Introduction to Programming

René Thiemann

Institute of Computer Science
University of Innsbruck

WS 2008/2009

Outline

- A first Java-program
- Basic types and type conversion
  - Numbers
  - Booleans
  - Strings
  - Type conversions
- Controlling execution
  - Conditions
  - Loops
- Arrays

Outline

- A first Java-program
- Basic types and type conversion
  - Numbers
  - Booleans
  - Strings
  - Type conversions
- Controlling execution
  - Conditions
  - Loops
- Arrays

Calculating a reduced price

Example1.java

```java
import static lib.IO.*;  // import for printing

public class Example1 {
    public static void main(String[] args) {
        double price, discount;   // declare two reals
        price = 62.00;            // assign values
        discount = 10;
        price = price * (100 - discount) / 100;
        print("The reduced price is ");
        println(price);  // and output the new price
    }
}
```

output:
```
The reduced price is 55.80
```

Structure of a Java-source-file

- class declaration: `public class Example1 { /*method−decls */}`
  - each source-file consists of exactly one class
  - each class has a name, here: Example1
  - the name of the file must be the class-name + .java (Example1.java)
  - a class consists of several sub-programs (methods)
  - the execution always starts in the main-method
- import statement: `import static lib.IO.*;`
  each file has several import statements to access methods of libraries
  here, all methods print, println, ... of the library lib.IO are imported
- **public**, **class**, **import**, ... are keywords and cannot be changed
- main, lib, IO, ... are names (identifier) which can be freely chosen

Structure of a method

- method declaration:
  `public static void main(String[] args) { /* body */ }`
  - each method has an identifier, here the identifier is main
  - a method can have arguments (String [] args) which are explained later
  - the body is a list of basic instructions (so called statements)
  - (currently just ignore the keywords **public**, **static**, and **void**)

Statements

- variable declarations: `double price, discount;`
  - each variable has to be declared and initialized before it is used
  - each variable has a type which is written in front of a variable declaration
  - here, two variables price and discount of type **double** (real numbers) are declared
- assignments: `price = price * (100 - discount) / 100;`
  - the variable on the left of "=" gets the value of the expression on the right of "="
  - so, if currently price has value 62 and discount has value 10 then first the right-hand side is evaluated to $62 \times (100 - 10)/100 = 55.80$ which will be the new value of price
- method-calls: `println(price)`
  - method-calls execute sub-programs of the class or of some library
  - here, the method println is called to display the value of price
  - (the difference of print and println is that the latter also jumps to the beginning of the next line)

Being interactive

```
import static lib.IO.*;  // import for reading
public class Example2 {
    public static void main(String[] args) {
        double price, discount;
        print("Please enter the price: ");
        price    = readDouble();    // read value
        print("Please enter the discount: ");
        discount = readDouble();
        price = price * (100 - discount) / 100;
        print("The reduced price is ");
        println(price);
    }
}
```

output:

```
Please enter the price:  19.99
Please enter the discount:  12.5
The reduced price is 17.49
```

A note on style

- class/variable/method-identifiers consist of letters (a-zA-Z), digits, and "_"
- identifiers must be different from keywords (**public**, **class**, . . .)
- identifiers must not start with a number
- class identifiers should start uppercase, variables lowercase
- identifiers should be meaningful but not too long
  - bad: a, b, c, d, e, f, . . .
  - good: discount, price, salary, . . .
  - bad: discountOfACustomerNamedJohnDoe, . . .
- capitalize when starting a new word
  - bad: specialdiscount, nOrmALDIScouNT
  - good: specialDiscount, normalDiscount

  alternatively, use underscores: special_discount, normal_discount
- use fixed indention (standard for Java: 4 spaces)
  - bad:

    ```
    public class C {public static void main(String[] a){ print ("a");}}
    ```
  - good: the other examples in this lecture

Outline

- A first Java-program
- Basic types and type conversion
  - Numbers
  - Booleans
  - Strings
  - Type conversions
- Controlling execution
  - Conditions
  - Loops
- Arrays

