Introduction to Java programming

Lecture 6: Rehearsal / Révisions
(pour la pale machine du 4 Juin 2008)

Frank Nielsen
LIX
nielsen@lix.polytechnique.fr

Monday, 2nd June 2008

Feel free to ask questions!!!

Philippe Chassignet
Salle 31
Groupes 1 & 7
philippe.chassignet@polytechnique.edu

Olivier Serre
Salle 34
Groupes 4 & 10
serre@liafa.jussieu.fr

David Monniaux
Salle 32
Groupes 2 & 8
david.monniaux@ens.fr

Stephane Redon
Salle 35
Groupes 5 & 11
stephane.redon@inria.fr

Etienne Duris
Salle 33
Groupes 3 & 9
etienne.duris@univ-paris-est.fr

Yann Hendel
Salle 36
Groupes 6 & 12
hendel@lix.polytechnique.fr

Videos of lectures are online

http://www.catalogue.polytechnique.fr/cours.php?id=2385

Cannot watch these videos?
Contact DSI
http://www.dsi.polytechnique.fr/
or
Coard Jean-Paul (M.)
<Jean-Paul.Coard@Polytechnique.edu>

Computer Science at Ecole Polytechnique

http://www.enseignement.polytechnique.fr/informatique/

Année	Septembre	Octobre	Novembre	Décembre	Janvier	Février	Mars	Avril	Mai	Juin	Juillet	Août
1	Incorporation								INF 311: Introduction à l'informatique INF 321: Principes des langages de programmation			
2	INF 421: Programmation et Algorithmique		INF 421: Programmation et Algorithmique		INF 431: Algorithmes et Programmation: du séquentiel au distribué							
									INF444: Modex		INF444: Modex	
3		Programmes d'approfondissement (Master 1)							Stage de recherche (Master 1)			
4	Année professionalisante (Master 2)											

INF311 half-road!!!

Filiere Image
MPRI, etc.

Theses (Ph. D)

Agenda

Lecture 6: Rehearsal
(pour la pale machine,Mercredi 4 Juin)

TD5: This afternoon

TD6: Pale machine, **Mercredi 4 Juin**

Lectures 7, 8,9 ,10:

Java Programming

+

Basic Algorithms/Data-Structures

Bien preparer la pale machine

- Lire le polycopie (4 chapitres): pages 11-57
- Finir les TDs (salle machine)
- S'entrainer avec les annales
- Bien comprendre et maitriser:
  - variables, affectation
  - typage et regle de coercion
  - fonctions (statiques)
  - passage par valeur
  - tableaux et passage par reference
  - pile d'execution (et appels recursifs)
  - chaines de caracteres & String (compareTo)

Bien preparer la pale machine

Les bases de la programmation et de l'algorithmique, année 2008 (promotion 2007)

Travaux dirigés, niveau débutant

Enseignants :
Cours : Frank Nielsen
groupes 1 et 7 : Stéphane Redon - Luca de Féo - Maria Naya Plasencia
groupes 2 et 8 : Etienne Duris - Guillaume Chapuy - David Savourey
groupes 3 et 9 : Olivier Serre - Sylvain Pradalier - Vincent Jost
groupes 4 et 10 : Yann Hendel - Marc Kaplan - Gaetan Laurent
groupes 5 et 11 : David Monniaux - Andrea Roeck - Bogdan Cautis
groupes 6 et 12 : Philippe Charignet - Giacomo Nannicini - Andrey Ivanov

Le poly est disponible en pdf

Exercices d'entraînement à la composition sur machine NEW!

Annales des compositions

Documentation :

- Quelques problèmes fréquents et leurs solutions
- Un mémento de Java
- Conventions d'écriture d'un programme Java NEW!
- Les classes de Java en local et chez Sun
- La classe TC
- Introduction à l'environnement Unix
- Les moyens matériels

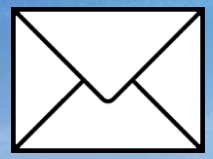
Certificat
Pour installer le certificat des serveurs de l'enseignement, cliquer ici. Ce certificat est utilisé ici dans le processus d'authentification des dépôts. On notera qu'il est déjà installé dans le navigateur Mozilla des salles de TD. Il faudra penser à l'installer si on veut déposer, lire son courrier, changer des mots de passe, depuis d'autres machines.

