Introduction to Programming

02 Control Structures

Stephan Krusche

8 November 2023
Technical University of Munich

📣 Announcement: exam registration

- Exam registration is still open: CIT5230000
- Ensure you register before the deadline on **November 13th, 2023**
  - You only need to register once for all exam activities in this course
- **Act promptly and ensure you meet this critical deadline**
  - If you do not register, you cannot participate in the graded activities

📣 Announcement: intermediate exam 2 date **changed**

- One-time exception to avoid conflicts with home visits just before Christmas
- Intermediate exam 2 takes place on

  December 11th, 2023, 19:00 - 20:40

- Please note that the new date is fixed and can no longer be changed

Schedule

#	Date	Subject
1	18.10.23	Introduction
1b	25.10.23	Central exercise
	01.11.23	**No lecture**
2	**08.11.23**	**Control Structures**
3	15.11.23	Data Types
4	22.11.23	Object Orientation I
5	29.11.23	Object Orientation II
6	06.12.23	Object Orientation III
7	13.12.23	Algorithms
	20.12.23	**No lecture**
8	10.01.24	Programming Languages
9	17.01.24	Graphical User Interfaces
10	24.01.24	Recursion
11	31.01.24	Beyond Programming
12	07.02.24	Course Review

Roadmap of today’s lecture

- **Context**
  - Understand the difference between objects and classes
  - Implement attributes, constructors and methods
  - Instantiate objects using constructors and invoke methods
- **Learning goals**
  - Implement conditional statements using **`if`** and **`switch`**
  - Properly deal with **`null`** values
  - Implement iterations using **`for`**, **`while`** and **`do-while`**
  - Use **`arrays`** to store multiple elements of the same type
  - Implement basic operations such as **search** and **sort**

Outline

- **Expressions and statements**
- Conditional selection
- Iteration
- Arrays
- Search
- Sort

Motivation

- A programming language should
  - Provide data structures
  - Allow operations on data
  - Provide control structures for flow control
- We have already looked first at data structures and operations
- Control structures use statements and expressions

Expressions

- Appear in a program…
  - … on the right side of assignments `x = expression;`
  - … as arguments of methods `f(expression1, ..., expressionn);`
  - … in the body of functions `return expression;`
- Each expression has a **type**
- Examples
  - Expression of type **`double`** `(3.0 + y) * 4.0`
  - Expression of type **`String`** `"Hello" + " " + "World" + 2`
  - Expression of type **`Point`** `new Point(1.0, 2.0)`
  - Expression of type **`boolean`** `x == 2` Also called condition

Expressions

- Expressions are composed as follows
  - `Variable` or `Attribute`
  - `f(expression`$_1$`, ..., expression`$_n$`)` (if return type is **not void**)
  - **`new ...`**
  - `Expression1 ⊕ Expression2`

    Binary operator, e.g. **`a + b`** or **`c – d`**
  - `⊙ expression`

    Unary operator, e.g. **`– fahrenheit(15)`**
- Two other expressions in Java
  - **`_ instanceof _`** class membership test
  - **`_ ? _ : _`** conditional expression (if-then-else, ternary operator)

Statements

- Typically cause a change of state
- End with a semicolon (;) and do **not** have a type
- In Java, there are a variety of statements
  - Declaration of local variables

```
int x;
```

  - Assignment

```
x = y;
```

  - Method call
    (if return type is **void**)

```
f(x, 1);
```

  - Block statement

```
{ statement1; ... ; statementn; }
```

  - Return statement

```
return expression;
```

Comparison expression - statement

	Expression	Statement
Typed	Yes	No
Purpose	Calculation	Execution
Effect	Evaluates a value	Changes the state
Syntax	Without `;`	With `;`

- Statements may contain expressions
- The body of each method is always a sequence of statements
- In Java, there are also so-called expression statements
  - These are statements that can also be used as an expression at the same time
  - Example: `i++;`

Sequence of statements

Example

```
int x, y, result;
x = InputReader.readInt("Number 1: ");
y = InputReader.readInt("Number 2: ");
result = x + y;
System.out.println("Sum: " + result);
```

Note: please copy the **`InputReader`** on https://gist.github.com/krusche/f8bdf092159cc272f5e3ff513f2b1cb5

Sequence of statements

- Only one operation is performed at any time
  - Each operation is performed exactly once one after the other
  - None is repeated, none omitted
  - Order of execution as defined in the program code
- At the end of the last operation, the program execution ends

→ A sequence of statements only allows very simple programs

→ **We need more powerful control structures**

Outline

- Expressions and statements
- **Conditional selection**
- Iteration
- Arrays
- Search
- Sort

Conditional selection

Example

```
int x, y, result;
x = InputReader.readInt("Number 1: ");
y = InputReader.readInt("Number 2: ");
if (x > y) {
    result = x + y;
}
else {
    result = x - y;
}
System.out.println(result);
```

- First, the condition is evaluated
  - If it is fulfilled (= **`true`**), the operation directly after **`if`** will be performed
  - If it is not fulfilled (= **`false`**), the operation directly after **`else`** will be performed

Conditional selection

Instead of individual operations, the alternatives can also consist of statements

Example

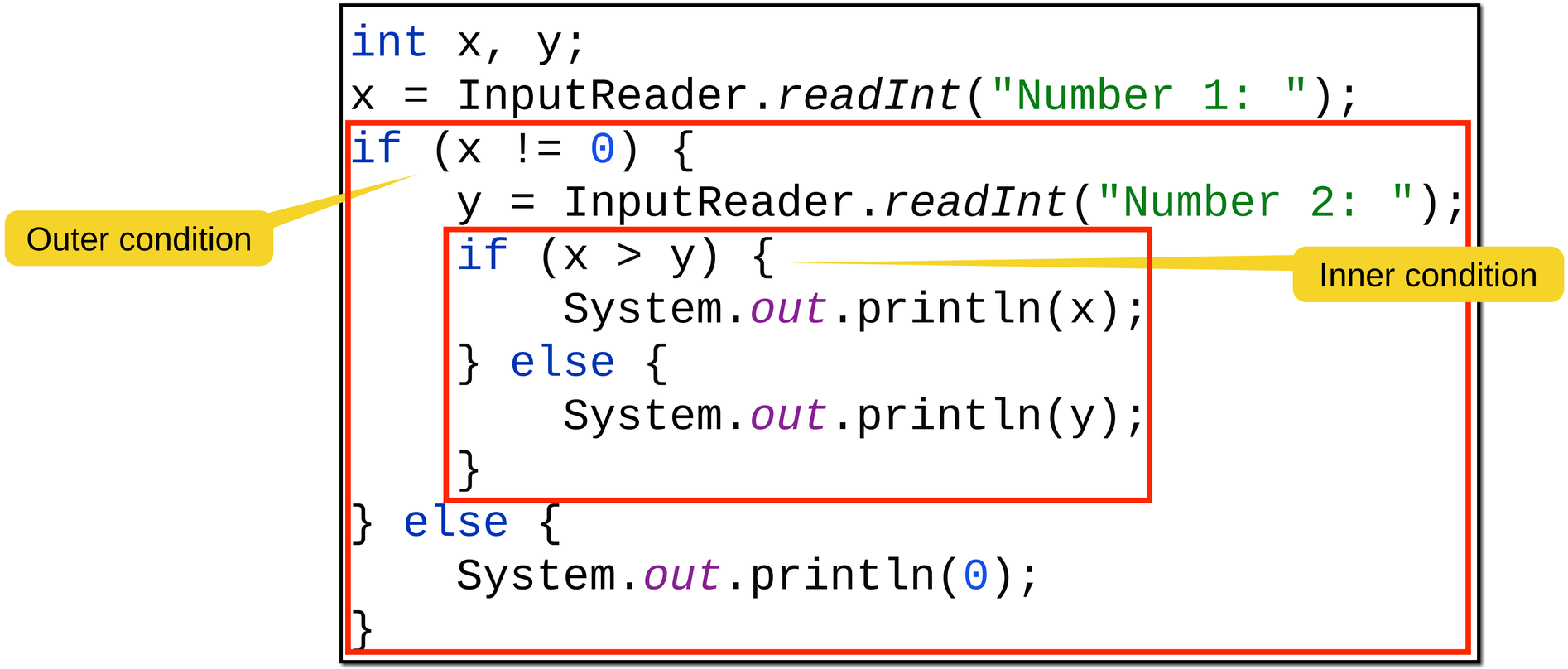
```
int x;
x = InputReader.
        readInt("Number 1: ");
if (x == 0)
    System.out.println("0");
else if (x < 0)
    System.out.println("-1");
else
    System.out.println("+1");
```

```
int x = InputReader.
        readInt("Number 1: ");
if (x == 0) {
    System.out.println("0");
} else if (x < 0) {
    System.out.println("-1");
} else {
    System.out.println("+1");
}
```

Even if there is only one statement, it is best practice to use curly braces **{** … **}**

Nested conditional selections

Example




```
int x, y;
x = InputReader.readInt("Number 1: ");
if (x != 0) {
    y = InputReader.readInt("Number 2: ");
    if (x > y) {
        System.out.println(x);
    } else {
        System.out.println(y);
    }
} else {
    System.out.println(0);
}
```

Nested conditional selections

Example

```
int x, y;
x = InputReader.readInt("Number 1: ");
if (x != 0) {
    y = InputReader.readInt("Number 2: ");
    if (x > y) {
        System.out.println(x);
    } else {
        System.out.println(y);
    }
}
```

You can also leave out the **`else`** part of an **`if`** statement when no alternative should be executed

`null`

- Represents the (potentially intentional) absence of any value or object
- Checking for **`null`** can prevent unexpected errors and crashes

```
if (object == null) {
    // Handle null case
}
```

```
if (object != null) {
    // You can safely invoke methods on this object now
}
```

