Recap of Part 1 & Introduction to Part 2

Lecture 9 of TDA 540
Object-Oriented Programming

Jesper Cockx

Fall 2018

Chalmers University of Technology — Gothenburg University

Welcome to Part 2 of the course!

Main topics in Part 2:

- review of Part 1 (today)
- object-oriented features of Java
  - classes, attributes, and methods
  - inheritance and polymorphism
  - abstraction and interfaces
- event-driven programming
- some useful standard libraries

Lab sessions

4 more lab sessions:

- Lab 5 (deadline 15 November):
  Translation and dice rolling
- Lab 6 (deadline 22 November):
  LCR dice game
- Lab 7 (deadline 6 December):
  Graphical interface for LCR
- Lab 8 (deadline 20 December):
  Tower defence game

Session 5 is online now, 6-8 will follow soon.

Assertions

`assert` *expr* ;

- *expr* is `true` ⇒ `assert` does nothing
- *expr* is `false` ⇒ program raises `AssertionError`

You can use assertions to:

- ensure a method is never called with invalid inputs
- test that the program produces the correct result

! To use assertions, add `-ea` to ‘VM options’ in IntelliJ

Some programming advice

- Don't write clever code
- Don't implement 'quick fixes'
- Don't repeat yourself

Some programming advice

- Don’t write clever code
  ...but make it easy to read and to change
- Don’t implement ‘quick fixes’
- Don’t repeat yourself

Some programming advice

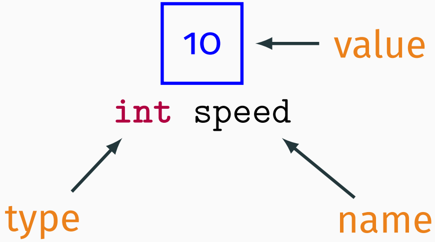
- Don’t write clever code
  ...but make it easy to read and to change
- Don’t implement ‘quick fixes’
  ...but try to understand what went wrong
- Don’t repeat yourself

Some programming advice

- Don’t write clever code
  …but make it easy to read and to change
- Don’t implement ‘quick fixes’
  …but try to understand what went wrong
- Don’t repeat yourself
  …but take the opportunity to introduce more abstraction

Variables

Variables



Identifiers

An identifier is a name of a Java entity
(a variable, a method, a class, ...)

Java identifiers rules:

- identifiers must consist of numbers, letters, underscores _, and dollar signs $
- the first character cannot be a number

Allowed: `myV4r14b13`, `_$1$`, `VERY_LONG_NAME`
Not allowed: `123abc`, `f^o*o`

A variable's life

OPERATION	CODE EXAMPLE	
declaration	`int speed;`	reserve room in memory for a variable with name `speed` and type `int`
initialization	`int speed = 10;`	set to `10` the initial value of `speed`
read/access	`if (speed > 5)`	use the current value of `speed` in an expression
write/modify	`speed = 8;`	change to `8` the value of `speed`

Blocks and scope

A block `{...}` groups together statements:

```
{ // outer block begins
  int x = 0, y = 1;
  { // inner block begins
    int z = 2;
    y = z + 1; // OK: y declared in outer block
  } // inner block ends
  y = z + 3; // Error: z not available here
} // outer block ends
```

Variables can only be used inside the block where they were declared (= the scope).

final variables

A variable declared as `final` can never change value.

```
final int ONE_DOZEN = 12;
int eggs = input.nextInt();
eggDozens = eggs / ONE_DOZEN;

ONE_DOZEN = 13;      // compile error
```

Convention: names are `ALL_CAPS`

It's a good idea to use `final` whenever possible.

Variable types

The type of a variable determines:

1. The values that a variable can take
2. The operations available on the variable

Example: a variable `speed` of type `int` can:

... take any value between -2^{31} and $2^{31}-1$

... be used in arithmetic operations (`+`, `-`, `*`, `/`, `%`, ...), assignments, and comparisons with variables of other compatible numeric types.