- TD 1 - Les outils indispensables et un corrigé
- TD 2 - Programmation en Java - déposer au moins jusqu'à l'exercice 5 avant le jeudi 22/5 au soir - et un corrigé.
- TD 3 - Tableaux et chaînes de caractères

URL page TD:
http://www.dix.polytechnique.fr/INF311/TD_08/

https://www.enseignement.polytechnique.fr/informatique/INF311/TD_08/INF311-entrainement-1.php

Answering Questions



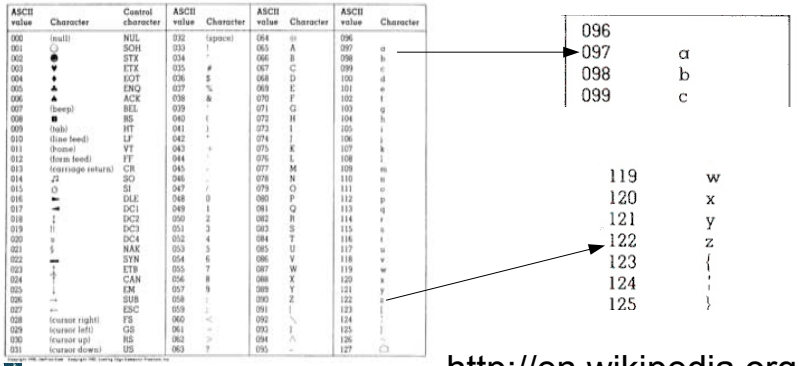
Frank Nielsen

nielsen@lix.polytechnique.fr

Lexicographic order... characters

American Standard Code for Information Interchange (ASCII)

- Distance between two characters:
  =span in ASCII code table
- Java *distinguishes* lower/upper cases: A is not a
- Java codes char using two bytes (UNICODE)



096
097 a
098 b
099 c

119 w
120 x
121 y
122 z
123 {
124 |
125 }

http://en.wikipedia.org/wiki/ASCII

Lexicographic order... characters

```
char c1,c2;

c1='a';
c2='z';
// Compare character code
if (c1<c2)
   {System.out.println(c1+" is before "+c2);}
   else
   {System.out.println(c1+" is after or equal to "+c2);}

int codec1=c1; // type casting/conversion
int codec2=c2; // type casting conversion

System.out.println("Code ASCII for "+c1+":"+codec1);
System.out.println("Code ASCII for "+c2+":"+codec2);
```

```
a is before z
Code ASCII for a:97
Code ASCII for z:122
```


String method **compareTo()**:
u.compareTo(v) compares **lexicographically** the strings u with v.

```
String u="Polycopie", v="Polytechnique";
System.out.println(u.compareTo(v));
// => -17
```

Polycopie
Polytechnique

c:99
t:116
-17

Differ at fifth position: return 'c'-'t'=-17
(using ASCII code value)

```
System.out.println("c:"+(int)'c');
System.out.println("t:"+(int)'t');
int diff='c'-'t';
System.out.println(diff);
```


Lexicographic order... characters

String method **compareTo()**:
u.compareTo(v) compares **lexicographically** the strings u with v.

In case there is no place characters differ, then
- 0 if strings are perfectly identical
- Length(u) – Length(v) otherwise (substring)

```
String a="champagne",b="champ";
System.out.println(a.compareTo(b));
System.out.println(a.length()-b.length());
```

champagne
champ

9-5=4

Lexicographic order... strings

```
// Static function for comparing two strings

static int LexicographicOrder(String p, String q)
{
int i=0;

while(i<p.length() && i<q.length())
   {
   if (p.charAt(i)==q.charAt(i))
              i++;
              else
                 return p.charAt(i)-q.charAt(i);
   }

return p.length()-q.length();
}
...
String p="Papillon", q="Papier", r="Papillonner";
System.out.println(LexicographicOrder(p,q)); //7
System.out.println(LexicographicOrder(p,r)); //-3
```


Converting lower to upper cases

```
//
// Convert to upper case
//
static String LowerToUpper(String s)
{
String result="";
char c;

for(int i=0;i<s.length();i++)
   {
   c=(char)(s.charAt(i)-32); //32=2^5
   result+=c; //concatenation, append c to result
   }

return result;
}
...
String s=LowerToUpper("convert a simple sentence");
```


Testing equality of characters

```
class TestEquality
{
// In Java, characters are stored using two bytes
//(for UNICODE, 65K characters)
public static void Identical(char c1, char c2)
{

System.out.println("Integer code for char "+c1+":"+(int)c1);
System.out.println("Integer code for char "+c2+":"+(int)c2);

if (c2==c1) System.out.println("Characters are identical:"+c1+"="+c2);
else
System.out.println("Characters are different:"+c1+"<>"+c2);
}

public static void main(String[] args)
{
char c1,c2;

Identical('a','A');
Identical('a','a');
Identical('a','b');

}
```

```
Integer code for char a:97
Integer code for char A:65
Characters are different:a<>A
Integer code for char a:97
Integer code for char a:97
Characters are identical:a=a
Integer code for char a:97
Integer code for char b:98
Characters are different:a<>b
```