Example

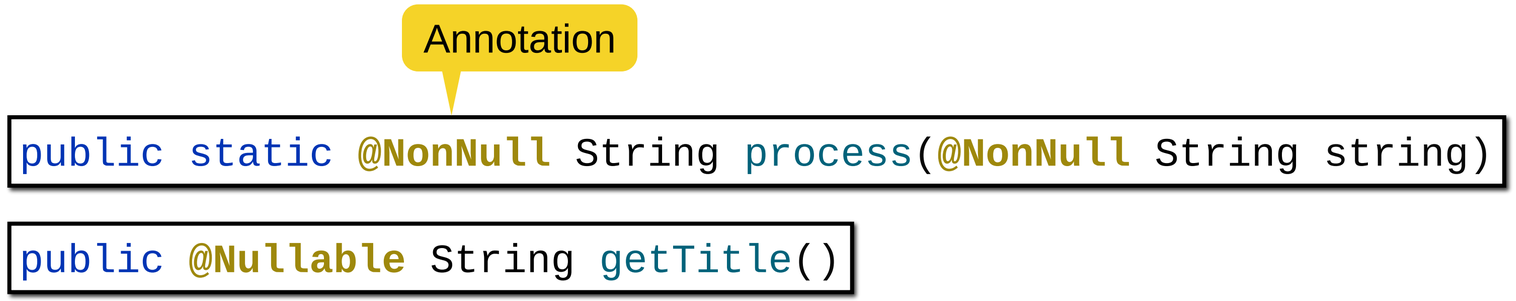
```java
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Pet {
    private String name;
    private LocalDate birthDate;

    public String getName() { return name; }
    public LocalDate getBirthDate() { return birthDate; }

    public void setName(String name) {
        this.name = name;
    }

    public long calculateAge() {
        return ChronoUnit.YEARS.between(birthDate, LocalDate.now());
    }
    public String uppercaseName() {
        return name.toUpperCase();
    }
}
```

Not **null** safe

Problem: the code is not **null** safe and can easily lead to **NullPointerExceptions**

Checker framework

- Created by the University of Washington
- Includes a **`NonNull`** module and additional functionality
- More information in the documentation on https://checkerframework.org
- Tutorial: https://github.com/glts/safer-spring-petclinic/wiki/Our-mission
- **`Nullness`** checker promise: if it issues no warnings for a given program, then running that program will never **`throw`** a **`NullPointerException`**



→ **Will be integrated into the exercises in the future**

Improved example

```java
import org.checkerframework.checker.nullness.qual.*;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Pet {
    @Nullable private String name;
    @NonNull private final LocalDate birthDate;

    public Pet(@NonNull LocalDate birthDate) {
        this.birthDate = birthDate;
   }

    @Nullable public String getName() { return name; }
    @NonNull public LocalDate getBirthDate() { return birthDate; }

    public void setName(@Nullable String name) {
        this.name = name;
    }

    public long calculateAge() {
        return ChronoUnit.YEARS.between(birthDate, LocalDate.now());
    }

    @Nullable public String uppercaseName() {
        if (name != null) {
            return name.toUpperCase();
        }
        else {
            return null;
        }
    }
}
```

Cannot become **null** anymore

Cannot become **null** anymore

It is safe to invoke this method with **null**

null safe

Switch statement

Example

```
static final char NEW = 'n';
static final char OPEN = 'o';
static final char SAVE = 's';
static final char QUIT = 'q';

void doCommand() {
    char command = InputReader.readChar("Command: ");
    switch (command) {
        case NEW  : createNewFile();
            break;
        case OPEN : openFile();
            break;
        case SAVE : saveFile();
            break;
        case QUIT : exitProgram();
            break;
        default   : System.out.println("Unknown command: " + command);
            break;
    }
}
```

Switch statement

- Realizes another form of branching

`switch` is as powerful as **`if`** statements, but allows more readable code

```
switch (expression) {
    case value_1 : statement_1;
    case value_n : statement_n;
    ...
    default      : statement;
}
```

- The **`expression`** must be of type **`char`**, **`byte`**, **`short`**, **`int`**, **`String`** or **`enumerated types`**

Covered later

- The values after the **`case`** must be constant (no variables)
- A "**`case`** `value`" only sets the entry point within the **`switch`** blocks
- The **`break`** statement causes the (immediate) **exit** of the entire **`switch`** block
- Without the **`break`**, all statements of the following **`case`** blocks are executed

Switch statement

Example

```
int daysOfMonth(int month) {
    int days = 0;
    switch (month) {
        case  1: days = 31; break;
        case  2: days = 28; break;
        case  3: days = 31; break;
        case  4: days = 30; break;
        case  5: days = 31; break;
        case  6: days = 30; break;
        case  7: days = 31; break;
        case  8: days = 31; break;
        case  9: days = 30; break;
        case 10: days = 31; break;
        case 11: days = 30; break;
        case 12: days = 31; break;
    }
    return days;
}
```

```
int daysOfMonth(int month) {
    int days = 0;
    switch (month) {
        case  2: days = 28; break;
        case  4:
        case  6:
        case  9:
        case 11: days = 30; break;
        default: days = 31; break;
    }
    return days;
}
```

Right variant shorter, but less readable

Months greater 12 and smaller 1 are also accepted

→ **More difficult to find errors**

Break

10 min

The lecture will continue at **14:55**

Outline

- Expressions and statements
- Conditional selection
- **Iteration**
- Arrays
- Search
- Sort

Iteration (repeated execution)

```
int x, y;
x = InputReader.readInt("Number 1: ");
y = InputReader.readInt("Number 2: ");
while (x != y) {
    if (x < y) {
        y = y - x;
    } else {
        x = x - y;
    }
}
System.out.println(x);
```

While loop

- Syntax

```
while (condition) {
    body;
}
```

Consists of multiple statements

- First, the condition is evaluated
  - If the condition evaluates to **`true`**, the **`body`** of the while loop is executed
  - After the **`body`** is executed, the **condition** is evaluated again
  - If the condition evaluates to **`false`**, the program continues after the **`while`** loop

Example: factorial function

```
int fac() {
    int i = InputReader.readInt("Positive number: ");
    int fac = 1;
    if(i < 0) {
        return -1;
    }
    if(i == 0) {
        return 1;
    }
    while(i > 0) {
        fac = fac * i;
        i--;
        // shorter alternative: fac *= i--;
    }
    return fac;
}
```

Error code as the factorial of a negative number is not really defined

`break` revisited

- Sometimes a program wants to exit a loop before all loop passes have been processed
- **`break;`** causes the innermost loop to be exited immediately
  - Example: calculate the sum of read-in numbers until the user input is '0'

```
void sum() {
    int i;                                  // Preparation
    int sum = 0;
    while (true) {                          // Loop
        i = InputReader.readInt("i = ");
        if (i == 0) {
            break;
        }
        sum = sum + i;
    }
    System.out.println("sum = " + sum);  // Postprocessing
}
```

Exits the while loop

→ **`break`** should only be used sparingly and selectively, so that the code remains clear and understandable

(Do-)while loop - **syntax**

```
while (condition) {
    body;
}
```

Consists of multiple statements

```
do {
    body;
} while (condition);
```

Consists of multiple statements

Do-while loop - example

Calculate the sum of read-in numbers until the user input is ‘0'

```
void sum() {
    int i;                                  // Preparation
    int sum = 0;
    do {
        i = InputReader.readInt("i = ");
        sum = sum + i;
    } while (i != 0);                       // Loop

    System.out.println("sum = " + sum);  // Postprocessing
}
```

→ **Do-while** loops are executed at least once, since the termination criterion is only checked at the end of the loop

L02E02 Reverse it

Not started yet.

Start exercise | in-class | bonus | Easy

Due date: end of today

10 min

4 pts

- **Problem**: write a program that asks the user to enter one positive natural number and outputs the number with its digits reversed
- Use a while loop
- Examples
  - 13579 → 97531
  - 8642 → 2468
- **Hint:** use this code and extend it

```
class ReverseNumber {

    public static void main(String[] args) {
        int number = InputReader.readInt("Enter a positive number: ");
        int reverse = 0;

        // Exercise: calculate the inversion of the number

        System.out.println("Reverse of " + number + " is " + reverse);
    }
}
```

Solution

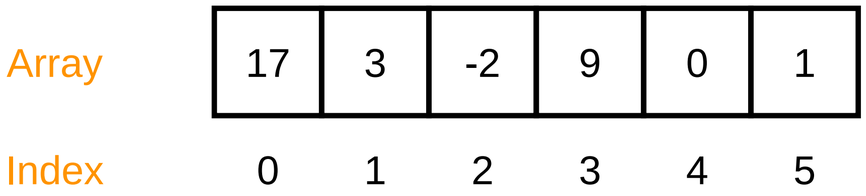
```
class ReverseNumber {

    public static void main(String[] args) {
        int number = InputReader.readInt("Enter the number: ");
        int reverse = 0;

        int temp = number;
        int remainder = 0;

        while (temp > 0) {
            remainder = temp % 10;
            reverse = reverse * 10 + remainder;
            temp /= 10;
        }

        System.out.println("Reverse of " + number + " is " + reverse);
    }
}
```

Outline

- Expressions and statements
- Conditional selection
- Iteration
- **Arrays**
- Search
- Sort