Primitive types in Java

Java has 8 built-in primitive types:

- 4 integer types of different size: `byte` (8 bits), `short` (16 bits), `int` (32 bits) and `long` (64 bits)
- 2 floating point types of different size: `float` (32 bits) and `double` (64 bits)
- 1 character type: `char` (8 bits)
- 1 Boolean type: `boolean` (1 bit)

Literals

```java
// Java literals and their types
1       // an int
1L      // a long (64 bit integer)
1.0     // a double
1.0f    // a float
"1"     // a string
'1'     // a char
true    // a boolean (either true or false)
```

Type conversions

Implicit, no precision loss:

- `byte` ⟶ `short` ⟶ `int` ⟶ `long`
- `float` ⟶ `double`
- `char` ⟶ `int` ⟶ `double`

Implicit, possible precision loss:

- `int` ⟶ `float`
- `long` ⟶ `float`
- `long` ⟶ `double`

Explicit with a cast, possible precision loss:

- `int` ⟶ `short`
- `double` ⟶ `int`
- ...

Example with no loss of precision

```
long companyValue = 651_500_000_000L;
  // USD 651.5 billion
int companyTaxes = 7_682_000_000;
  // USD 7.682 billion

// companyTaxes implicitly converted
// to long, no loss of precision:
long valueAfterTaxes =
  companyValue - companyTaxes;
```

Example with loss of precision

```
double width = 10.8;
int height = 11;

// casting double to int,
// with precision loss:
// 10.8 gets truncated to 10
int area = height * (int) width;
// area is 110
```

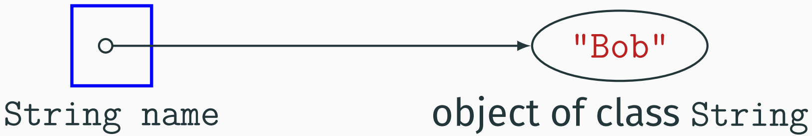
Reference types in Java

All other Java types are Reference types:

- The `String` type
- Array types such as `int[]`, `double[][]`, ...
- 8 wrapper types (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`)
- Each class in the Java standard libraries or your own code defines a new type

Variables of reference types

A variable of a reference type contains a reference to an object of that type:



Expressions

Expressions

An expression consists of variables, method calls, and operators:

- Arithmetic expressions – numeric types:

```
speed + 3
2 * time
velocity / time
time % 60
```

- Comparison expressions – Boolean type:

```
initialSpeed < finalSpeed
3 == time     // equality
answer != 42 // non-equality
```

Expressions (cont.)

An expression consists of variables, method calls, and operators:

- Boolean expressions – Boolean type:

```
true && false        // and (conjunction)
found || outOfBound // or (disjunction)
!(speed < 0)         // not (negation/complement)
```

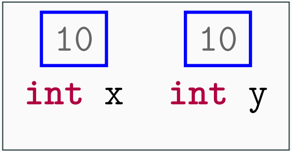
- Method calls – any type:

```
Math.sqrt(20.0)
oneString.equals(anotherString)
```

Equality comparison: primitive types

For primitive types, == denotes value equality:

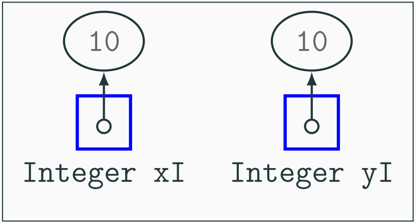
```
int x, y;
x = 10;
y = 10;
x == y
// evaluates to true
```



Equality comparison: reference types

For reference types, == denotes reference equality:

```
Integer xI, yI;
xI = new Integer(10);
yI = new Integer(10);
xI == yI
// evaluates to false
```



⇒ Always use `equals` to test equality for reference types:

```
xI.equals(yI)  // evaluates to true
```

Side effects

Some expressions have side effects: They change the value of one or more variables.