Built-in datatypes: numbers

type($\subseteq \mathbb{Z}$)	*range*	*range*	*bits*
byte	$-128 \dots 127$	$-2^7 \dots 2^7 - 1$	8
short	$-32768 \dots 32767$	$-2^{15} \dots 2^{15} - 1$	16
char	$0 \dots 65535$	$0 \dots 2^{16} - 1$	16
int	$-2147483648 \dots 2147483647$	$-2^{31} \dots 2^{31} - 1$	32
long	$\approx 9.22 \cdot 10^{18} \dots \approx -9.22 \cdot 10^{18}$	$-2^{63} \dots 2^{63} - 1$	64

`short x = 32767; x = x + 1; print(x);` outputs

type($\subseteq \mathbb{R}$)	*range*	*min*	*precision*	*bits*
float	$[+/-]3.4 \cdot 10^{38}$	$1.4 \cdot 10^{-45}$	23	32
double	$[+/-]1.8 \cdot 10^{308}$	$4.9 \cdot 10^{-324}$	52	64

`float x = 2147483647; float y = x - 10; print(x-y);` outputs

Arithmetic

Standard mathematical operations

operator	operation	example
+	addition	
−	subtraction	`int x = 5 - 3;`
−	negation	`int x = 5; x = -x;`
*	multiplication	
/	integer-division	`int x = 10; x = x / 3;` yields
/	division	`float x = 10; x = x / 3;` yields
%	remainder	`int x = 10; x = x % 3;` yields

Shortcuts

operator	short version	long version
+=,−=,*=,...	`x -= 5;`	`x = x - 5;`
++,−−	`x++;`	`x = x + 1;`

recall standard precedence: `5 + 3 * 7` is the same as `5 + (3 * 7)`

Built-in datatype: Boolean

- a Boolean is a truth-value: true or false
- Boolean expressions evaluate to a Boolean and are build as follows
  - using the constants **true** or **false**
  - using variables of type Boolean
  - combining arithmetic expressions with comparison operators == (equality), != (non-equality), > (greater), >=, <, ...
  - using one of the Boolean operators ! (negation), || (disjunction), && (conjunction)
    b1 || b2 is true iff at least one of b1 or b2 is true
    b1 && b2 is true iff both b1 and b2 are true
  - || and && are evaluated lazily in their second argument:
    whenever b1 is evaluated to true then b2 is not evaluated in b1 || b2

  ⇒
- binding precedence: binary Boolean operators < comparison operators < ! < arithmetic operators

⇒ 3 > y && !4 == x && b

Datatype: String

- a String is sequence of characters
- two ways to build strings:
  - using double-quotes: `"some text"`
  - using the string concatenation operator +: s1 + s2

  example: `String s = "look "; s = s + "at this"; print(s);`
  outputs
- problem: construct string like I said "hello".

⇒ solution: use special escape sequences starting with \

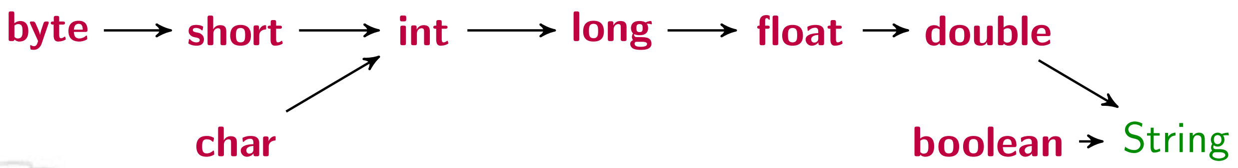
escape sequence	character
\"	"
\\	\
\n	newline
\t	tabulator

Automatic type conversion

- consider

```
float f = 4; int i = 3; byte b = 127; print(b + i); print(f / i);
```

- problem: arithmetic operation on different types
  - b+i yields -126 or 130: overflow or not?
  - f/i yields 1 or 1.33: integer-division or not?
- solution: automatic conversion of operands into greater compatible type



⇒ above program yields values

- conversion into String is only possible with operator + where one of the operands is a String