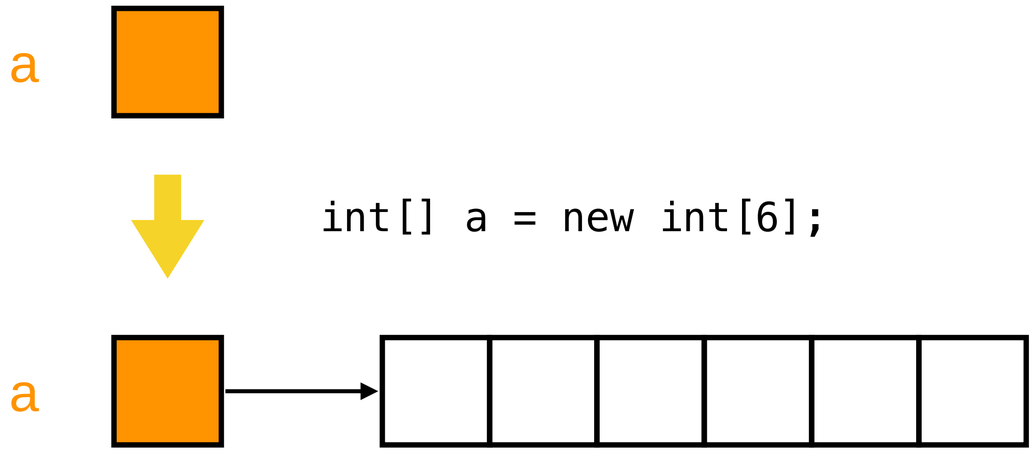
Arrays

- Often many values of the same type have to be stored
- Idea
  - Store them consecutively
  - Access individual values via their index



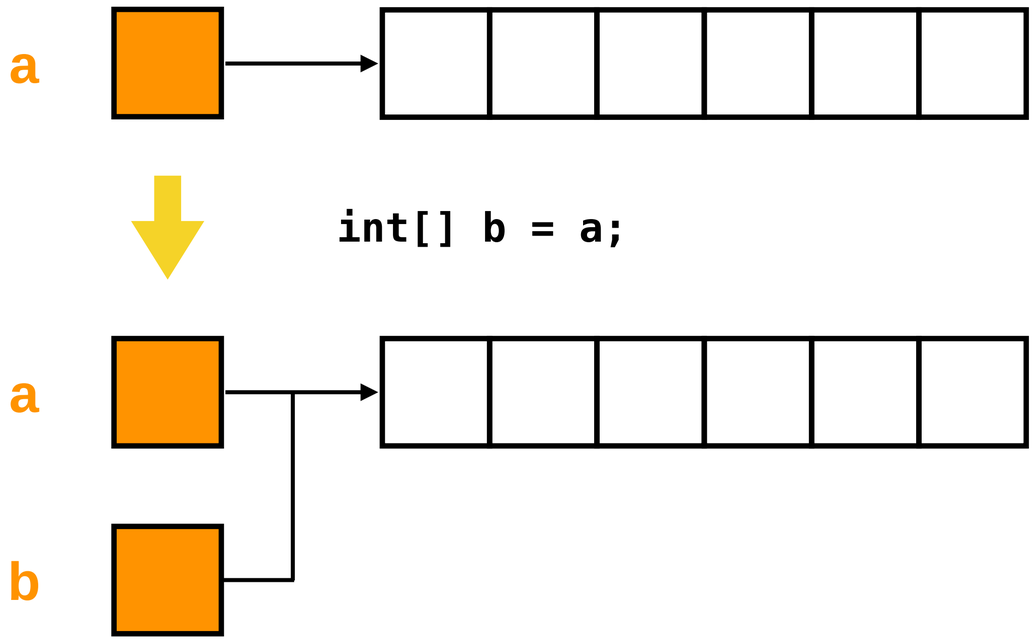
Basics (1)

- `type[] name;` declares a variable for an array whose elements are of `type`
- Alternative notation: `type name[];`
- The `new` command creates an array of a given size and returns a reference to it



Basics (2)

- The value of an array variable is therefore a reference
- **`int[] b = a;`** copies the reference of the variable **`a`** into the variable **`b`**:

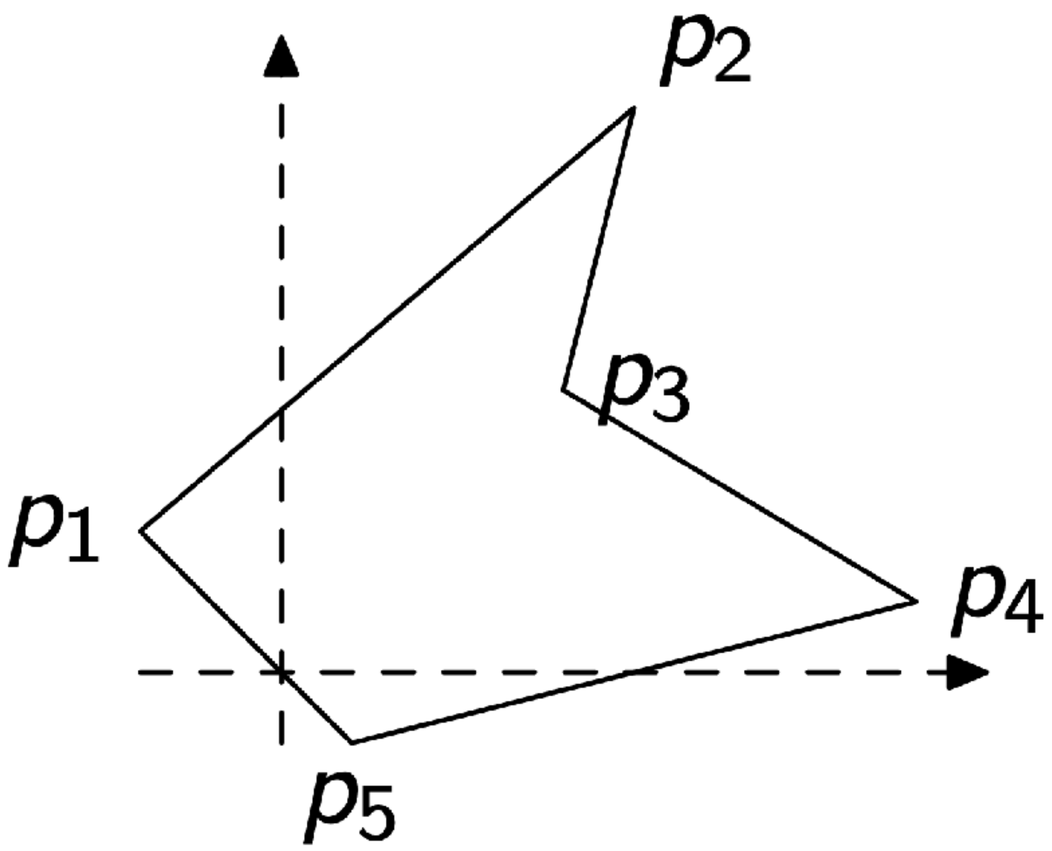


Basics (3)

- Elements of an array are numbered consecutively starting with the **index 0**
- The **i-th** element of the array `name` is accessed by `name[i]`
- The number of array elements is `name.length`
- At each access it is checked whether the index is allowed, i.e. in the interval `{0, . . . , name.length-1}`
- If the index is outside the interval, an **`ArrayIndexOutOfBoundsException`** is thrown (↑Exceptions)

Covered later

Example: polygons in R^2




```
class Polygon {
    // Attributes : array of points
    Point[] points;

    // Constructor
    Polygon(Point[] points) {
        this.points = points;
    }
    // other methods
}
```

Create polygons (1)

- **Variant 1**: explicit specification of the array length

```
Point[] points = new Point[5];

points[0] = new Point(-2.0, 2.0);
points[1] = new Point(5.0, 8.0);
points[2] = new Point(4.0, 4.0);
points[3] = new Point(9.0, 1.0);
points[4] = new Point(1.0, -1.0);
Polygon poly = new Polygon(points);
```

Create polygons (2)

- **Variant 2**: explicit specification of points

```
Point p1 = new Point(-2.0, 2.0);
Point p2 = new Point(5.0, 8.0);
Point p3 = new Point(4.0, 4.0);
Point p4 = new Point(9.0, 1.0);
Point p5 = new Point(1.0, -1.0);

Point[] points = new Point[] { p1, p2, p3, p4, p5 };
Polygon poly = new Polygon(points);
```

Example: filling an array - with **`while`**

```
int[] array;              // Declaration
int n = InputReader.readInt("Quantity: ");

array = new int[n];   // Create the array

int i = 0;
while (i < n) {
    array[i] = InputReader.readInt("Next number: ");
    i = i + 1;
}
```

Iteration pattern

- Typical form of iteration over arrays
  - Initialization of the run **index**
  - **`while`** loop with entry condition for the body
  - Modification of the run **index** at the end of the body

Example: determine the minimum with a `while` loop

```
int[] array = new int[] { 1, 4, -1, 5, 3 };  // Example int array
// Assumption: array has at least one element;
// array != null
int result = array[0];
int i = 1;                  // Initialization

while (i < array.length) {
    if (array[i] < result) {
        result = array[i];
    }
    i++;                    // Modification
}

System.out.println(result);
```

Question: why does **`i`** have the initial value **1** and not **0**?