- `x++` and `x--` increase/decrease `x` by `1` and evaluate to the old value of `x`
- `++x` and `--x` increase/decrease `x` by `1` and evaluate to the new value of `x`
- Many methods also have side effects e.g. `array.sort()`, `robot.move()`

Strings

The `String` class

```
String greeting = "Hello, TDA540!";

String longString =
  "This is a veeeeeeeeeeeeeeeeeery"
  + "long string that doesn't fit"
  + "on one line.";

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

Some `String` methods

`int length()`	length of a string
`char charAt(int index)`	character at the specified index
`char[] toCharArray()`	convert string to array of `char`s
`String replace(char oldChar, char newChar)`	create new string with `oldChar` replaced by `newChar`
`String toUpperCase()`	create new string with all characters converted to UPPER CASE
`String toLowerCase()`	create new string with all characters converted to lower case
`String substring(int beginIndex, int endIndex)`	take substring from `beginIndex` to `endIndex-1`
`String[] split(String sep)`	split string into parts separated by `sep`

Example:

```
String myString = "This is a string";
String[] words = myString.split(" ");
// words == ["This", "is", "a", "string"]
```

Pretty printing

```
static String format(String format,
    Object... args)
```

Any parts starting with a `%` sign are replaced by the corresponding argument:

- `"%s"` : a string
- `"%d"` : an integer number
- `"%f"` : a floating-point number in decimal notation
- `"%e"` : a floating-point number in scientific notation

Pretty printing

```java
static String format(String format,
    Object... args)
```

More options after `%`:

- `"%.5f"` : a floating-point number with 5 digits of precision
- `"%20s"` : a string with up to 20 extra spaces in front
- `"%-5s"` : a string with up to 5 extra spaces after

Arrays

Arrays

An array = a sequence of elements of the same type.

```
int[] a;                 // declare new array
a = new int[5];          // initialize array
                         // with 5 elements
a[0] = 100;              // store 100 in position 0
a[a.length - 1] = 8;     // store 8 in last position

int[] b = new int[4];    // declare + initialize array
a[0] = b[0];             // a[0] == b[0] == 0

int[] c = {1,1,1,1,2};   // declare + initialize array
                         // with given values
```

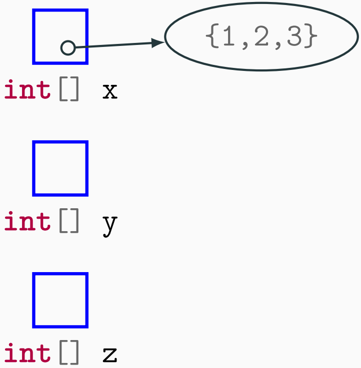
Two-dimensional arrays

A two-dimensional array = an array of arrays:

```
int[][] data = {
                 { 16,  3,  2, 13 },
                 {  5, 10, 11,  8 },
                 {  9,  6,  7, 12 },
                 {  4, 15, 14,  1 },
               };

for (int row = 0; row < data.length; row++) {
  for (int col = 0; col < data[0].length; col++) {
    // do something with data[row][col]
  }
}
```

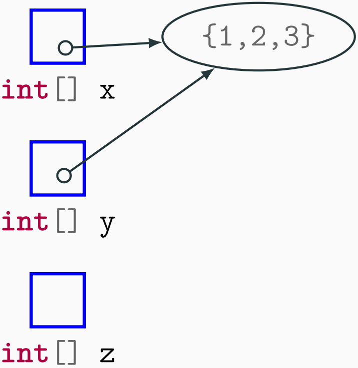
Arrays are reference types:

```
int[] x = { 1, 2, 3 };
int[] y = x;
x[1] = 5;
y[1];        // == 5

z = { 1, 5, 3 };
x == z;      // == false
Arrays.equals(x,z);
             // == true
```



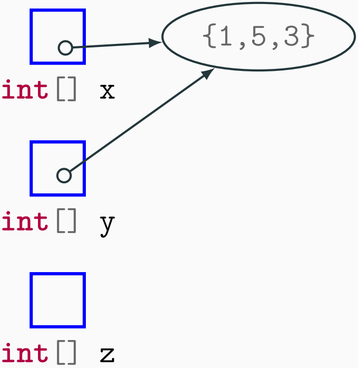
Arrays are reference types:

