

Introduction to Programming - Concepts and Tools

Lecture 1

Programming Fundamentals and Elementaries of Java

Overview Lecture 1

- Course presentation
 - Practical information
- Programming fundamentals
 - What is programming?
 - Compiling and running Java programs
 - Elementaries of Java

Practical Information

- Course lecture: Thursday 17:00 – 19:00 in AUD4
- Exercises (Labs): Thursday 19:00 – 21:00 in room 4A54 and 4A56.
- Lecturer:
 - Nina Bohr
 - Jens Chr. Godskesen (course responsible)
- Teaching assistant: Giovanni Crudele
- Homepage www.itu.dk/courses/IPBR/E2004

Course literature

– *[JbD]: Java by Dissection – The Essentials on Java Programming*, Ira Pohl & Charlie McDowell, Addison-Wesley; 2000.

ISBN 0-201-61248-8

– Notes and slides posted on the homepage

Mandatory Assignments

10 mandatory assignments will be posed

- 8 have to be completed successfully to be allowed to sit the exam
- assignments have to be handed in on Thursdays at the start of the lab session
- hand in to the *teaching assistant*
- no assignment this and the last week of the course

Etiquette

- Check the course **homepage** frequently!
- Make **groups** of 2-3 persons already today!!!
Mandatory assignments are to be handed in one per group.
- Check that you can logon and run Java at ITU today.

What is programming?

Programming is the process of instructing a computer of how to solve a specific problem.

A (computer) **program** is a set of instructions for solving a particular problem.

Program like lists of instructions are often called **algorithms**.

GCD Algorithm

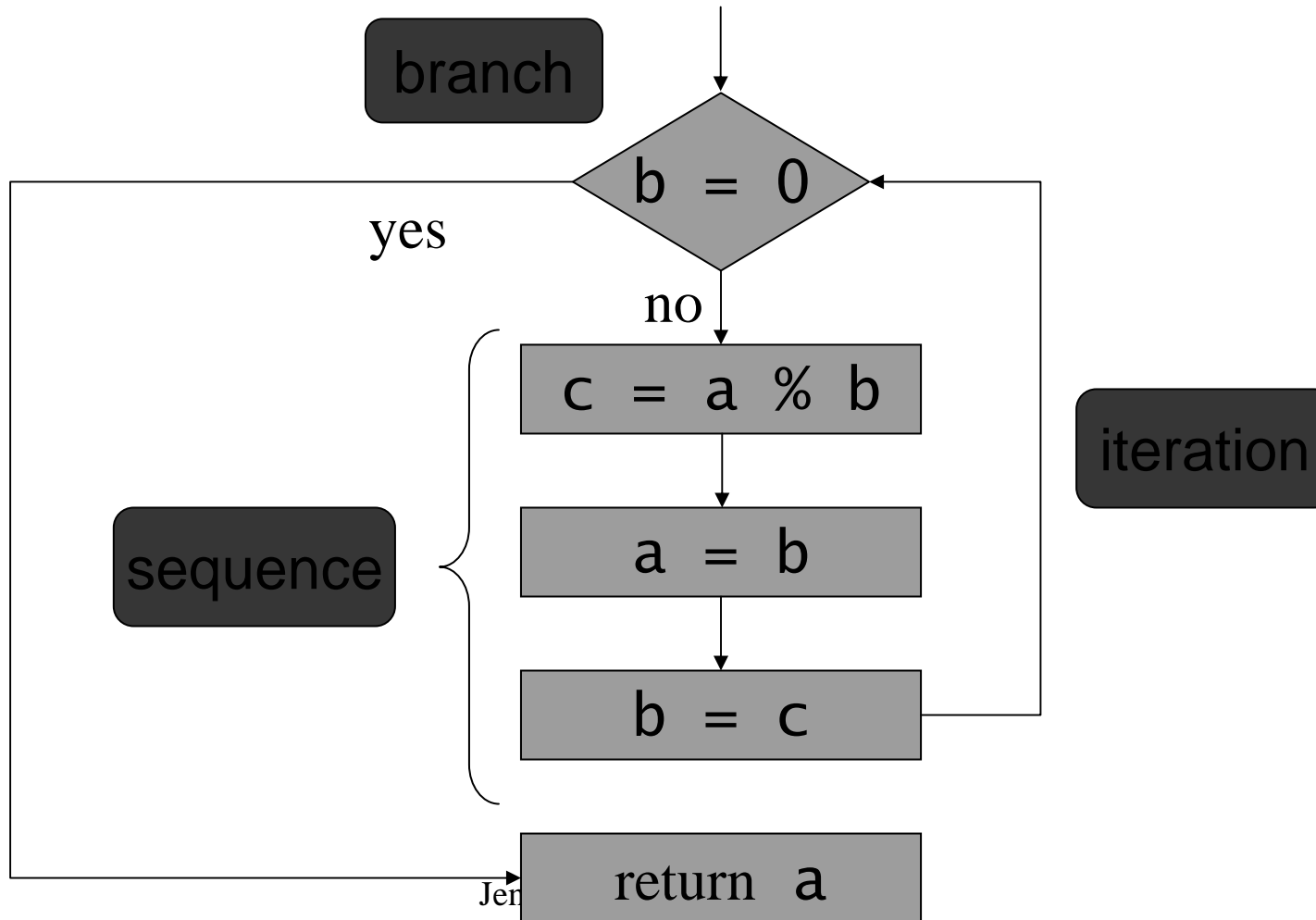
Problem: Compute the *greatest common divisor* (gcd) of two natural numbers **a** and **b** using **Euclids algorithm**. E.g. $\text{gcd}(15, 6) = 3$.