Example: determine the minimum with a `for` loop

```
int[] array = new int[] { 1, 4, -1, 5, 3 };  // Example int array

// Assumption: a has at least one element;
// a != null
int result = array[0];

for (int i = 1; i < array.length; i++) {
    if (array[i] < result) {
        result = array[i];
    }
}

System.out.println(result);
```

Semantics of the `for` loop

```
for (initialization; condition; modification) {
    statements;
}
```

corresponds to

```
initialization;
while (condition) {
    statements;
    modification;
}
```

where `i++` is equivalent to `i = i + 1`

Recommendation: prefer `for` loops when you iterate through arrays or when you know the number of iterations beforehand

Recommendation for the usage of loops in Java

Comparison	`for` loop	`while` loop	`do-while` loop
When to use	number of iterations is fixed	number of iterations is **not** fixed	number of iterations is **not** fixed and the loop must execute at least once

Syntax	`for(init;cond;incr/decr) {`
	`  // statements`
	`}`
		`while(cond) {`
		`  // statements`
		`}`
			`do {`
			`  // statements`
			`} while(cond);`

Syntax for infinite loop	`for(;;) {`
	`  // statements`
	`}`
		`while(true) {`
		`  // statements`
		`}`
			`do {`
			`  // statements`
			`} while(true);`

`++` and `--`

- The operators **`++x`** and **`x++`** both **increment** the value of the variable **`x`** by **`1`**
- The operators **`--x`** and **`x--`** both **decrement** the value of the variable **`x`** by **`1`**
- **`++x`** and **`--x`** do this before the value of the expression is determined (**pre-increment** / **pre-decrement**)
- **`x++`** and **`x--`** do this after the value has been determined (**post-increment** / **post-decrement**)
- **`a[x++] = 7;`** corresponds to

```
a[x] = 7;
x = x + 1;
```

- **`a[++x] = 7;`** corresponds to

```
x = x + 1;
a[x] = 7;
```

Attention

- In Java, variable assignments are not only statements, but also expressions
- The assignments **`x = 5`** and **`i = i + 1`** are expressions
- The value is the value of the right side
- The modification of the variable **`i`** is done as a side effect
- The semicolon **`;`** after an expression just throws away the value

→Can lead to hard-to-find errors in conditions

```
boolean x = false;
if (x = true) { // Attention!
    System.out.println("Sorry! This must be an error ...");
}
```

Example: reading an array with the `for` loop

`public` means: can be used from other classes

`static` denotes a class method

Local variables: only visible and usable within the block `{ ... }`

```
public static int[] readArray(int number) {
    // number = Number of elements to read
    int[] result = new int[number]; // Create the array
    for (int i = 0; i < number; i++) {
        result[i] = InputReader.readInt("Next number: ");
    }
    return result;
}
```

Example: copying arrays

```java
// Assumption array != null
static float[] copy(float[] array) {
    float[] copy = new float[array.length];
    for (int i = 0; i < array.length; i++) {
        copy[i] = array[i];
    }
    return copy;
}
```

- Application: **"`float`[] copy = copy(array);"**
- **"`float`[] copy = array;"** does not copy the array!
  (Why? → see references)
- Faster variant: `Arrays.`*`copyOf`*`(array, array.length)`

Example: copying arrays

```
public static void main(String[] args) {
    float[] array = { 2.3f, 1.2f, 4.8f, 5.46f, 1.23f };

    float[] copy = copy(array);     // Copy
    float[] alias = array;          // Alias

    copy[2] = 0.0f;
    System.out.println(array[2]);

    alias[2] = 1.0f;
    System.out.println(array[2]);
}
```

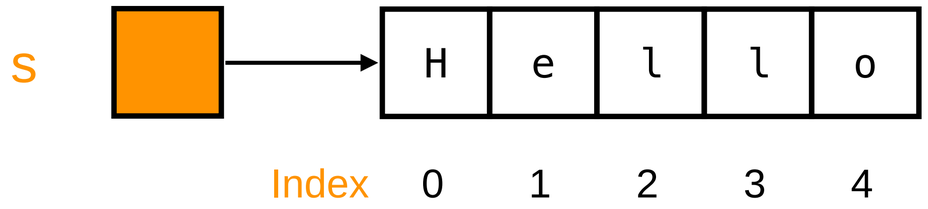
Output: 4.8

Output: 1.0

- The original array is **not** affected by the change to the copy
- The original array is affected by the change to the alias

String - char array

```
String s = "Hello"
```



Internal representation

Some convenience methods

```
System.out.println(s.length());

System.out.println(s.charAt(4));

System.out.println(s.contains("e"));

System.out.println(s.indexOf("e"));
```

Output: `5`

Output: `o`

Output: `true`

Output: `1`

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

L02E03 Character Block

Not started yet.

Start exercise | in-class | bonus | Easy

Due date: end of today

7 min

3 pts

- Problem: write a program that will print a box of **#** characters using the inputs **height** and **width** from the user
- Examples
  - Enter height: 4
  - Enter width: 3
  - Output:
    ```
    ###
    ###
    ###
    ###
    ```
- **Hint:** use this code and extend it

```
class CharacterBlock {

    public static void main(String[] args) {
        int height = InputReader.readInt("Enter height: ");
        int width = InputReader.readInt("Enter width: ");

        // Exercise: print the character block
    }
}
```

Solution

```
class CharacterBlock {

    public static void main(String[] args) {
        int height = InputReader.readInt("Enter height: ");
        int width = InputReader.readInt("Enter width: ");

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                System.out.print("#");
            }
            System.out.println();
        }
    }
}
```

Break

30 min

The lecture will continue at **16:20**

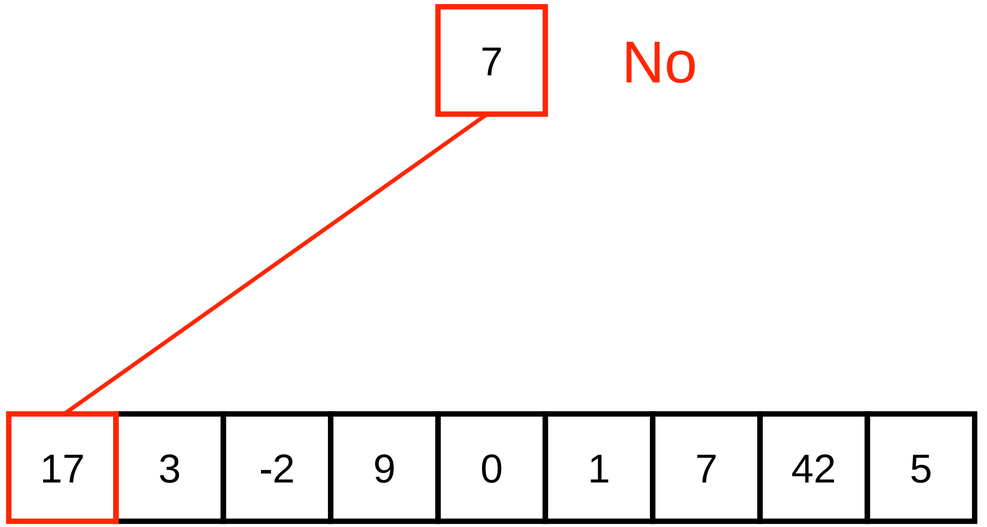
Outline

- Expressions and statements
- Conditional selection
- Iteration
- Arrays
- **Search**
- Sort

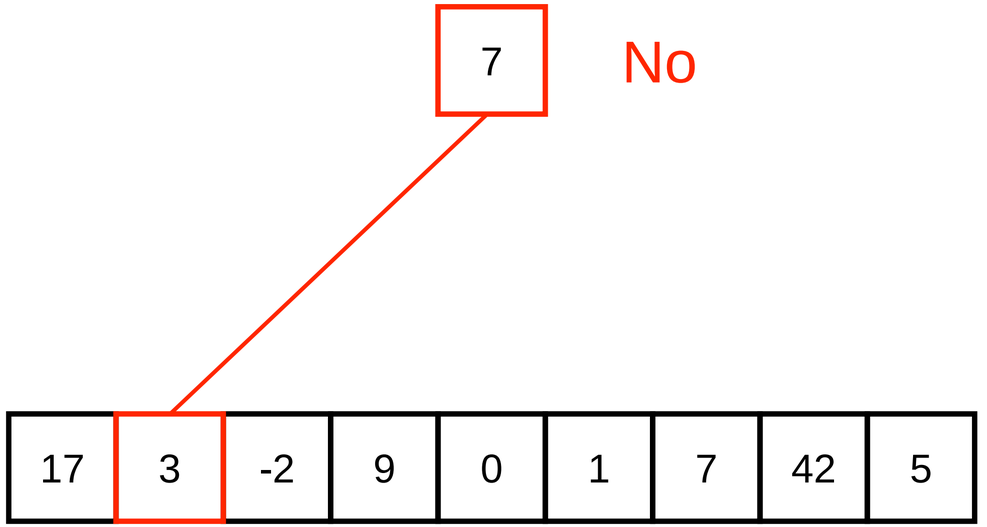
Example: search in arrays

```java
// Assumption array != null
static boolean has(long[] array, long x) {

    for (int i = 0; i < array.length; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;
}
```