⇒

⇒

⇒

Casting

- consider

```
int i = 5; byte b,c; b = i; i = 130; c = i; print(b+","+c);
```

- problem: b cannot store **int**-value, automatic conversion only yields greater type

⇒ above program is invalid, rejected by compiler

- solution: explicit conversion (casting) to lower type: (type)expr

```
int i = 5; byte b,c;
b = (byte)i; i = 130; c = (byte)i;
print(b+","+c);
```

outputs

⇒ with casting you can introduce overflows

- remark: it not possible to cast strings into numbers or Booleans
  solution: `Integer.parseInt("4")` or `Boolean.parseBoolean("true")`

Outline

- A first Java-program
- Basic types and type conversion
  - Numbers
  - Booleans
  - Strings
  - Type conversions
- Controlling execution
  - Conditions
  - Loops
- Arrays

Conditional statements

```
import static lib.IO.*;
public class Example3a {
    public static void main(String[] args) {
        print("Please enter the price: ");
        double price    = readDouble();
        print("Please enter the discount: ");
        double discount = readDouble();
        price = price * (100 - discount) / 100;
        println("The reduced price is "+price);
    }
}
```

Output:

```
Please enter the price:  50
Please enter the discount:  120
The reduced price is -10.00
```

- up to now: no control of execution

⇒ how can we forbid to use discounts over 100 or below 0 percent?

Conditional statements

solution: use conditional statement where
a condition is just a Boolean expression

```
if (/* condition */) { /* stmts1 */ } else { /* stmts2 */ }
```

the execution of a conditional statement works as follows

- if condition is satisfied (resulting value is true) then the first statements are executed
- otherwise the second statements are executed

shortcut:

```
if (/* condition */) { /* statements */ }
```

is identical to

```
if (/* condition */) { /* statements */ } else { }
```


Conditional statements

```
import static lib.IO.*;
public class Example3b {
    public static void main(String[] args) {
        print("Please enter the price: ");
        double price    = readDouble();
        print("Please enter the discount: ");
        double discount = readDouble();
        if (discount < 0 || discount > 100) {
            println("The discount is invalid");
        } else {
            price = price * (100 - discount) / 100;
            println("The reduced price is "+price);
        }
    }
}
```

output:

```
Please enter the price:  50
Please enter the discount:  120
The discount is invalid
```

Nesting of conditional statements

```
import static lib.IO.*;
public class Example3c {
    public static void main(String[] args) {
        print("Please enter the price: ");
        double price    = readDouble();
        print("Please enter the discount: ");
        double discount = readDouble();
        if (discount > 100) {
            println("The discount is too high");
        } else {
            if (discount < 0) {
                println("The discount is too low");
            } else {
                price = price * (100 - discount) / 100;
                println("The reduced price is "+price);
            }
        }
    }
}
```


Loop statements

- up to now: each statement is executed at most once

⇒ output cannot get longer than program + input

- consider problem:
  - given capital (10,000), ask for interest rate and number of years
  - output for each year the corresponding capital

```
Please enter the interest rate:  4.5
Please enter the number of years:  6
capital after 0 year(s):  10000.00
capital after 1 year(s):  10450.00
capital after 2 year(s):  10920.25
capital after 3 year(s):  11411.66
capital after 4 year(s):  11925.19
capital after 5 year(s):  12461.82
capital after 6 year(s):  13022.60
```

While-loops

a while-loop is a statement of the following form

```
while (/* condition */) { /* body statements */ }
```

the execution of a while-loop works as follows

- if the condition is satisfied then the body statements are executed once and afterwards the whole procedure is iterated once again
- otherwise the execution of the while-loop is finished

Capital growth over many years

```
import static lib.IO.*;
public class Example4a {
    public static void main(String[] args) {
        double capital = 10000;
        print("Please enter the interest rate: ");
        double factor = 1 + readDouble()/100;
        print("Please enter the number of years: ");
        int years = readInt();
        int year = 0;
        while (year <= years) {
            print("capital after "+year+" year(s): ");
            println(capital);
            capital *= factor;
            year++;
        }
    }
}
```