Pseudo-code: with **variables** (value holders) **a**, **b**, and **c**

1. if **b** is zero goto 6
2. assign to **c** the remainder of dividing **a** by **b**
3. assign to **a** the value of **b**
4. assign to **b** the value of **c**
5. goto 1
6. return **a**

Euclids algorithm is one of the oldest algorithms known, it appeared in around 300 BC.

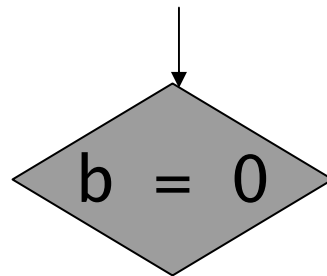
GCD Flowchart



Bench Testing

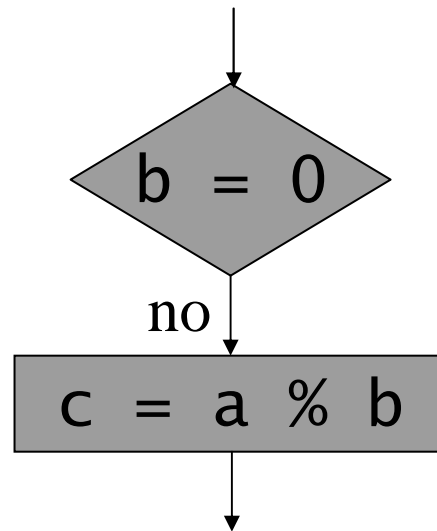
The execution of an algorithm is a **computation**.

Let's run GCD *manually* with $a = 15$ and $b = 6$:



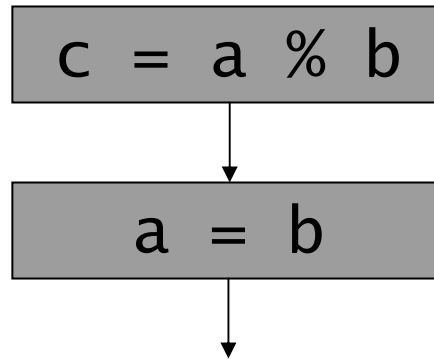
a	15						
b	6						
c							

Bench Testing



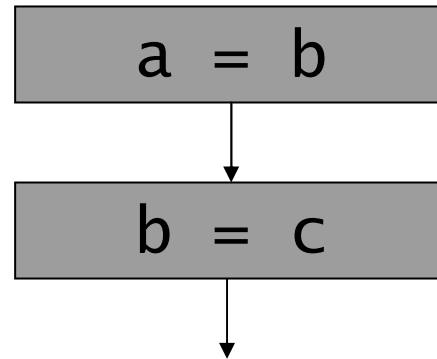
a	15	15					
b	6	6					
c		3					

Bench Testing



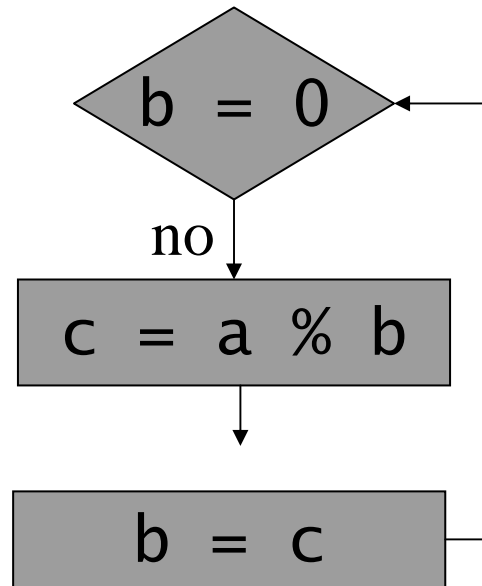
a	15	15	6				
b	6	6					
c		3					

Bench Testing



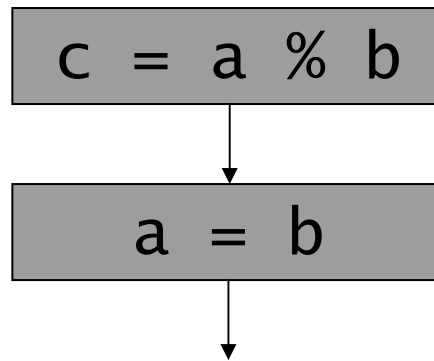
a	15	15	6	6			
b	6	6	6	3			
c		3	3				

Bench Testing



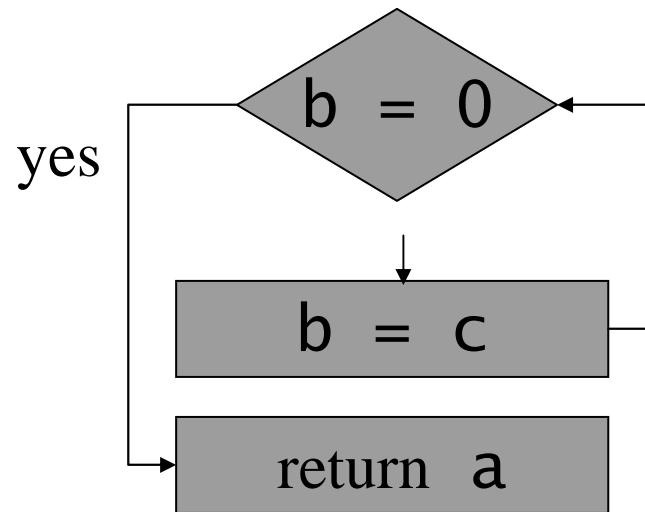
a	15	15	6	6	6		
b	6	6	6	3	3		
c		3	3	3	0		

Bench Testing



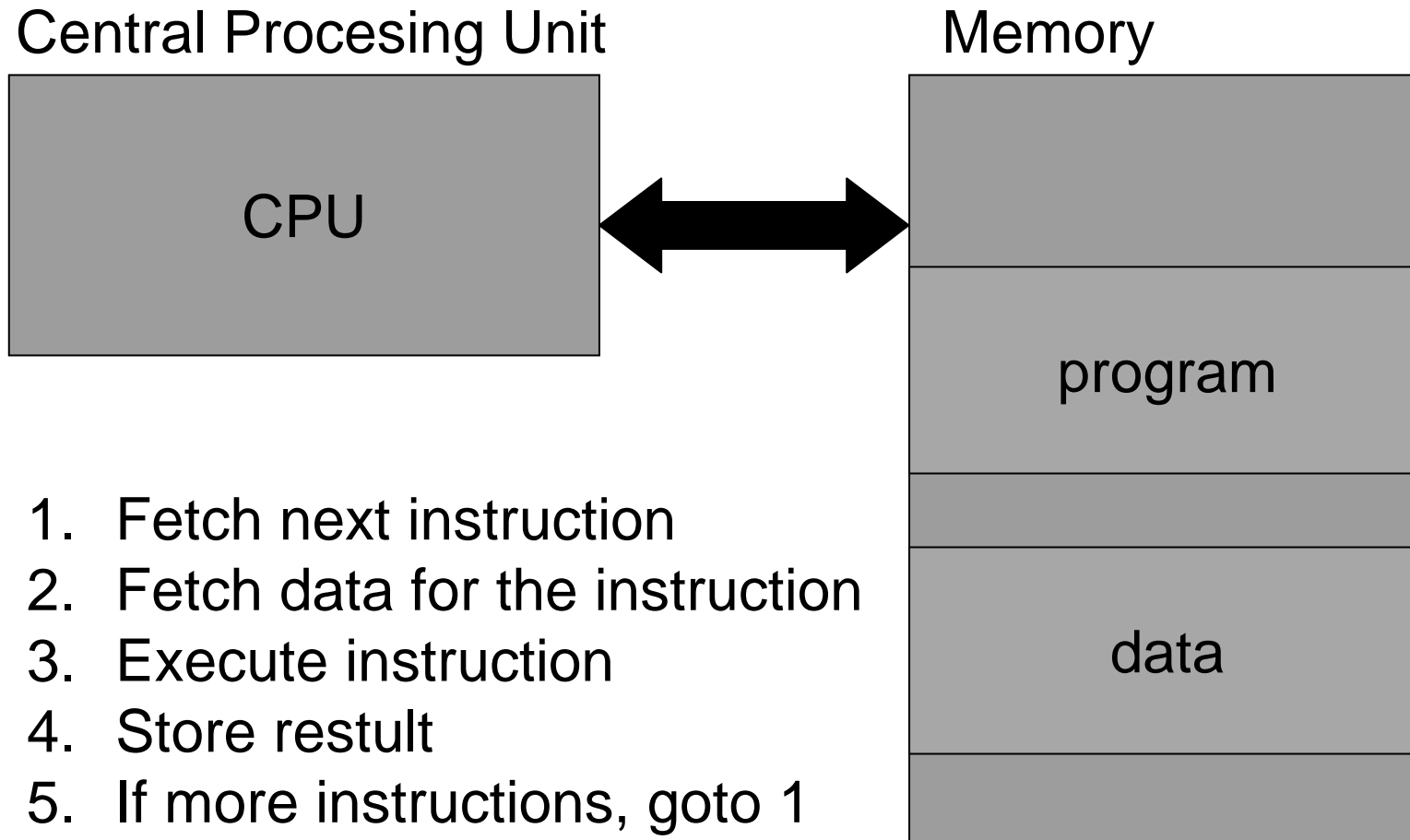
a	15	15	6	6	6	3	
b	6	6	6	3	3		
c		3	3	3	0		