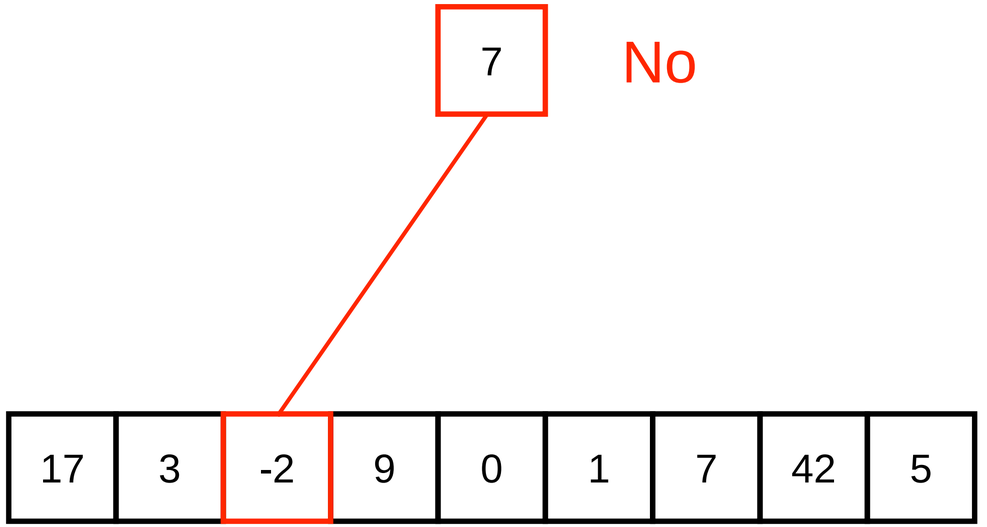
Naive search



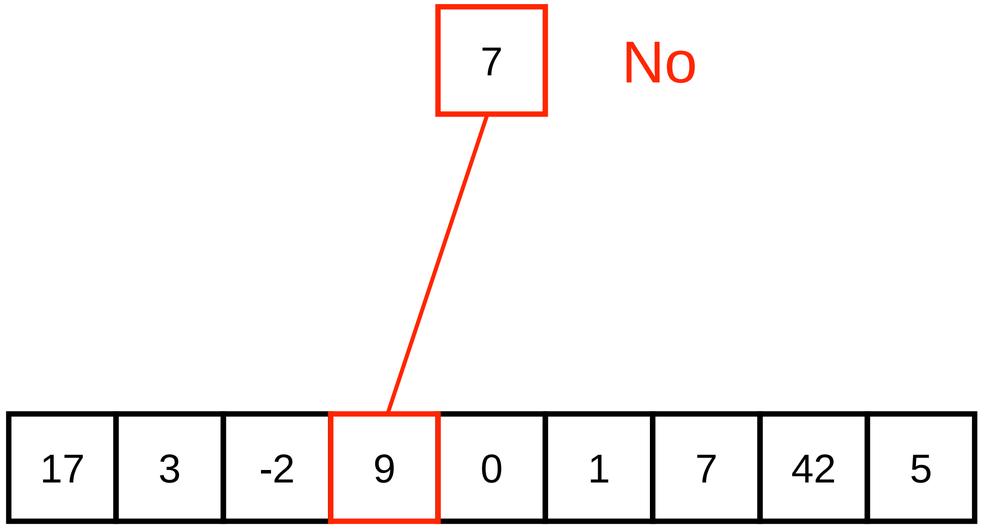
Naive search



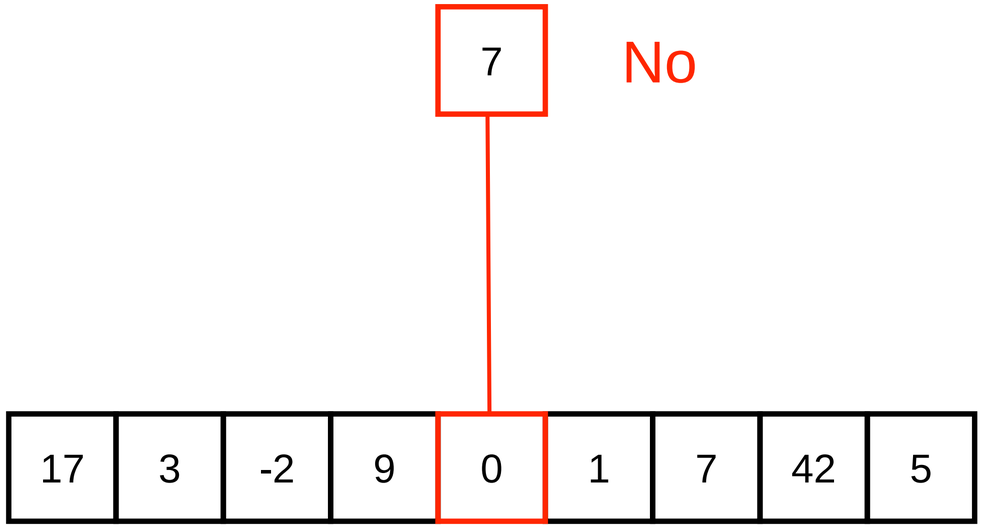
Naive search



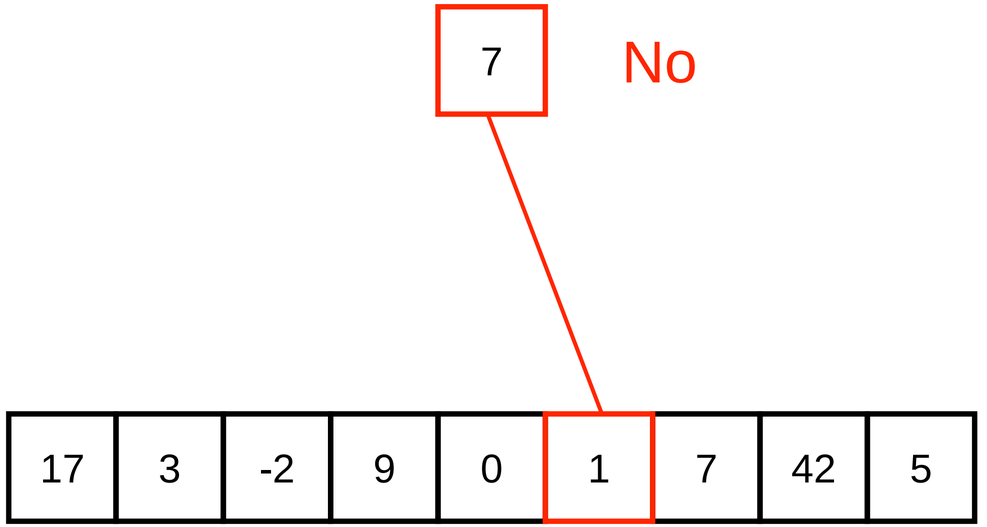
Naive search



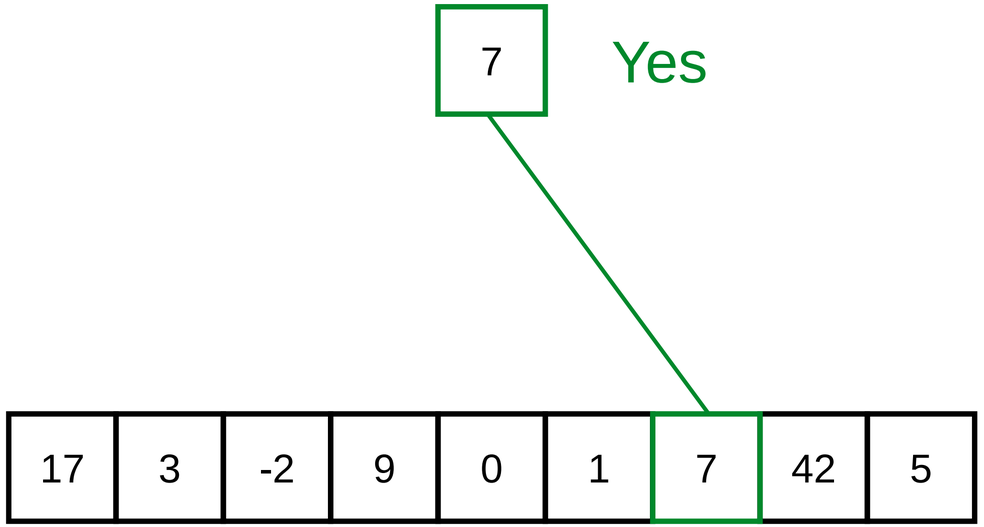
Naive search



Naive search



Naive search



Example: search in arrays

```
// Assumption array != null
static boolean has(long[] array, long x) {
    int i;

    for (i = 0; i < array.length; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;
}
```

Example: alternative search in arrays with while loop

```java
// Assumption array != null
static boolean has(long[] array, long x) {
    int i = 0;
    boolean found = false;


    while (!found && i < array.length) {
        if (array[i] == x) {
            found = true;
        }
        i++;
    }
    return found;
}
```

!found is the same as **found == false**

Exercise

7 min

- Copy the code in your Playground and invoke it with three different examples
- Also try out the example with the **`for`** loop
- Debug the code to fully understand what is going on

Break

10 min

The lecture will continue at **17:00**

Outline

- Expressions and statements
- Conditional selection
- Iteration
- Arrays
- Search
- **Sort**

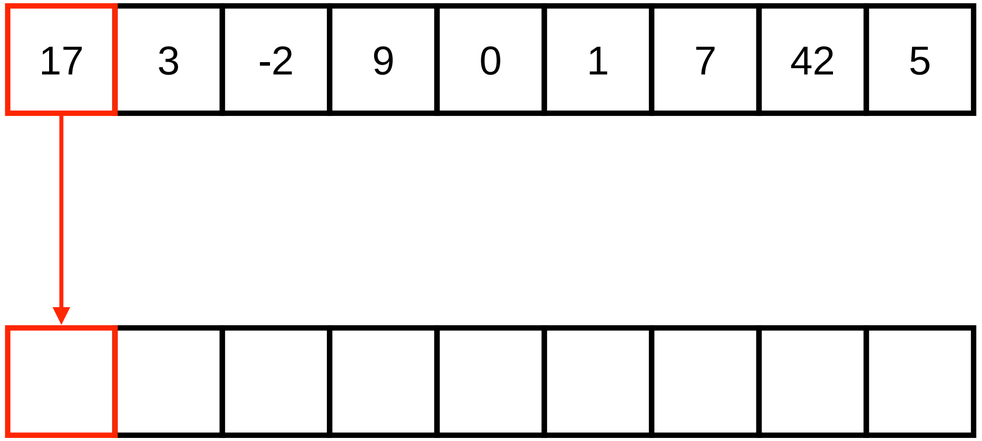
Example: sorting

- Given: a sequence of integers
- Wanted: the corresponding sequence sorted in ascending order

