Educational Objectives

- You understand the advantage of defensive programming - “fail fast”
- You know how to use assertions.
- You understand the concept of arrays and you can create and use them.
- You develop an understanding of reference semantics.
- You know the basics of Strings and multi-dimensional arrays.

9. Defensive Programming

Programming with Assertions

Sources of Errors

- Errors that the compiler can find: syntactical and some semantical errors
- Errors that the compiler cannot find: runtime errors (always semantical)

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior
   
   “It’s not a bug, it’s a feature!”

2. Check at many places in the code if the program is still on track

3. Question the (seemingly) obvious, there could be a typo in the code
**Against Runtime Errors: Assertions**

```java
assert expr : msg;
```

- halts the program if the boolean expression `expr` is false
- print the message `msg` in case the assertion doesn’t hold (optional)
- gets activated by the flag `-ea` upon startup of the program
- In Code Expert the flag is activated in the playground and in future exercises.

---

**Assertions for the `gcd(x, y)`**

Check if the program is on track ...

```java
// Input x and y
Out.print("x =? ");
x = In.readInt();
Out.print("y =? ");
y = In.readInt();
```

... // Compute gcd(x,y), store result in variable a

---

**Fail-Fast with Assertions**

- Real software: many Java files, complex control flow
- Errors surface late(r) → impedes error localisation
- Assertions: Detect errors early
10. Java Arrays and Strings

Allocation, references, element access, multidimensional arrays, strings, string comparison

Arrays

Declare array variables: `int[] z;

Create an array: `z = new int[5];`

`z` is a reference to the array data, but only after the assignment to the created data otherwise it points to nowhere: `null`.

Elements are indexed. The first index is `0` and the last index is `array size - 1`

Element access: `name[index]`

Arrays are dynamic objects

Arrays are always created dynamically

`int[] b;
b = new int[10];` // 10 elements with index `0...9`

...`b = new int[20];` // can be reassigned

Size of an array can be set at runtime
But an array doesn’t grow automatically!
Arrays are not primitive

Arrays carry *metadata*:

```java
int sq = new int[7];
for (int i = 0; i < sq.length; ++i) {
    sq[i] = i * i;
}
```

```java
sq[8] = 64; // java.lang.ArrayIndexOutOfBoundsException!
```

... even over method boundaries (next time):

```java
static void print(int[] a){
    for (int i = 0; i < a.length; ++i){
        Out.println("a[" + i + "]=" + a[i]);
    }
}
```

Array assignments

```java
int[] z = new int[5];
for (int i = 0; i < z.length; ++i) {
    z[i] = i * i;
}
```

```java
int[] x = z;
int j = x[2];
x[1] = 99;
```

When assigning arrays, *references are being copied*, not data!

Example

Given an (unsorted) array `x` containing numbers from the range `[0, ..., 9]`.

Task: Write an efficient program that output for each number, how many times it occurs in `x`.

Idea
public class CountNumbers {
    public static void main(String[] args) {
        int[] numbers = {5, 4, 2, 4, 5, 7, 8, 9, 1, 1, 3, 2, 4, 5};
        int[] index = new int[10];
        for (int i = 0; i < numbers.length; i++) {
            index[numbers[i]]++;
        }
        for (int i = 0; i < index.length; i++) {
            System.out.println("Count (" + i + ")=" + index[i]);
        }
    }
}

Multidimensional arrays

double[][] matrix = new double[4][4];

// Identity matrix
for (int r = 0; r < matrix.length; ++r){
    for (int c = 0; c < matrix[r].length; ++c){
        if (r==c)
            matrix[r][c] = 1;
        else {
            matrix[r][c] = 0;
        }
    }
}

A two-dimensional array is an array of references to a one-dimensional array. Thus, the following is possible:

double[][] matrix = ...;

for (int r = 0; r < matrix.length; ++r){
    double[] vector = matrix[r];
    for (int c = 0; c < vector.length; ++c){
        System.out.print(vector[c] + " ");
    }
    System.out.println();
}
Multidimensional arrays

Even this is possible:

double[][] m = new double[5][];
for (int r = 0; r < m.length; ++r) {
m[r] = new double[r+1];
}

Array comparisons

Attention (again): Arrays are references!
double[] x = {1,2,3};
double[] y = x;
double[] z = {1,2,3};

if (y == x) {...} // y==x is true
if (z == x) {...} // z==x is false!

For the experts:
if (z.equals(x)) {...} // z.equals(x) is also false !
if (Arrays.equals(x,z)) {...} // Arrays.equals(x,z) is true.

Attention when using Arrays.equals on multidimensional arrays! (What is compared?)

Strings

String: an object that stores a character array.

String name = "Informatics";
String university = "ETH";
String lecture = name + " at " + university;
int x = 3;
int y = 5;
String coordinates = "(" + x + "," + y + ")"; // "(3,5)"

Mind the evaluation order:

int x = 3;
int y = 5;
String s1 = x+y+"X"; // s1 = "8X"
String s2 = "X"+x+y+""; // s2 = "X35"
**Characters**

Elements of a string can be accessed by index (but not replaced)

```java
String info = "Informatics";
char c = info.charAt(3); // c = 'o'
```

Strings are references as well!

```
info
I
0
n
1
f
2
o
3
r
4
m
5
a
6
t
7
i
8
c
9
s
10
```

**String comparisons**

The comparison with `==` compares references, not content!

```java
String n1 = In.readWord();
String n2 = In.readWord();
if (n1 == n2){
    Out.println(n1 + "==" + n2);
} else {
    Out.println(n1 + "!=" + n2);
}
```

Input: Info Info
Output: Info != Info

---

The comparison with `equals` compares the content!

```java
String n1 = In.readWord();
String n2 = In.readWord();
if (n1.equals(n2)){
    Out.println(n1 + " equals " + n2);
} else {
    Out.println(n1 + " not equals " + n2);
}
```

Input: Info Info
Output: Info equals Info

---

This doesn’t apply to arrays.