Bench Testing



a	15	15	6	6	6	3	3
b	6	6	6	3	3	3	0
c		3	3	3	0	0	

CPU and Memory



```

import tio.*;
class GCD {
    public static void main (String[] args) {
        int a, b, c;
        System.out.println("Type two natural numbers: ");
        a = Console.in.readInt();
        b = Console.in.readInt();
        while (b != 0) {
            c = a % b;
            a = b;
            b = c;
        }
        System.out.println("The gcd is: " + a);
    }
}

```

use package `tio`

class name

main method

integer variables

prompts for input

reads input,
uses `tio`

main computation

writes output

The Java Programming Language

Java = **imperativeness** (assignments, sequencing, conditionals, and iteration)

+ **object orientation** (classes, methods, inheritance)

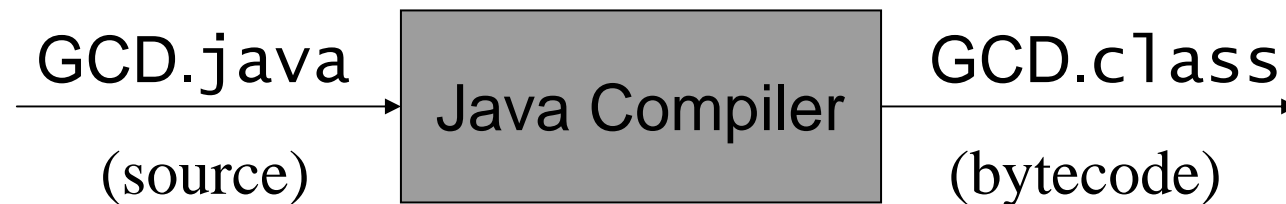
+ easy GUI

+ easy Web (e.g. applets, servlets)

+ (concurrency)