Strings: Testing physical equality

```
public static void PhysicalIdentical(String c1, String c2)
{
if (c2.compareTo(c1)==0)
   {  System.out.println("Strings are identical:"+c1+"="+c2)
      }
   else
   {
   System.out.println("Strings are different:"+c1+"<>"+c2);
   }
}

 String s1="Coucou",s2="Coucou", s3=s1, s4="Salut";

 PhysicalIdentical(s1,s2);
 PhysicalIdentical(s3,s1);
 PhysicalIdentical(s1,s4);
 ...
```

```
Strings are identical:Coucou=Coucou
Strings are identical:Coucou=Coucou
Strings are different:Coucou<>Salut
```

Physical = Are the contents the same?
Logical (references) equality => Physical equality

Testing equality

Physical = Are the contents the same?

```
public static void Message(boolean b)
{
if (b) System.out.println("Equal");
   else System.out.println("Different");
}

public static boolean PhysicalIdentical(int [] tab1, int [] tab2)
{
if (tab1.length!=tab2.length)
   return false;
   else
   {
      for(int i=0;i<tab1.length;i++)
         if (tab1[i]!=tab2[i])
            return false;
   }
   return true;
}
```

```
int [] t1={1,2,3};
int [] t2={2,3,4};
int [] t3={1,2,3};
int [] t4=t1;

Message(PhysicalIdentical(t1,t4));
Message(PhysicalIdentical(t1,t3));
Message(PhysicalIdentical(t1,t2));
```

Equal
Equal
Different

Testing equality

Logical = Are the references the same?

```
public static boolean LogicalIdentical(int [] tab1, int [] tab2)
{
System.out.println("References:"+tab1+" "+tab2);

if (tab1!=tab2)
   return false;
   else
   return true;
}
```

```
int [] t1={1,2,3};
int [] t2={2,3,4};
int [] t3={1,2,3};
int [] t4=t1;

Message(LogicalIdentical(t1,t4));
Message(LogicalIdentical(t1,t3));
Message(LogicalIdentical(t1,t2));
```

```
References:[I@3e25a5 [I@3e25a5
Equal
References:[I@3e25a5 [I@19821f
Different
References:[I@3e25a5 [I@addbf1
Different
```

Caution!
You can be logically different but physically identical:
See test (t1,t3)

Strings...: if you want to know more

Use methods/constructors(11) described in the documentation (javadoc)
Tutorial at http://java.sun.com/docs/books/tutorial/java/data/strings.html

Constructor Summary

`String()`
Initializes a newly created `String` object so that it represents an empty character sequence.

`String(byte[] bytes)`
Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.

`String(byte[] ascii, int hibyte)`
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

`String(byte[] bytes, int offset, int length)`
Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.

`String(byte[] ascii, int hibyte, int offset, int count)`
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

`String(byte[] bytes, int offset, int length, String charsetName)`
Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.

`String(byte[] bytes, String charsetName)`
Constructs a new `String` by decoding the specified array of bytes using the specified charset.

`String(char[] value)`
Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.

`String(char[] value, int offset, int count)`
Allocates a new `String` that contains characters from a subarray of the character array argument.

`String(String original)`
Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

`String(StringBuffer buffer)`
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

Eleven constructors!!!!

Managing/reporting errors

```
class TestException
{

public static double [] AddVector(double [] v1, double [] v2)
{
double [] result=new double[v1.length];

for(int i=0;i<v1.length;i++)
    result[i]=v1[i]+v2[i];

return result;
}

public static void main(String [] args)
{
double [] x={1.0, 2/3.0};
double [] y={0.5, 0.2};
double [] z={0.0, 1.0, 2.0};

double [] a=AddVector(x,y);
double [] b=AddVector(x,z);

}
```

Place sentinels to avoid program misuses.