```
int[] x = { 1, 2, 3 };
int[] y = x;
x[1] = 5;
y[1];          // == 5

z = { 1, 5, 3 };
x == z;        // == false
Arrays.equals(x,z);
               // == true
```



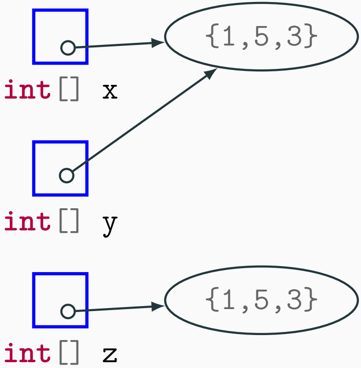
Arrays are reference types:

```
int[] x = { 1, 2, 3 };
int[] y = x;
x[1] = 5;
y[1];          // == 5

z = { 1, 5, 3 };
x == z;        // == false
Arrays.equals(x,z);
               // == true
```



Arrays are reference types:

```
int[] x = { 1, 2, 3 };
int[] y = x;
x[1] = 5;
y[1];          // == 5

z = { 1, 5, 3 };
x == z;        // == false
Arrays.equals(x,z);
               // == true
```



Statements

Statements

A statement is a single instruction to the computer:

Variable declaration `int x;`

Assignment `x = 1 + 1;`

Control statements `if`, `while`, `for`, ...

Each statement is followed by a semicolon `;`

Control flow

Control flow statements: `if`, `else`, `while`, `try`, ...

```
if (velocity > 0) {
  speed = velocity;
} else {
  speed = -1 * velocity;
}

for (int i = 0; i < a.length; a++) {
  total = total + a[i];
}
```

Conditionals: if-then-else

```java
if (amount < balance) {
   // then branch
   balance = balance - amount;
} else {
   // else branch (optional)
   System.out.println(
     "Cannot withdraw amount!");
}
```

Loops: while

```
int sum = 0; i = 0;
while (i < a.length) {
   sum = sum + a[i];
   i++;
}
// sum of all values in array 'a'
```

Loops: do-while

```
int sum = 0; i = 0;
do {
    sum = sum + a[i];
    i++;
} while (i < a.length);
// sum of all values in array 'a'
// only works if 'a' is not empty
```

Loops: for

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
}
// sum of all values in array 'a'
```

Loops: for-each

```
int sum = 0;
for (int v : a) {
    // v takes all values in array 'a',
    // one per iteration
    sum = sum + v;
}
// sum of all values in array 'a'
```

Classes and methods

Classes

A class consists of methods and variables:

```
class Interest {   // in a file Interest.java
  private static double[] rates = { ... };    // class variable

  public static double interestYear(int year) { // class method
    return rates[year-2010];
  }

  public static void main(String[] args) {    // entry point
    int year = 2018;                          // of the program
    double interest = interestYear(year - 3);
    System.out.println("The interest for"
      + (year-3) + " is " + interest);
  }
}
```

Methods

```
public static double interestYear(int year) {
  return rates[year-2010];
}
```

- `public` defines the method's visibility
- `static` identifies a class method
- `double` is the return type
- `int year` is the argument (also called parameter) declaration
- `return rates[year-2010];` is the method's body

Private vs public methods and variables

- A private member can only be used in other methods in the same class.
- A public member can be used from any class.
- (A protected member can be used from any class in the same package.)

Unless there is a good reason, most methods should be private!

Formal vs actual parameters

Method declaration:

```
double interestYear(int year) { ... }
```

`year` is the formal argument

Method call:

```
double interest = interestYear(year - 3);
```

`year - 3` is the actual argument

How method calls work

IN GENERAL	IN THE EXAMPLE
the (actual) argument is evaluated	evaluate `year - 3` to `2015`
the (formal) argument is initialized	initialize `year` to `2015`
the method body is executed	execute `interestYear(2015)`
when execution reaches a `return`, the argument of `return` is evaluated	evaluate `return rates[year-2010]` to `return 0.03`
result becomes value of the method call	`interestYear(year - 3)` evaluates to `0.03`
execution continues in the caller	variable `interest` is updated to `0.03`