Compiling and Running Code

> javac GCD.java compiles GCD.java



> java GCD executes GCD.class

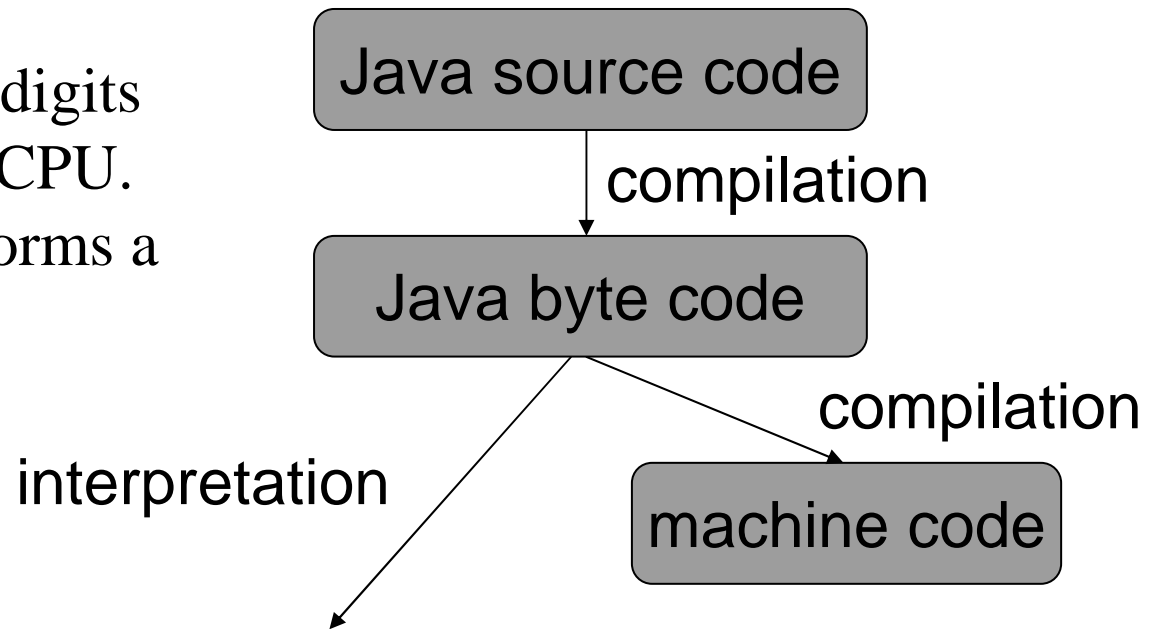
Platform independence

Most programming languages are **compiled** into *machine code*. I.e. there must be a compiler for each *platform* (Macintosh with MacOS, Intel with Microsoft Windows, Sun with Unix, ...).

Java is *platform independent*. Programs are compiled to bytecode that are **interpreted** by a platform specific Java Virtual Machine (JVM). Any bytecode hence runs on any platform (with a JVM).

Interpretation vs. Compilation

Machine code is expressed by binary digits and specific to each CPU. Each command performs a very simple task.



Interpreters, like JVM, are programs that translates each instruction to machine code and executes them, one by one.

How to construct Java programs

There are certain *rules* to follow when writing Java programs.

A program is composed of **lexical elements**:

- *white space*,
- *comments*, and
- *tokens*.

White space and comments are discarded during compilation.

Tokens are groups of characters divided into five types: *keywords*, *identifiers*, *literals*, *operators*, and *separators*.

```
import tio.*;
```

```
class GCD {
```

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

```
        int a, b, c;
```

```
        System.out.println("Type two natural numbers: ");
```

```
        a = Console.in.readInt();
```

```
        b = Console.in.readInt();
```

```
        while (b != 0) {
```

```
            c = a % b;
```

```
            a = b;
```

```
            b = c;
```

```
        }
```

```
        System.out.println("The gcd is: " + a);
```

```
    }
```

```
}
```

identifiers

keywords

separators

literals

operator

White Space

The program below don't use white space to help *readabilty*:

```
import tio.*; class GCD {
static void main (String[] args) {
int a, b, c; System.out.print("Type two natural numbers: \n");
a = Console.in.readInt(); b = Console.in.readInt();
while (b != 0) {c = a%b; a = b; b = c;}
System.out.print("The gcd is: " + a + "\n");}}
```

White spaces are:

- space bar,
- tab character (\t), and
- the newline character (\n).

Comments

```
/* GCD.java  
 * Takes as input two natural numbers and computes their  
 * greatest common divisor  
 * Author: Jens Chr. Godskesen, 26.01.2004  
 */
```

```
import tio.*;
```

```
class GCD {
```

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

```
        int a, b, c;
```

```
        // prompts for input
```

```
        System.out.println("Type two natural numbers: ");
```

Keywords

abstract	assert	boolean	break	byte	case
catch	char	class	const ¹	continue	default
do	double	else	extends	final	finally
float	for	goto	if	implements	import
instanceof	int	interface	long	native	new
package	private	protected	public	return	short
static	strictfp	super	switch	synchronized	this
throw	throws	transient	try	void	volatile
while					

In addition, the words `true`, `false`, and `null` are **reserved words**.

Identifiers

An **identifier** is used to give names to elements in a program.

An identifier is a sequence of Java letters and digits *starting with a letter* (i.e. `tio`, `GCD`, `a1b2c` but not `a+1`, `321`, `3ab`, `3.12`)

- except that keywords and reserved words can't be identifiers.

`$` and `_` are *Java letters* but a space ' ' is not a letter

- `$true` and `_123` are identifiers
- `no space` is not an identifier

Java is case sensitive! I.e. `MyId` is not `myid`.

Literals

Literals are program representations of *values*, e.g.

- 123, -9, and 0 are *integers*
- 1.23 and -0.9 are *floating points*
- 'a' and '1' are *characters*
- "a" and "The gcd is: " are *strings*
- true and false are the *booleans*

Data Types and Variables

A **data type** defines how data is represented in the memory and what *operations* can be performed on the data.