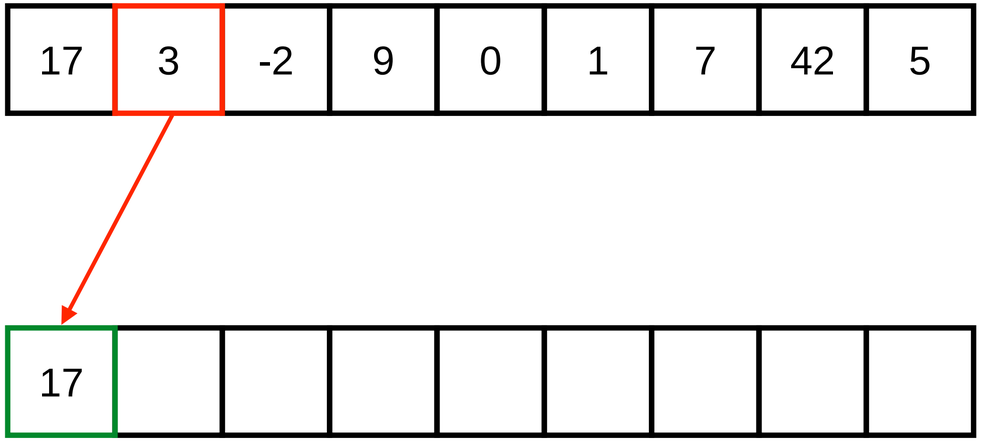
- Idea
  - Store the sequence in an array
  - Create another array
  - Insert each element of the first array in turn into the second array at the right place

