Software and Programming I

Branching and Boolean Expressions

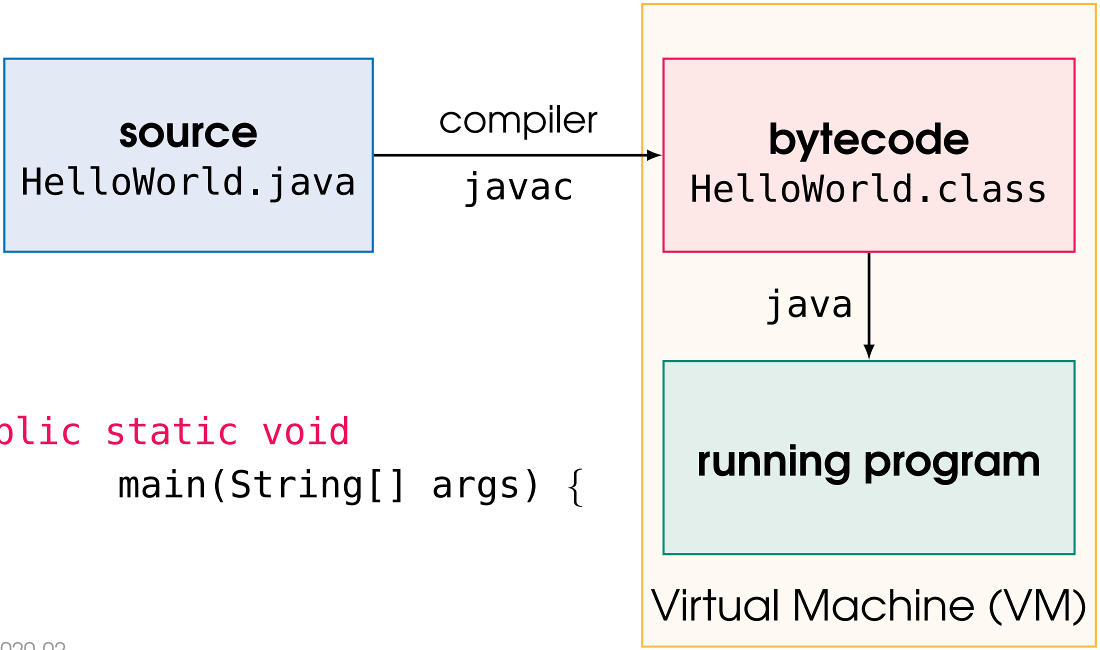
Roman Kontchakov / Carsten Fuhs

Birkbeck, University of London

Outline

- The `if` statement
- Comparing numbers and strings
- Nested branches
- Boolean variables and expressions
    - Sections 3.1–3.4, 3.7
- Return Statement
    - Section 5.4

Java Compilation and JRE



```
public static void
        main(String[] args) {
...
}
```

Python:
```
    n = "World"
    print("Hello, " + n + "!")
```

My First Program

```
/* HelloWorld.java
   Purpose: printing a hello message on the screen
*/
public class HelloWorld {
    // each program is a class (week 6)
    // almost everything in Java is an object
    public static void main(String[] args) {
        String n = "World";
        System.out.println("Hello, " + n + "!");
    }
}
```

NB. watch out for **semicolons** — they are compulsory

NB. names and reserved words are **case-sensitive**

Methods

Python:
```
def  sq(x):
     Python code
```

- A **method** is a named sequence of instructions

method name

```
public static int sq(int x) {
      Java code (sequence of instructions)
}
```

parameter
(type and name)

type of **return value**

`void` means the method does not return any value

- **Parameter values** are supplied when a method is called
- The **return value** is the result that the method computes
- Method ≈ algorithm ≈ function (in Python)

NB: until week 6, all methods will be `public static`

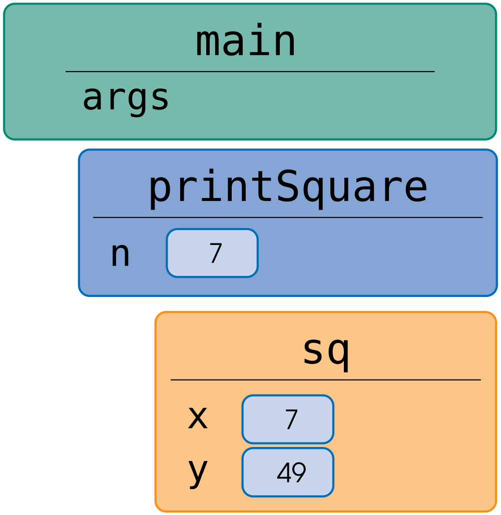
Example 2: $y = x^2$ as a Method

```java
public class PrintSquares {
    public static void main(String[] args) {
        printSquare(7);
        printSquare(9);
    }
    public static int sq(int x) { // x is a parameter
        int y = x * x; // compute x^2
        return y; // return the value
    }
    public static void printSquare(int n) {
        System.out.println(n + "^2=" + sq(n));
    }
}
```

the output:

