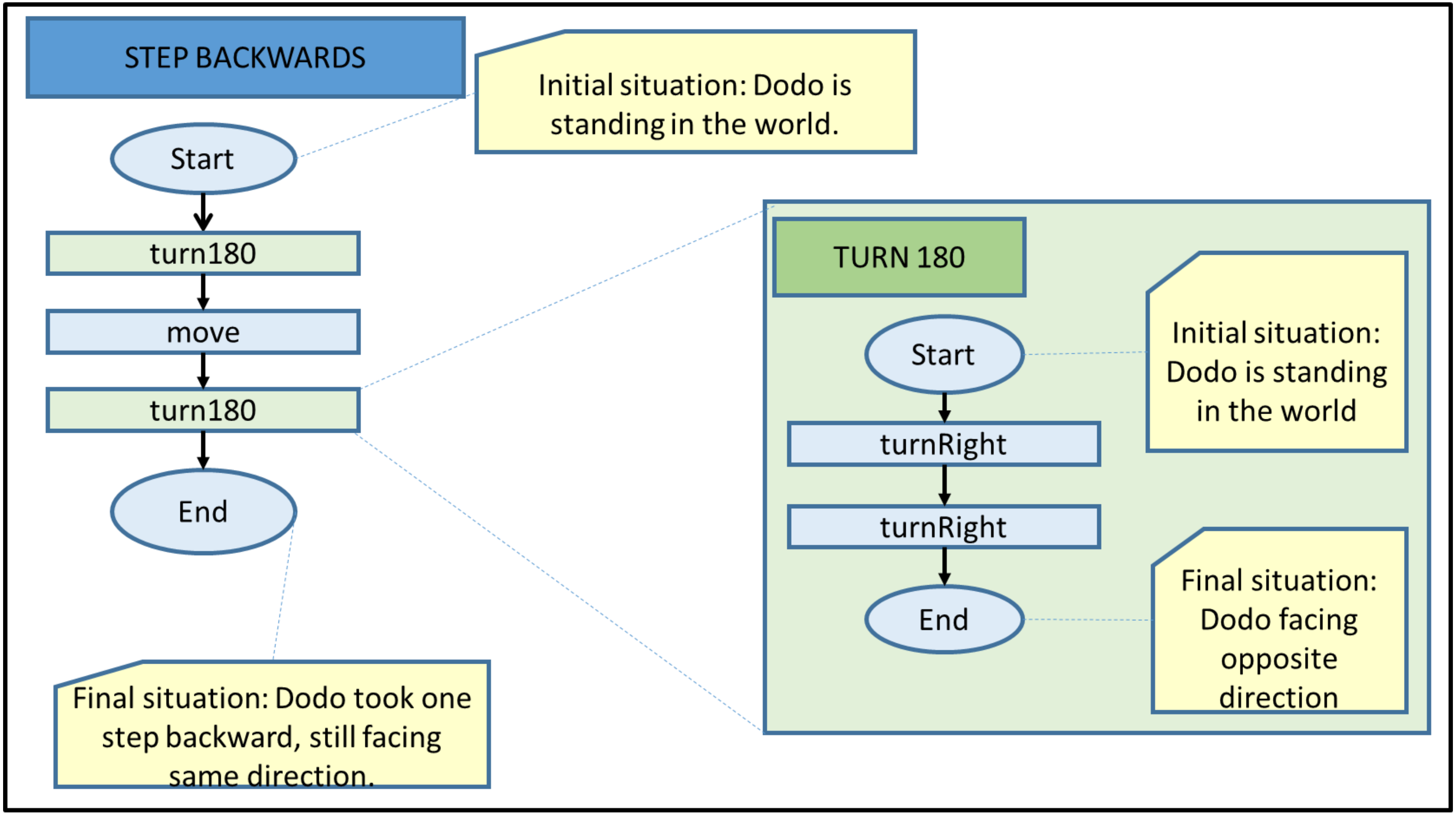
Method with repeating code



Initial situation: Dodo is standing in the world.

Final situation: Dodo took one step backward, still facing same direction.

Use submethods



Initial situation: Dodo is standing in the world.