→ Sorting by insertion

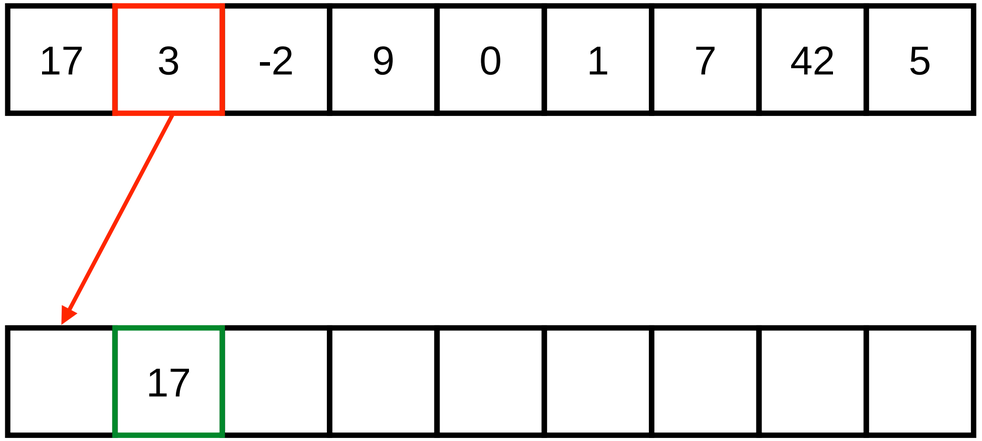
Sorting by inserting



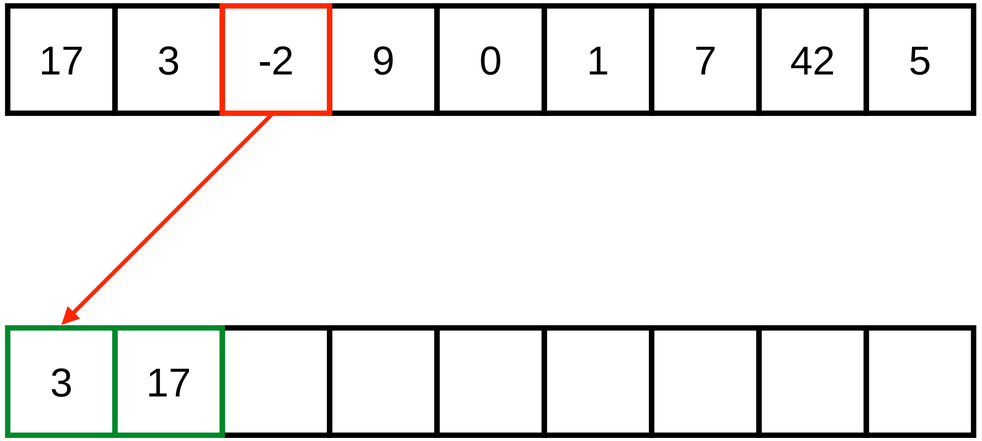
Sorting by inserting



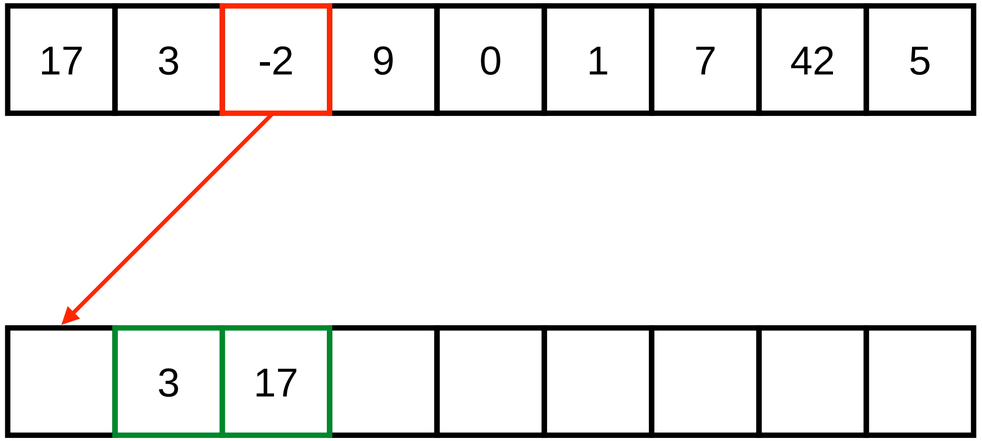
Sorting by inserting



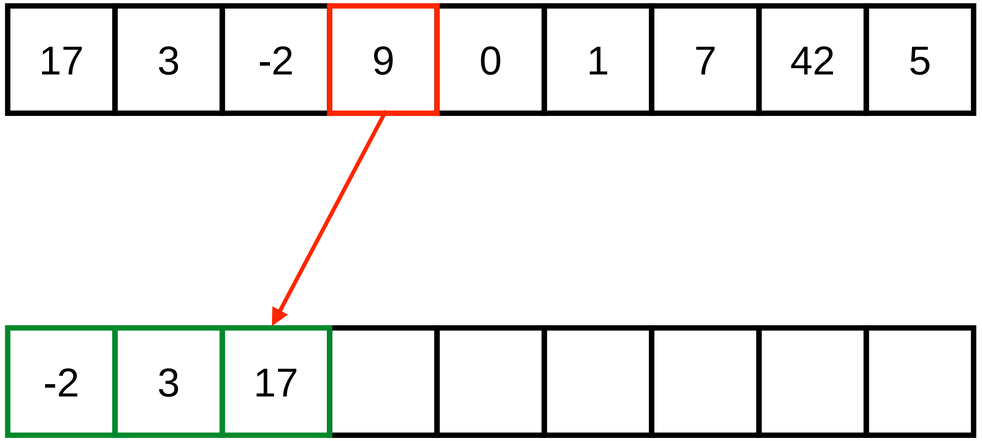
Sorting by inserting



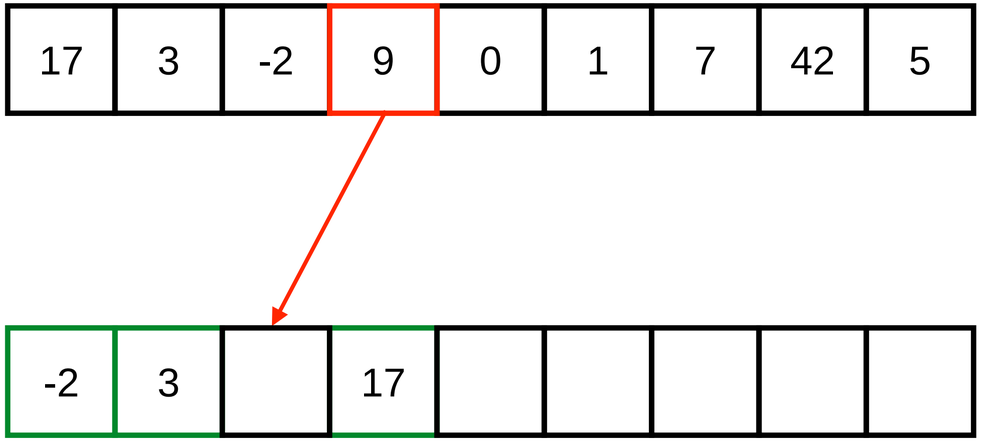
Sorting by inserting



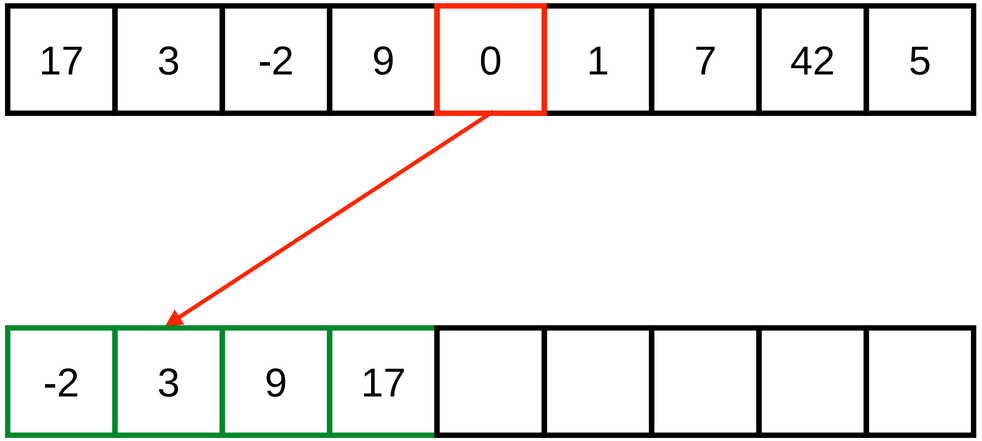
Sorting by inserting



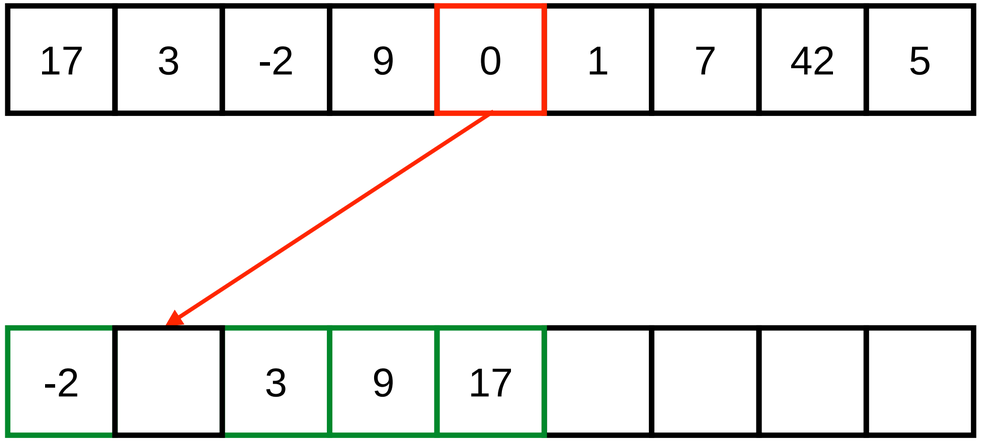
Sorting by inserting



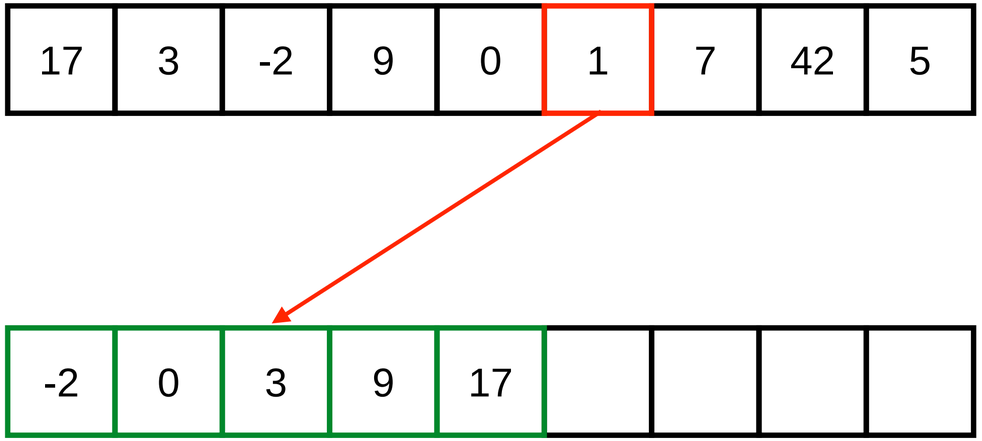
Sorting by inserting



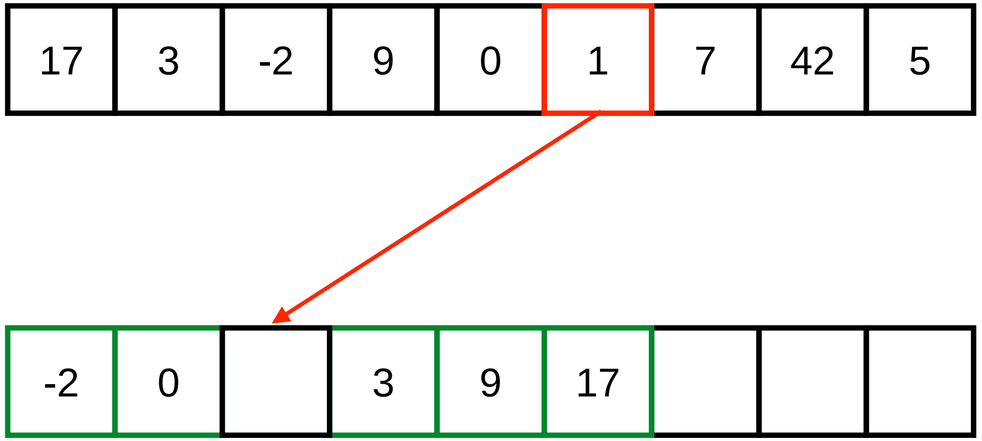
Sorting by inserting



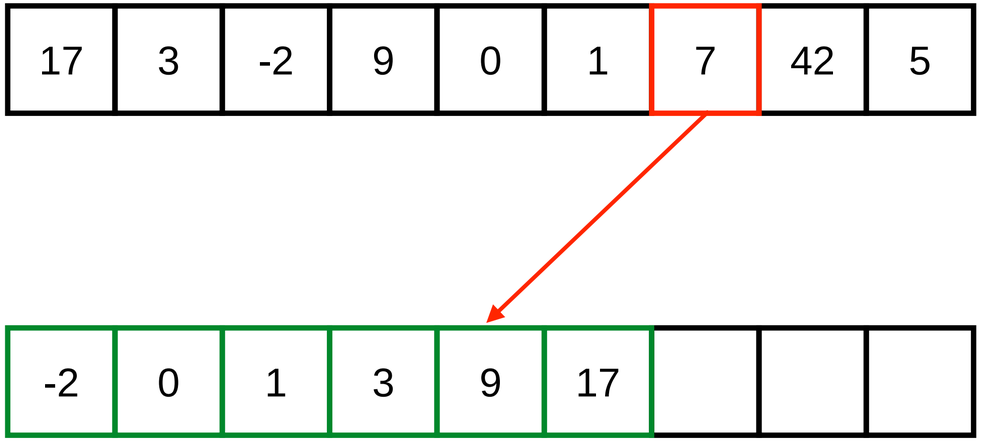
Sorting by inserting



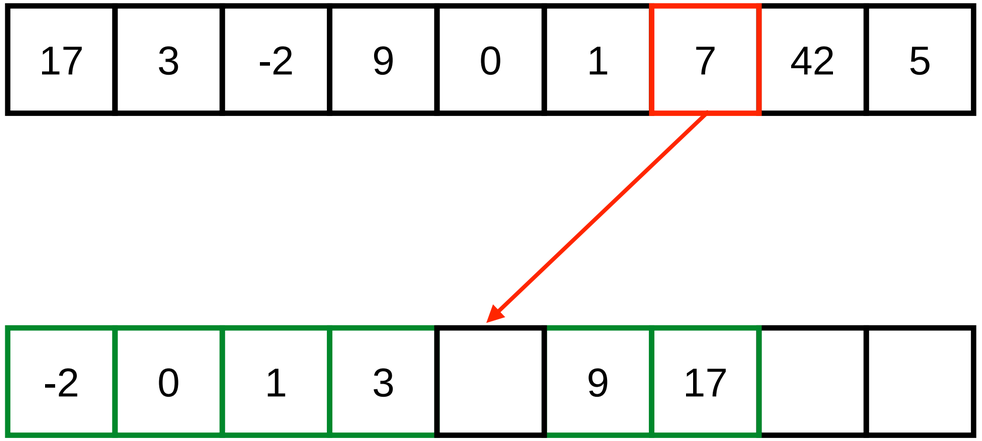
Sorting by inserting



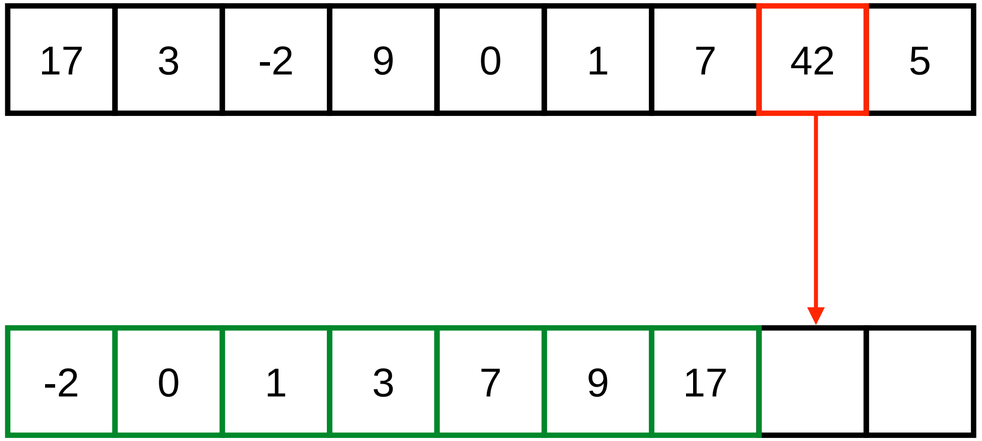
Sorting by inserting



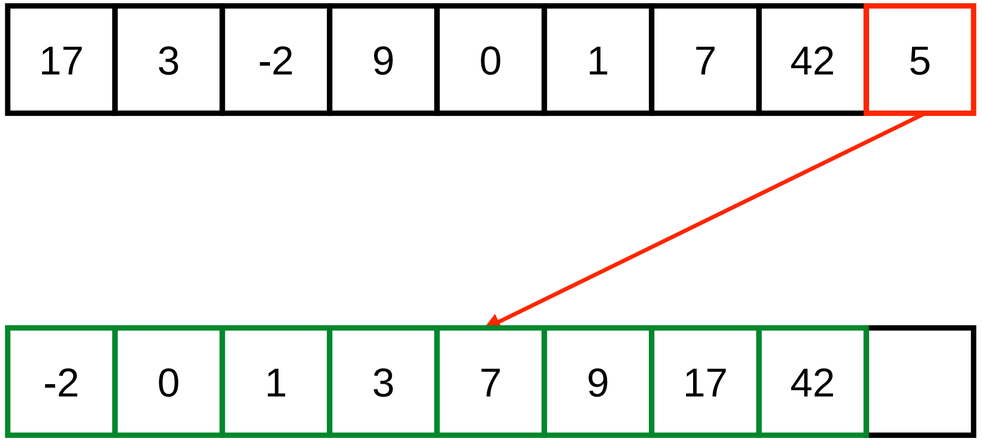
Sorting by inserting



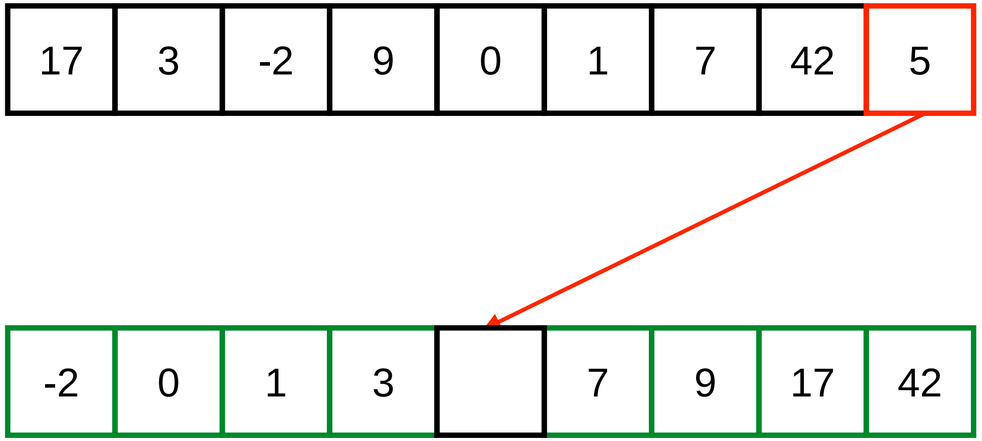
Sorting by inserting



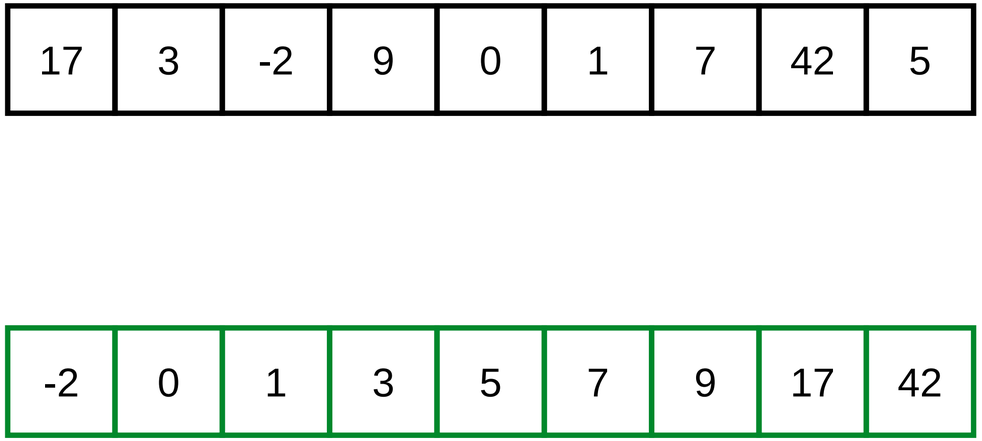
Sorting by inserting



Sorting by inserting



Sorting by inserting



Sort: code

```
// Assumption array != null
static int[] sort(int[] array) {
    int len = array.length;
    int[] result = new int[len];
    for (int i = 0; i < len; i++) {
        insert(result, array[i], i);

    }
    return result;
}
```

Result array in which the element is inserted

The element which should be inserted

Current index

Subproblem: how to **insert**?

Insert: code

This is the reference to the copy of the original array

```
static void insert(int[] array, int element, int endIndex) {

    int insertIndex = locate(array, element, endIndex);

    shift(array, insertIndex, endIndex);

    array[insertIndex] = element;
}
```

Finds the insertion point **element** in **array**

Shifts the elements **array[insertIndex], ..., array[endIndex−1]** to the **right** in **array**

New subproblems

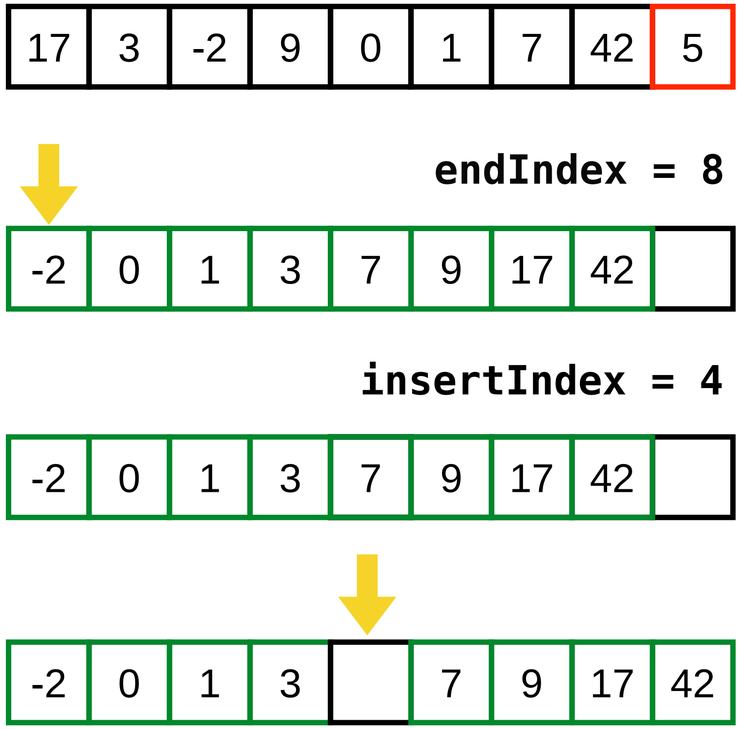
- How to find the insertion point (**locate**)?
- How to move to the right (**shift**)?

Locate and **shift**: code

```java
static int locate(int[] array, int element, int endIndex) {
    int insertIndex = 0;
    while (insertIndex < endIndex && element > array[insertIndex]) {
        insertIndex++;
    }
    return insertIndex;
}
```

```java
static void shift(int[] array, int insertIndex, int endIndex) {
    for (int i = endIndex - 1; i >= insertIndex; i--) {
        array[i + 1] = array[i];
    }
}
```

Explanation



1. The second argument of the operator **`&&`** in **`locate()`** is evaluated only, if the first is **`true`** (short evaluation) → otherwise, a non-existing array element might be accessed here (→ exception)
2. Why does the **`for`** loop in **`shift()`** run downwards from **`endIndex - 1`** to **`insertIndex`**?

Explanations

- The **`array`** is (originally) a local variable of **`sort()`**
  - Local variables are only visible in their own function body, not in the called functions