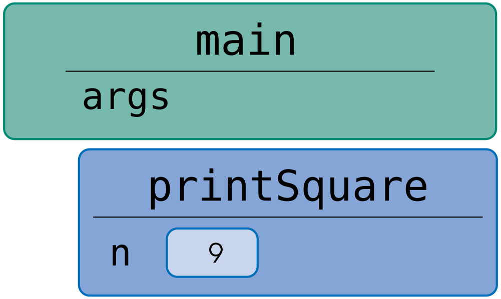
```
7^2=49
9^2=81
```

Method Call Stack



```
printSquare(7);


public static void printSquare(int n) {
    System.out.println(n + "^2=" + sq(n));


public static int sq(int x) {
    int y = x * x;
    return y;
}
```

Method Call Stack



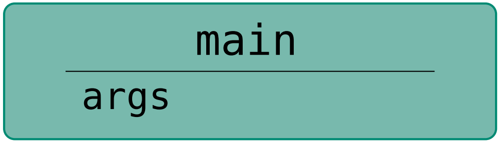
```
printSquare(7);

printSquare(9);
```

```
7^2=49
9^2=81
```

```
public static void printSquare(int n) {
    System.out.println(n + "^2=" + sq(n));
}
```

evaluated to **81**

Method Call Stack



```
printSquare(7);

printSquare(9);
```

```
7^2=49
9^2=81
```

Return Statement

The `return` statement

(1) terminates a method call

(2) yields the method result

```java
public static double cubeVolume(double sideLength) {
    if (sideLength < 0)
        return 0;
    // more code
    return sideLength * sideLength * sideLength;
}
```

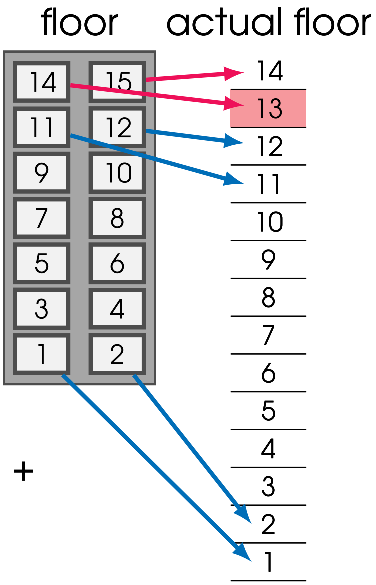
NB: if a method has no return value (`void`),

then it can contain `return;` only (without any value)

The if Statement

The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed

```
Scanner s = new Scanner(System.in);
int floor = s.nextInt();
int actualFloor;
if (floor > 13) {
     actualFloor = floor - 1;
}
else {
     actualFloor = floor;
}
System.out.println("Actual floor: " +
                              actualFloor);
```



Statement Blocks

A **block** is a group of 0 or more statements between balanced { and }

(in Python: a group of statements indented at the same level)

A **block** can be used anywhere a **single statement** is allowed

the following is equivalent to the code on p. 10, lines 3–9

```
int actualFloor;
if (floor > 13)
    actualFloor = floor - 1;
else
    actualFloor = floor;
```

The Else Branch is Optional

The `else` branch is **optional**

```
int discountedPrice = originalPrice;
if (originalPrice > 100)
    discountedPrice = originalPrice - 10;
```

is **equivalent** to

```
int discountedPrice;
if (originalPrice > 100)
    discountedPrice = originalPrice - 10;
else
    discountedPrice = originalPrice;
```

If: Common Mistakes

; is a valid Java statement — it does nothing

NB: do not put the semicolon after `if(...)`

```java
int discountedPrice = originalPrice;
if (originalPrice > 100); { // empty statement in if
    // LOGICAL ERROR: this block is executed anyway
    discountedPrice = originalPrice - 10;
}   // it is, however, NOT a syntax (compile) error
```

Types of Errors

- **compile-time error**: error in syntax (Java grammar) or type detected by the compiler: an error message is produced and **no bytecode** is generated

```
int i = "string"; //incompatible types
```

- **run-time error**: the program is compiled and runs, but the JVM **terminates** execution when the error is encountered

```
int i = 1/0; //ArithmeticException: / by zero
//StringIndexOutOfBoundsException
String s = "tomato".substring(0, -2);
```

- **logical error**: the program is compiled and runs, but the outputs are **not** as expected

Constants

- the reserved word `final` ensures that the value of the variable never changes:

```
final double BOTTLE_VOLUME = 2;
```

- use names for constants and avoid “magic numbers”

- “all uppercase with words separated by _” is a coding **convention**

  Java does not check it!