Watching the execution

Do- and for-loops

- a do-loop is similar to the while loop, but body is executed before condition is checked

```
do { /* body stmts */ } while (/* condition */ );
```

  equivalent program with while-loop

```
/* body stmts */ while (/* condition */) { /* body stmts */ }
```

- a for-loop has additional an initialization and iteration statement

```
for (/*init stmt*/; /* condition */; /*iter stmt*/) {
  /* body statements */
}
```

  equivalent program with while-loop

```
/* init stmt */
while (/* condition */) {
  /* body stmts */
  /* iter stmt */
}
```

Capital growth over many years with do-loop

```
import static lib.IO.*;
public class Example4b {
    public static void main(String[] args) {
        double capital = 10000;
        print("Please enter the interest rate: ");
        double factor  = 1 + readDouble()/100;
        print("Please enter the number of years: ");
        int years = readInt();
        int year = 0;
        do {
            print("capital after "+year+" year(s): ");
            println(capital);
            capital *= factor;
            year++;
        } while (year <= years);
    }
}
```


Capital growth over many years with for-loop

```
import static lib.IO.*;
public class Example4c {
    public static void main(String[] args) {
        double capital = 10000;
        print("Please enter the interest rate: ");
        double factor  = 1 + readDouble()/100;
        print("Please enter the number of years: ");
        int years = readInt();
        for (int year = 0; year <= years; year++) {
            print("capital after "+year+" year(s): ");
            println(capital);
            capital *= factor;
        }
    }
}
```

More control over loop execution

Outline

- A first Java-program
- Basic types and type conversion
  - Numbers
  - Booleans
  - Strings
  - Type conversions
- Controlling execution
  - Conditions
  - Loops
- Arrays

The need for arrays

- up to now: number of storable values = number of variables in code
- but one may need to process and store more input
- simple example: palindrom sequences (sequence = reversed sequence)

```
length of sequence: 5
1. number: 7
2. number: 2
3. number: 5
4. number: 2
5. number: 7
[7, 2, 5, 2, 7]
palindrom
```

```
length of sequence: 5
1. number: 7
2. number: 2
3. number: 5
4. number: 2
5. number: 6
[7, 2, 5, 2, 6]
no palindrom
```

- program needs to store all numbers

⇒ use arrays

Palindrom program

```
print("length of sequence: ");
int n = readInt();
int[] numbers = new int[n];
for (int i=0; i<n; i++) {
    print((i+1)+". number: ");
    numbers[i] = readInt();
}
println(numbers);
boolean palindrom = true;
for (int i=0; i<n/2; i++) {
    if (numbers[i] != numbers[n-1-i]) {
        palindrom = false;
        break;
    }
}
if (!palindrom) {
    print("no ");
}
println("palindrom");
```

Dealing with arrays

- array: sequence of fixed length of elements of some type
  - `["foo", "bar"]` – array of strings, length is 2
  - `[0, 4, −2]` – array of integers, length is 3
- specifying the type of an array: `elementtype []`
  - `String [] array1;` – array of strings
  - `int [] array2;` – array of integers
  - `int [] [] array3;` – array of array of integers
- expressions of array type
  - the non-existing array: `null`
  - creation of new arrays: `new elementtype[length]`
    - new memory will be allocated
    - each element in array is initialized by default value
      numbers:0 **boolean**: false String: **null** arrays: **null**

    ⇒ `new int[2]` yields array [0,0], `new int [2][]` yields array [**null**, **null**]
    - good praxis: perform own initialization of elements

Dealing with arrays, ctd.