For instance: `int a, b = 0, c;`

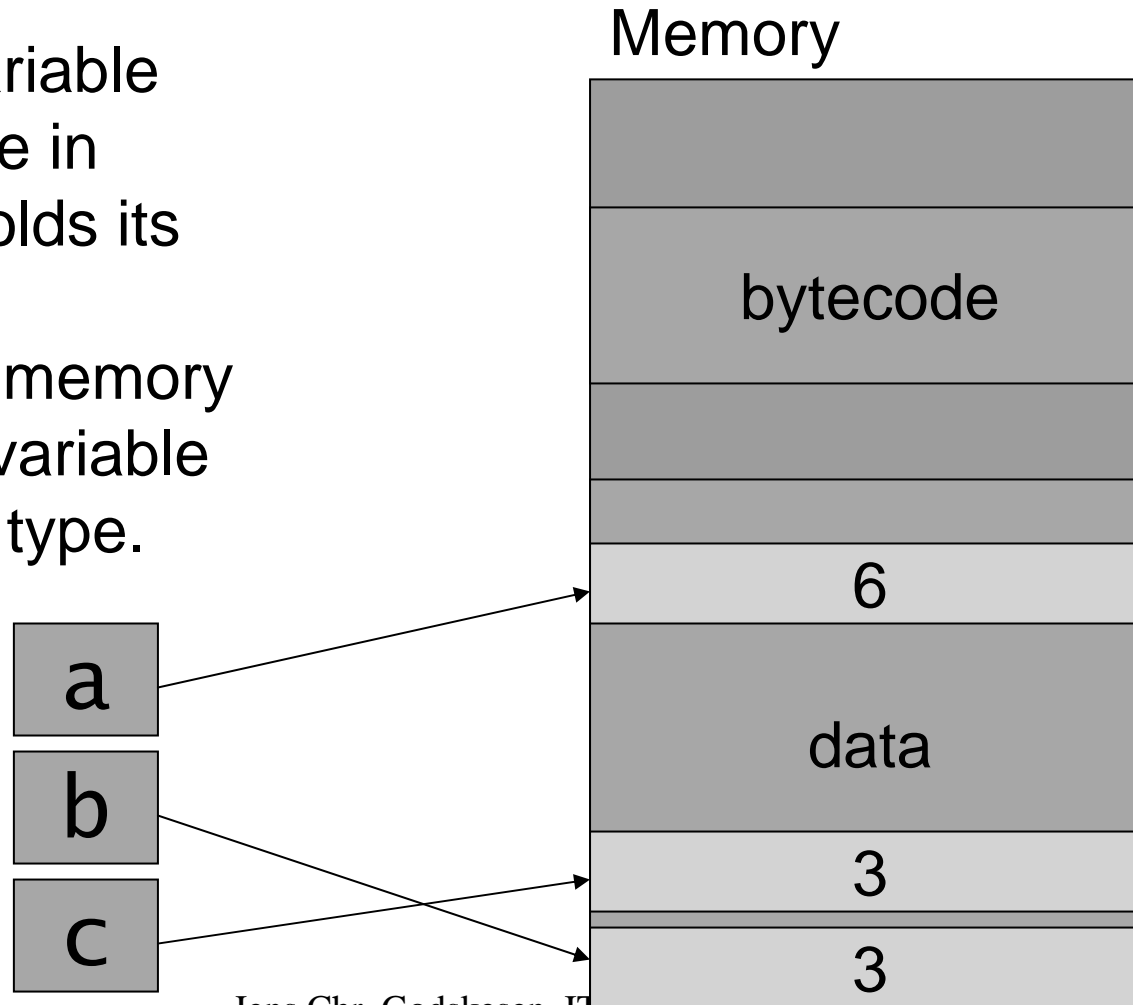
declares the identifiers `a`, `b`, and `c` to be of *integer type* and **initializes** `b` to zero.

An integer occupy 32 bits of memory each, take integer values and allows operations like `+`, `-`, `*`, `\`, ...