  - In order for the called helper functions to access the **`array`**, it must be passed explicitly as a parameter (call by reference)
  - Attention: the **`array`** is **not** copied → the argument is the value of the variable **`array`**, thus only a reference
  - Since the array is **not** copied, the changes performed in **`insert()`** and **`shift()`** affect the original array in the background (see section on copying arrays)
- Therefore neither **`insert()`**, nor **`shift()`** need a separate return value
- Because the problem is relatively small, an experienced programmer would not use subroutines here …

Sorting by insertion in one method

InsertionSort has a quadratic average complexity

```
// Assumption array != null
static int[] insertionSort(int[] array) {
    int[] result = new int[array.length];
    for (int endIndex = 0; endIndex < array.length; endIndex++) {
        // begin of insert
        int insertIndex = 0;

        while (insertIndex < endIndex && array[endIndex] > result[insertIndex]) {
            insertIndex ++;
        }

        for (int i = endIndex - 1; i >= insertIndex; i--) {
            result[i + 1] = result[i];
        }

        result[insertIndex] = array[endIndex];
        // end of insert
    }
    return result;
}
```

locate()

shift()

Exercise

10 min

- Copy the code for insertion sort into your playground and debug it to understand it better

Next steps

- **Tutor group** exercises
  - T02E01 - Cuff n Fluff
  - T02E02 - SQLtimate Penguin Genome
- **Homework** exercises
  - H02E01 - The Transporter
  - H02E02 - Panic at Burger House
- Read the following articles
  - https://www.w3schools.com/java/java_arrays.asp
  - https://www.w3schools.com/java/java_while_loop.asp
  - https://www.w3schools.com/java/java_for_loop.asp

→ Due until **Wednesday, November 15, 13:00**

Summary

- `if` and `switch` statements allow controlling the flow of the program execution based on conditions
- `for`, `while` and `do-while` loops allow for iteration, e.g. over arrays
- Arrays are simple data structures that allow to store multiple elements of the same type
- Arrays use **reference** semantics: they can be passed into methods and modified inside the method
- **Search** and **sort** are basic operations on arrays that are often required
  - Performance is an important topic when dealing with large data structures
  - There are built-in methods in Java for sorting and searching

References

- https://www.javatpoint.com/control-flow-in-java
- https://www.w3schools.com/java/java_while_loop.asp
- https://www.w3schools.com/java/java_for_loop.asp
- https://www.w3schools.com/java/java_arrays.asp
- https://www.geeksforgeeks.org/java-while-loop-with-examples
- https://www.javatpoint.com/java-for-loop
- https://www.geeksforgeeks.org/bubble-sort
- https://www.geeksforgeeks.org/insertion-sort + https://www.youtube.com/watch?v=OGzPmgsI-pQ
- https://www.baeldung.com/java-control-structures