How method calls work: primitive types

Assigning a new value to the formal argument does not affect the actual argument...

```java
void dontSet(int v)
{
  v = 10;
}
```

```java
int x = 0;
dontSet(x);
// x is still 0
```

`int v` `int x`

How method calls work: primitive types

Assigning a new value to the formal argument does not affect the actual argument...

```
void dontSet(int v)
{
  v = 10;
}
```

```
int x = 0; ⟸
dontSet(x);
// x is still 0
```

How method calls work: primitive types

Assigning a new value to the formal argument does not affect the actual argument...

```
void dontSet(int v)
{
  v = 10;
}
```

```
int x = 0;
dontSet(x); ⟸
// x is still 0
```

0
int v

0
int x

How method calls work: primitive types

Assigning a new value to the formal argument does not affect the actual argument...

```
void dontSet(int v)
{
  v = 10;  ⇐
}
```

```
int x = 0;
dontSet(x);  ⇐
// x is still 0
```

10	0
int v	int x

How method calls work: primitive types

Assigning a new value to the formal argument does not affect the actual argument...

```
void dontSet(int v)
{
  v = 10;
}
```

```
int x = 0;
dontSet(x);
// x is still 0
```

⟸

10
int v

0
int x

How methods work: reference types

...but modifying a variable of reference type can still change the value.

```
void set(int[] a)
{
  a[0] = 10;
}
```

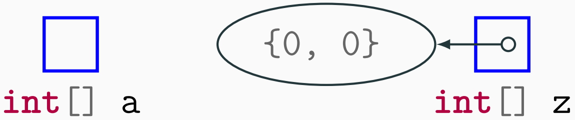
```
int[] z = {0, 0};
set(z);
// z[0] is 10
```

How methods work: reference types

…but modifying a variable of reference type can still change the value.

```
void set(int[] a)
{
  a[0] = 10;
}
```

```
int[] z = {0, 0};   ⟸
set(z);
// z[0] is 10
```

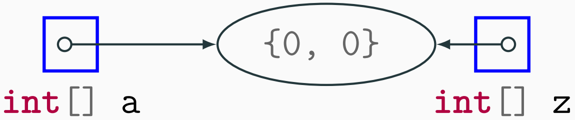


How methods work: reference types

...but modifying a variable of reference type can still change the value.

```java
void set(int[] a)
{
  a[0] = 10;
}
```

```java
int[] z = {0, 0};
set(z); ⟸
// z[0] is 10
```

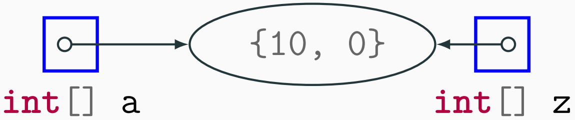


How methods work: reference types

...but modifying a variable of reference type can still change the value.

```
void set(int[] a)
{
  a[0] = 10;  ⟸
}
```

```
int[] z = {0, 0};
set(z);  ⟸
// z[0] is 10
```



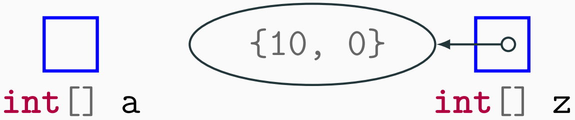
How methods work: reference types

...but modifying a variable of reference type can still change the value.

```
void set(int[] a)
{
  a[0] = 10;
}
```

```
int[] z = {0, 0};
set(z);
// z[0] is 10
```

⟸



Exceptions

Exceptions signal unusual (often erroneous) conditions.

Use a try-catch block to deal with ('handle') exceptions.

```
int n;  // Why declare 'n' outside try block?
Scanner sc = new Scanner(System.in);
try {
   n = sc.nextInt();    // may throw exception
   System.out.println("Found integer " + n);
} catch (InputMismatchException e) {
   System.out.println("Invalid integer as string!");
} finally {
   sc.close();
}
```

Kahoot: Recap of part I

What's next?

Next lecture on Thursday at 15:00:
Objects and classes.

To do:

- Read the book:
  - Today: chapter 1-7
  - Next lecture: chapter 8
- Start on lab #5