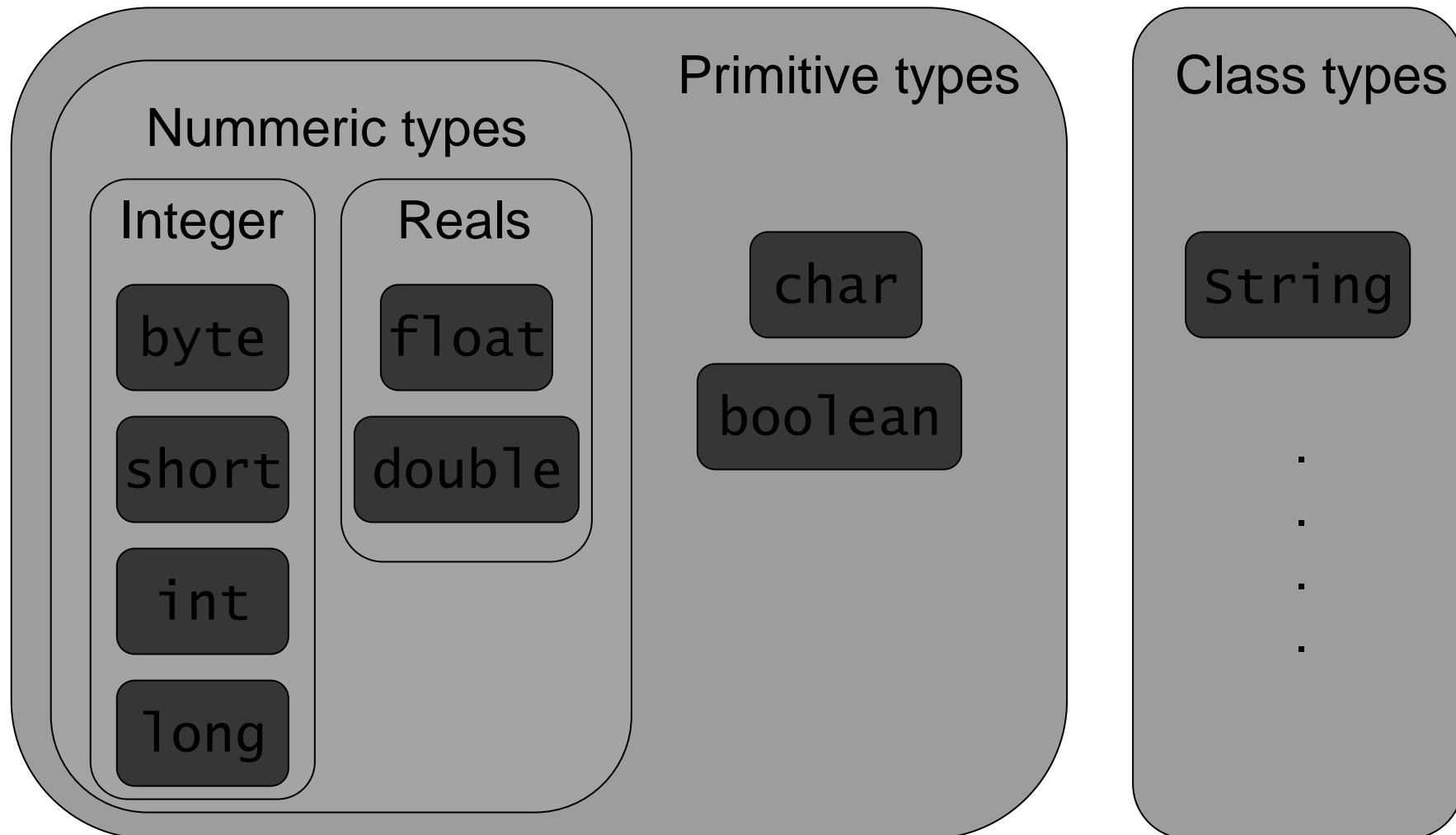
Variables

At runtime a variable refers to a place in memory that holds its **current** value.

The amount of memory occupied by a variable depends on its type.



Data Type Overview



Boolean Types

The type `boolean` consists of the values *true* and *false*, i.e. the values of **boolean expressions** like e.g. in

`b != 0`

Boolean variable declaration and initialization may look like:

– `boolean p = false, q = true;`

The operations are the typical boolean connectives `&&` (and), `||` (or), and `!` (not), hence we may write

– `p && q, p || q, and !p`

Integer Types (int)

Literals like 89, -345, and 0 are of type `int`.

An `int` value (a decimal number) is stored as a **binary number**. E.g. 123 by **1111011**.

$$\begin{aligned}123 &= 1 * 64 + 1 * 32 + 1 * 16 + 1 * 8 + 0 + 2 + 1 \\ &= 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\ &= 1111011_2\end{aligned}$$

Integer Types (byte, short, long)

To save memory `byte` or `short` may be used, and `long` should be used in case of integers outside the range of `int`.

Type	Bits	Range
<code>byte</code>	8	<i>-128 ... 127</i>
<code>short</code>	16	<i>-32.768 ... 32.767</i>
<code>int</code>	32	<i>-2.147.483.648, ..., 2.147.483.647</i>
<code>long</code>	64	<i>$-2^{63} \dots 2^{63} - 1$</i>

Integer Types (byte, short, long)

```
class IntTypes {  
    public static void main (String[] args) {  
        byte b = 127;  
        short s = -32768;  
        int i = 2147483647;  
        long l = 2147483648L; // illegal without L;  
        int j = 023;         // octal number 2*8 + 3  
        int k = 0x23;       // hex number 2*16 + 3  
    }  
}
```

How are negative numbers represented in memory?

If the left most bit designate the *sign* of a number, then e.g. we have that 9 is

00001001

and -9 is

10001001

But then 0 has two representations (+0, -0) and doing arithmetic is inconvenient.