Comparing Numbers

relational operators `>`, `>=`, `<`, `<=`, `==`, `!=`

are applicable to `int`, `double` and other numerical datatypes

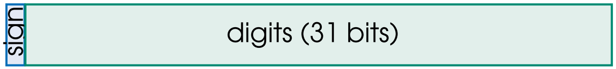
NB: NOT $\geq$ but why not, e.g., => instead of >=? (assignment operators, week 3)

relational operators return `boolean` values (`true` or `false`),
which, for example, can be stored in `boolean` variables

```
int floor = 2;
int top = 12;
boolean over = floor > top;
```

Integer Datatypes

- `int` 32-bit integers



hexadecimal literals

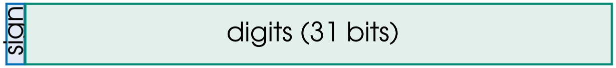
`0x00000000`	is	0
`0x00000001`	is	1
`0xFFFFFFFF`	is	-1
`0x7FFFFFFF`	is	2,147,483,647 = `Integer.MAX_VALUE`
`0x80000000`	is	-2,147,483,648 = `Integer.MIN_VALUE`

NB: do not use `,` in Java as a digits separator, use the underscore (_) instead: e.g., `Integer.MIN_VALUE` is `-2_147_483_648`

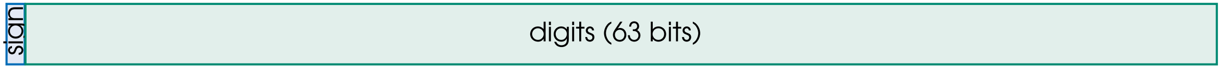
? What is the value of `Integer.MAX_VALUE + 1`?

Integer Datatypes (2)

- `int` 32-bit integers from -2,147,483,648 to 2,147,483,647

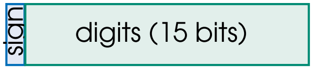


- `long` 64-bit integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

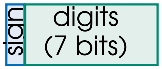


NB: use suffix L for `long` literals: e.g., `4_000_000_000L`

- `short` 16-bit integers from -32,768 to 32,767



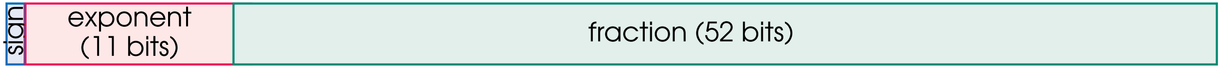
- `byte` 8-bit integers from -128 to 127



(Python 3 uses "arbitrary-precision" integers)

IEEE Floating Point Numbers

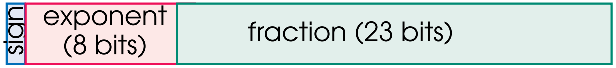
- `double`: $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$
  with approx. 15 decimal digits
  3 decimal digits (1000) $\approx$ 10 binary digits (1024)



NB: `double` literals: $\texttt{12.3} = \texttt{1.23e1} = \texttt{0.}\boxed{\texttt{123}}\texttt{e}\boxed{\texttt{2}} = \texttt{123e-1}$

$$12.3 = 1.23 \times 10 = 0.123 \times 10^2 = 123 \times 10^{-1}$$

- `float`: $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
  with approx. 7 decimal digits



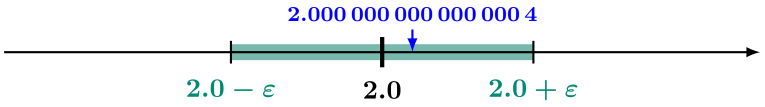
NB: `float` should never be used for precise values, e.g., currency;
use `java.math.BigDecimal` class instead