Magic formula:

```
throw new RuntimeException("message")
```

Magic formula:
`throw new RuntimeException("message")`

```
class TestException
{

public static double [] AddVector(double [] v1, double [] v2)
{
double [] result=new double[v1.length];

if (v1.length!=v2.length)
       throw new RuntimeException("Vectors do not have same dimension!");

for(int i=0;i<v1.length;i++)
     result[i]=v1[i]+v2[i];

return result;
}

public static void main(String [] args)
{
double [] x={1.0, 2/3.0};
double [] y={0.5, 0.2};
double [] z={0.0, 1.0, 2.0};

double [] a=AddVector(x,y);
double [] b=AddVector(x,z);

}
```

```
Exception in thread "main" java.lang.RuntimeException: Vectors do not have same dimension!
        at TestException.AddVector(TestException.java:9)
        at TestException.main(TestException.java:24)
```


Writing safe programs is hard

(programs that do not crash)

```
public static double [] AddVector(double [] v1, double [] v2)
{
double [] result=new double[v1.length];

if (v1==null || v2==null || v1.length!=v2.length)
       throw new RuntimeException("Vectors do not have same dimension!");

for(int i=0;i<v1.length;i++)
     result[i]=v1[i]+v2[i];

return result;
}

 ...
 double [] w=null;
 double [] c=AddVector(x,w);
 ...
```

```
Exception in thread "main" java.lang.RuntimeException: Vectors do not have same dimension!
        at TestException.AddVector(TestException.java:9)
        at TestException.main(TestException.java:24)
```

How do you know that this function is always safe?

=> In practice: You need to prove it (eg., static analysis)

Functions can have objects as arguments... ... and also return an object as a result:

Static function returns an object ObjRe

```
public static ObjRes Function(Obj1 o1, ..., ObjN oN)
{
ObjRes result=new ObjRes();
...
return result;
}
```

Objects given as parameters


```
class Obj1{} // Default constructor
class Obj2{}
class Obj3{}
class Obj4{}
class Obj5{}
class ObjRes{}


class TestObject
{

public static ObjRes F(Obj1 o1, Obj2 o2, Obj3 o3, Obj4 obj4, Obj5 obj5)
   {
   ObjRes res=new ObjRes();
   return res;
   }

public static void main(String [] args)
   {
   Obj1 obj1=null;
   Obj2 obj2=null;
   Obj3 obj3=null;
   Obj4 obj4=null;
   Obj5 obj5=null;
   ObjRes res;

   res=F(obj1, obj2,obj3, obj4, obj5);
   }
}
```

Question on:
Static functions and non-static methods on objects

- Static functions of a class (INF311)
- Non-static functions of a class are methods (OO)

```
class Date
{int dd;int mm;int yyyy;
public static final String[ ] months={
    "January", "February", "March", "April", "May","June", "July", "August", "September", "October",
      "November", "December" };

// Static function
public static void Display(Date day)
{System.out.println(day.dd+" "+months[day.mm-1]+" "+day.yyyy);   }

// Method for the object date
void Display()
{System.out.println(this.dd+" "+months[this.mm-1]+" "+this.yyyy);   }

// Constructor
public Date(int day, int month, int year)
{
this.dd=day;
this.mm=month;
this.yyyy=year;
}
}
```



```
…
// Static function
public static void Display(Date day)
{System.out.println(day.dd+" "+months[day.mm-1]+" "+day.yyyy);   }


// Method for the object date
void Display()
{System.out.println(dd+" "+months[mm-1]+" "+yyyy);   }

…

public static void main(String[] args)
{
Date day1=new Date(23,12,1971);
Date day2=day1; // beware not copying here. Just memory reference
Date day3=new Date(23,12,1971);

System.out.println(isEqual(day1,day3));

System.out.println(day1);
System.out.println(day2);
System.out.println(day3);

// call static function Display (give object as argument)
Date.Display(day1);
// call the method on the object (no args)
day1.Display();
}
```


Question on:
Objects, reserved keywords, and type

Declaring a class Object => new type Object created
But Object does not become a reserved keyword.

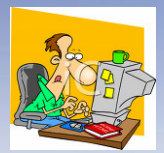
```
class TestD
{
    public static void main(String[] args)
    {
        System.out.println("Type de la classe");
        Date d=new Date(27,05,2008);
        int Date=3;
        d.Display();
     System.out.println("La valeur entiere de la var Date est:"+Date);

        // Date.Display(d); invalid!!!
        // Date is now considered as a variable of type int

    }
}
```

Compiler and semantic of source codes
Convention: var begins with lower case
class begins with an upper case

Today...