Two's Complement

In (8-bit) *two's complement* a negative number $-k$ is represented by

$$2^8 + (-k)$$

E.g. -128 is $256 - 128 = 128$ i.e. **10000000** and -1 is $256 - 1 = 255$, i.e. **11111111**. Non-negative numbers are represented standardly, i.e.

00000000	00000001	...	01111111	10000000	...	11111111
<i>0</i>	<i>1</i>		<i>127</i>	<i>-128</i>		<i>-1</i>

Floating Point Types

Type	Bits	Decimals	Range
float	32	7	+/-10 ⁻⁴⁵ ... +/-10 ³⁸
double	64	15	+/-10 ⁻³²⁴ ... +/-10 ³⁰⁸

A floating point literal, say `3.14159`, is of type `double`.

```
double d = 3.14159;  
d = 3.14e208; // 3.14*10^208  
float f = 3.14e-45F; // F required
```

The char Type

Type	Bits	Range
char	16	0 ... 65.535

Represents single **characters**. Each character is represented by an integer.

```
char c = 'a'; // corresp. value is 97
c = 'A'; // corresp. value is 65
c = '0'; // corresp. value is 48
c = '+'; // corresp. value is 43
c = '\t'; // horizontal tab, corresp. value is 9
c = '\n'; // newline, corresp. value is 10
int i = (int)c; // retrieving int code
```

Arithmetic Expressions

Arithmetic operators (+, -, *, /, %) can be used on all primitive types, except booleans. Arithmetic is carried out only on `int`, `long`, `float`, and `double`.

```
byte b = 127;
```

```
short s = 32640;
```

```
s = b + s;    // illegal !!!
```

```
int i = b + s; // b and s converted to int
```

Operands must match on the *largest* of their types:

```
int < long < float < double
```

```
long l = 2147483648L;
```

```
double d = l + i; // i converted to long
```

```
d = d + i;    // i converted to double
```

Overflow, Non Numbers, Exceptions

```
class Arithmetic {  
    public static void main (String[] args) {  
        int i = 2/3;                // int's can't be fractions  
        double d = (double)2/3;    // but double's can  
  
        System.out.println("d/d: " + d/d);    // 1.0  
        System.out.println(2147483647+1);    // int overflow  
        System.out.println("d/i: " + d/i);    // Infinity  
        System.out.println("0.0/i: " + 0.0/i); // NaN  
        System.out.println("i/i: " + i/i);    // Arithmetic exception  
    }  
}
```

`int` division by 0 raises **exception**. Floating points never raises exceptions, but special values, **Infinity** and **NaN** results.

Type Conversion (I)

Widening is to convert one (primitive numeric) type to another containing at least as many bits, e.g.

```
int i = 2147483647;  
long l = 2147483648L;  
double d = l + i; // l + i converted to double
```

Widening to floating point may (rarely) cause *information loss* (due to floats binary representation) as e.g.

```
float f = (float)1234567890; // widening an int  
System.out.println(f); // 1.23456794E9
```

Type Conversion (II)

Type narrowing (or type cast) is to convert one (primitive numeric) type to another containing fewer bits, e.g.

```
double d = 3.14159;  
int i = (int)d; // i = 3
```

```
short s = 128; // 0000000010000000  
byte b = (byte)s; // b is 10000000, i.e. -128
```

Assignments operators

`a = 7`

is an **assignment expressions** (`var = expr`),

`a = 7;`

is an **assignment statements** (`var = expr;`).

The sequence of assignment statements:

`a = 7; b = 9; c = a + b;`

is equivalent to the statement: `c = (a = 7) + (b = 9);`

Multiple assignments, like `a = b = c = 1;` should be read as `a = (b = (c = 1));` because `=` is right *associative*.

Assignments operators

Other assignment operators exist, e.g. `+=` and `-=`
(and similar for other binary operators)

`a += 7` is equivalent to `a = a + 7`

`b -= 9` is equivalent to `b = b - 9`

The unary **increment** (`++`) and **decrement** (`--`):

- `b = a++` is equivalent to `a = (b = a) + 1`
- `b = ++a` is equivalent to `b = (a = (a + 1))`
- (likewise for unary decrement)

Precedence and Associativity

Operators are associated with a **precedence** and **associates** to either right or left.

The assignment $b = a = a + 1;$ is equivalent to
 $b = (a = (a + 1));$

(but not to $b = (a = a) + 1;$) because $+$ has higher **operator precedence** than $=$.

The operators $*$ and $/$ have equal precedence higher than $+$ so, $18/6*3+2$ is $(18/6*3)+2$ and because $*$ and $/$ **associates** from left to right we get $((18/6)*3)+2$.