- accessing the elements of an array: `somearray[index]`

  if array has length *n* then indices must be between 0 and n-1!

```
int[] a = new int[3]; // array for 3 integers
a[0] = 8; a[1] = a[0]+7; a[2] = a[1*1] - 5;
a[3] = 5; // index-out-of-bounds exception
a = null;
a[0] = 8; // null-pointer exception
```

- accessing the length of an array: `somearray.length`

Visualization: swapping elements

given integer array a, two indices i, j

problem: swap elements at position i and j

```
int i = 0; int j = 2; int x; int[] a;
```

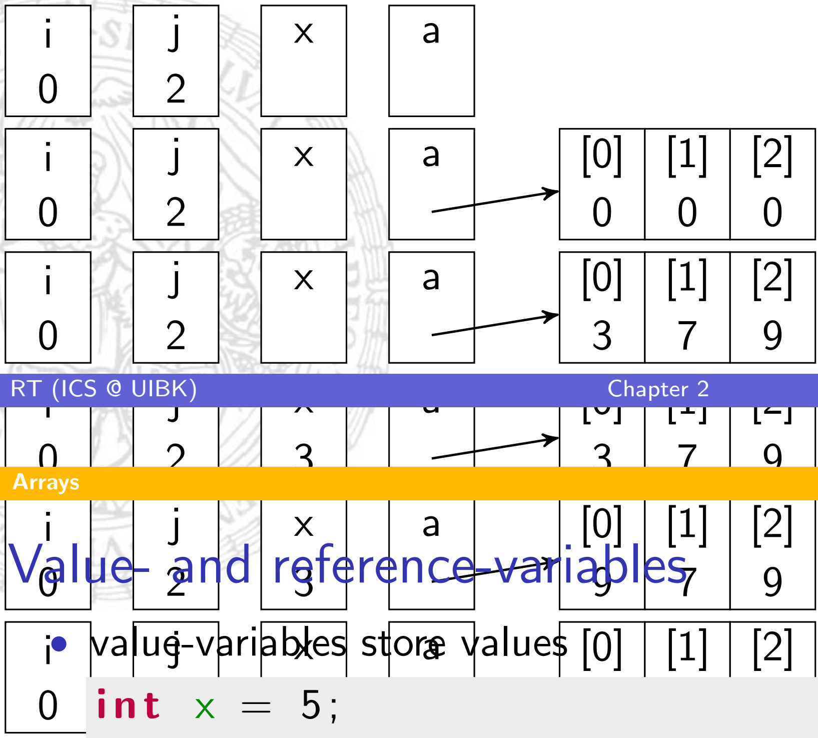
```
a = new int[3];
```

```
a[0] = 3; a[1] = 7; a[2] = 9;
```

```
x = a[i];
```

```
a[i] = a[j];
```

```
a[j] = x;
```



Value- and reference-variables

- value-variables store values

```
int x = 5;
int y = x;
y++;
```

afterwards x stores value 5 and y stores value 6

- reference-variables only store a reference to an object

```
int[] a = new int[] {3,4,7};
int[] b = a;
b[0]++;
```

afterwards a and b reference the same array; this array contains the numbers 4,4,7

- in Java, only variables of primitive datatypes (starting with lowercase letters, e.g., `int`, `double`, `boolean`, ...) are value-variables
- all other variables are reference variables (`String` , arrays, . . .)

References and side-effects

- the assignment `a = b` for arrays only sets the reference a to the object that is referenced by b

⇒ no new memory is allocated

⇒ afterwards, each change of the elements in a will also change b and vice versa, e.g., `a[3] = 7;` will also change b[3] to 7!

this phenomenon is called side-effect

- whether a and b reference the same object can be checked by `a == b`

References and side-effects, example

```
int[] a = new int[] {0,5,7}; int b[] = null;
b = a; // now a and b point to the same array
a[0] = 1; b[1] = 3; // side-effect
a = new int[] {2,7,9};
a[0] = 5; b[0] = 4; // no side-effect
b = a; // old array no longer accessible
```

Example illustrated

Side-effects

- side-effects are often not desired and can lead to subtle bugs (errors)
- consider the code-fragment to check whether an array is sorted

```
int[] a = ... // array to be checked
int[] b = a;
java.util.Arrays.sort(b) // method sorts b
boolean sorted = true;
for (int i=0; i<a.length; i++) {
  if (a[i] != b[i]) {
    sorted = false;
    break;
  }
}
```