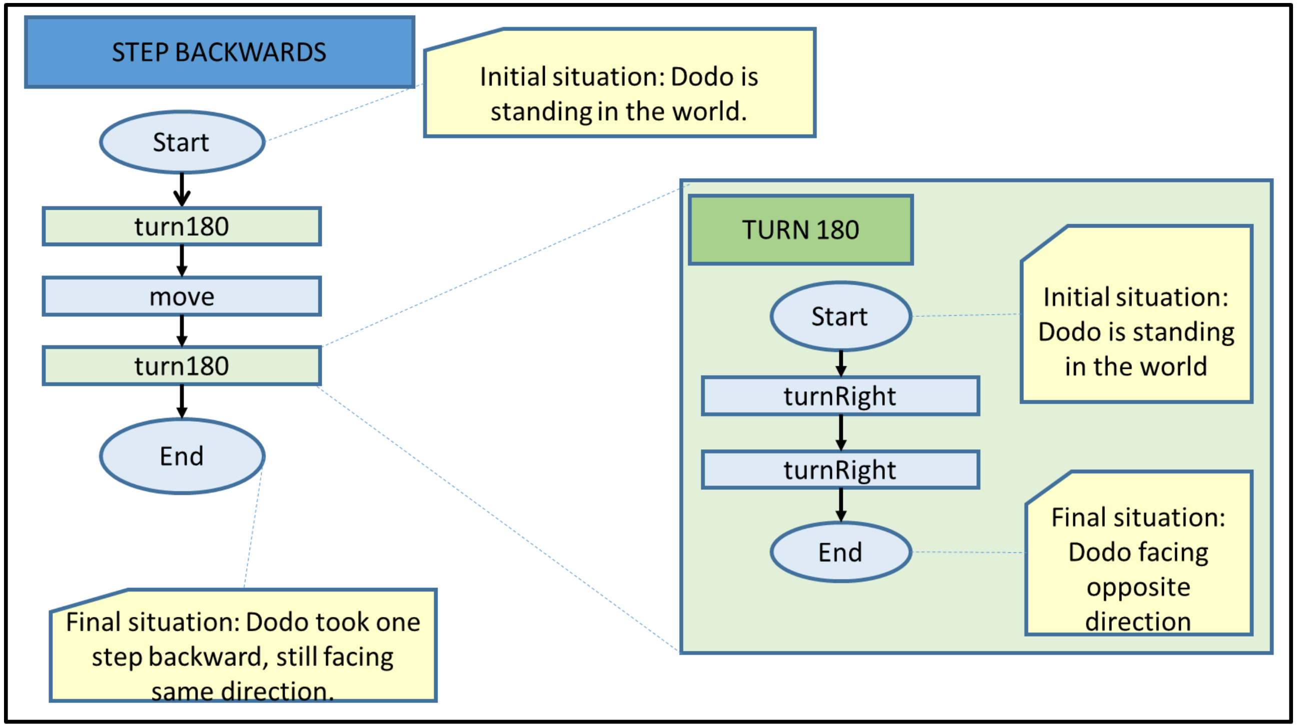
Final situation: Dodo took one step backward, still facing same direction.

Initial situation: Dodo is standing in the world

Final situation: Dodo facing opposite direction

Advantages submethods

- Easier to read / **understand**
- Code can easily be **adjusted**
- **Testing** of smaller (code) units
- Submethods can be **re-used** in other algorithms



STEP BACKWARDS

Initial situation: Dodo is standing in the world.

Start → turn180 → move → turn180 → End

Final situation: Dodo took one step backward, still facing same direction.

TURN 180

Start → turnRight → turnRight → End

Initial situation: Dodo is standing in the world

Final situation: Dodo facing opposite direction

Advice when 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!

Computational thinking

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

Wrapping up [1]

Save your work!

Discuss how/when to finish off and who will turn it in.

Homework:

- Course downloads can be found at: http://www.cs.ru.nl/~S.Smetsers/Greenfoot/Dominicus/
- Finish Assignment 2
- Finish Assignment 3
- **Hand via email to sjaaksm@live.com**

Wrapping up [2]

- Quiz: what to expect?
  - Topics: Assignment 1 & 2
  - Difference between accessor/mutator methods
  - Signature of a method (incl. parameters, results)
  - Types (such as int, boolean, String, void)
  - Explain flowcharts: sequence, selection, repetition
  - Devise an algorithm in words
  - Transform an algorithm into flowchart

- Reflection/evaluation: tips/tops