Comparing Floating-Point Numbers

arithmetic operations on floating-point numbers
cannot be precise

use small $\varepsilon = 10^{-14}$ to compare floating numbers:

```java
final double EPSILON = 1e-14;  // 0.000_000_000_000_01
double r = Math.sqrt(2.0);
// r*r is 2.000_000_000_000_000_4 rather than  2.0
if (Math.abs(r * r - 2.0) < EPSILON) {
    System.out.println("Math.sqrt(2.0) squared " +
                                        "is approx 2.0");
}
```



the interval contains all x
with $-\varepsilon < x - 2 < \varepsilon$
or, equivalently, $|x - 2| < \varepsilon$

Strings

- strings are sequences of characters:

$$\texttt{String name = }\underbrace{\texttt{"foo bar"}}_{\texttt{literal}}\texttt{;}$$

A string is **not interpreted**, e.g., the compiler does not replace **"$h"** with the value of variable $h

- the `length` method yields the number of characters in the string:

```
int n = name.length();
```

(in Python: `n = len(name)`)

  the empty string `""` is of **length** 0

- string **positions** are counted starting with **0**

	f	o	o		b	a	r
position	0	1	2	3	4	5	6

NB: there are **no negative** positions

Comparing Strings

`"Tomato".substring(0,3).equals("Tom")` is `true`

NB: == is **not** useful for strings in Java: (in contrast to Python)

`"Tomato".substring(0,3) == ("Tom")` is `false`

why? `String`s are objects (week 6)

`s1.compareTo(s2)` compares strings **lexicographically**:

`s1.compareTo(s2) < 0` if `s1` comes before `s2` in the dictionary

`s1.compareTo(s2) == 0` if `s1` equals `s2`

`s1.compareTo(s2) > 0` if `s1` comes after `s2` in the dictionary

NB: `equals` and `compareTo` are **method names** in Java

in Python, use `s1 < s2`, `s1 == s2` and `s1 > s2`, respectively

Characters

`char` is a 16-bit numerical datatype

`' '` is `0x20` (32), `'\n'` is `0x0A`, bell is `0x07`, `'!'` is `0x21`,
`'0'–'9'` is `0x30–0x39`, `'A'–'Z'` is `0x41–0x5A`, `'a'–'Z'` is `0x61–0x7A`

Unicode 12.1 (May 2019) contains 137,994 characters covering 150 modern and historic scripts, as well as multiple symbol sets and emoji
Unicode includes ASCII (as the 128 characters of the Basic Latin range)

NB. do not confuse characters (`'H'`) and strings containing a single character (`"H"`)
(Python has no special datatype for characters: there, `'H'` and `"H"` are both strings)

Class `Character` provides the following methods:

- `Character.isDigit(ch)`: `'0','1',...,'9',...`
- `Character.isLetter(ch)`: `'A','B',...,'Z','a','b',...,'z',...`
- `Character.isUpperCase(ch)`: `'A','B',...,'Z',...`
- `Character.isWhiteSpace(ch)`: `' ','\n',...`

Nested Branches and the Dangling else

What is **wrong** in the following example?

```java
double shippingCharge = 5.00;
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is
                                // more expensive
else
    shippingCharge = 20.00; // as are foreign
                            // shipments
```

NB: the Java compiler does not care about indentation

(in contrast to Python)

Nested Branches and the Dangling else

```
double shippingCharge = 5.00;
if (country.equals("USA")) {
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is
                                // more expensive
}
else
    shippingCharge = 20.00; // as are foreign
                            // shipments
```

NB: use curly brackets {} to avoid the **dangling else** problem

Boolean Variables and Operators

The Boolean type `boolean` has two values, `false` and `true`

three Boolean operators that combine conditions:

`&&` (and), `||` (or), `!` (not)

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

A	!A
false	true
true	false

Lazy (Short-Circuit) Evaluation

&& and || are computed using **lazy evaluation**:
stops as soon as the result is known

```
price/quantity < 10 && quantity > 0
```

can result in a run-time error (division by 0) if `quantity` is `0`

```
quantity > 0 && price/quantity < 10
```

never divides by 0

NB: do not confuse with & and |

Conditional Operator

conditional operator ? :

lets us write simple conditional statements as **expressions**

```
int actualFloor = (floor > 13) ? floor - 1 : floor;
```

an expression

is equivalent to

```
int actualFloor;
if (floor > 13)
    actualFloor = floor - 1;
else
    actualFloor = floor;
```

Take Home Messages

- The `if` statement allows a program to carry out different actions depending on the data to be processed
- Relational operators are used to compare numbers
- Use `equals` and `compareTo` to compare strings
- `else` is matched with the preceding `if`
- && and || are computed lazily
- The `return` statement terminates a method call and yields the method result