Lecture 6:

Rehearsal



Finir les TDs
S'entrainer (annales)

Lire le polycopie

I.1. Variable: Declaration

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

- A variable is *uniquely* named (not a reserved keyword)
- A variable *stores* a value in a memory slot (reference)
- A variable has a *type*

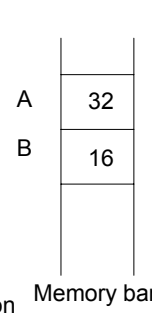
```
int A;
int B;
```

Variable declaration

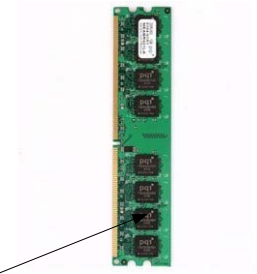
A=32;

B=16;

Check types at compilation



Memory bank



I.2. Variable: Assignment

```
var=value;
```

Stores `value` at memory location *referenced by* `var`

Thus the semantic of a variable in assignment is:
- Left hand side: (memory) reference
- Right hand side: value

I.2. Variable: Variable assignment

```
varQ=varP;
```

- Get the value at memory location referenced by `varP`
- Store that value at memory location referenced by `varQ`

Left hand side is STORE.. ....Insert
Right hand side is GET..... ....Retrieve

```
STORE(varQ)=GET(varP);
```


I.3. Variable: Incrementing

```
var=var+constant;
```

- Get the value at memory location @var referenced by `var`
- Increment that value by the `constant`
- Store the incremented result at memory @var

`var+constant` is a well-formed expression

```
var+=constant;
```

I.4. Variable: Pre-/post-Incrementing

```
var=var+1;
```

```
++var;   var++;
```

Pre-incrementation:
increment var by one
return the value of var

Post-incrementation:
return the value of var
increment var by one

++/-- are **unary operators** (in expressions)

I.5. Variable: Expression Assignment

```
var=Expression;
```

```
var=16;
var=14*3/5-2*3;
```

- Evaluate `Expression`, and then
- Store the value at memory location referenced by `var`

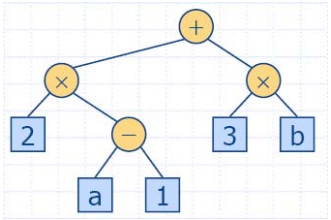
```
var=Expr1 op Expr 2;
```

```
var=varP*varQ-3;
var=varP%3-5*2;
```


I.5. Expressions: Priority rules

```
var=ComplexExpression;
```

```
M=325%27; // Integer division (modulo)
delta=b*b-4*a*c;
```



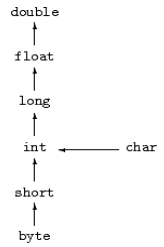
internal nodes: operators
external nodes: operands

arithmetic expression tree:
2*(a-1)+3*b

(2 × (a −1) + (3 × b))

Priority rules of operators: Disambiguate lack of parenthesis

I.5. Common errors

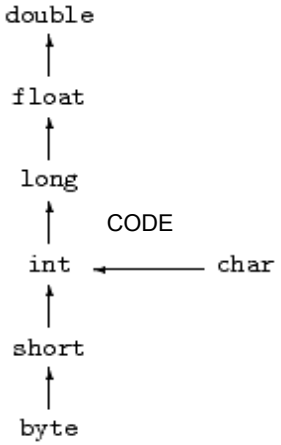


- Incrementation
- Integers (`int long`) and reals (`float double`)
- Type checking and implicit casting

```
int i=0;
double i;// ERROR: already defined
double I;// OK: lower case different from upper case

i=i++; // i is 0, ERROR? Mistyping?
// meant i++;

int p=2;
int q=3;
// Take care p/q is 0 (integer division)
if (p/q<0.5) System.out.println("2p<q");
     else System.out.println("2p>q");
```

I.6. Explicit/implicit casting

```
// Explicit casting
double p=2.3;
int ptrunc=(int)p; // loss of precision ptrunc=2

// Implicit casting
char c='T';
int codec=c; // ASCII code of c
System.out.println(c+" "+codec);
// we get T 84
```



Example: byte+char=int

```
byte b=3;
double res=b+c;
System.out.println(res); // 87.0
```


I.7. Java operators....Expressions

Unary, binary, ternary
Operators
Operands

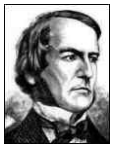
```
int i=4;
int j=3;

int res=++i*j++%--i; //3

res=--i-++j*(i<4?3:4);
System.out.println(res);// -12
res=--i-++j*i<4?3:4;// 3
System.out.println(res);
}
```

Java has many operators...

Operators	Precedence
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

II.1. Boolean expressions

Bool algebra is 0/1 (true/false) algebra



George Boole (1815 - 1864)

+ OR Gate ||

TRUE + TRUE = TRUE
TRUE + FALSE = TRUE
FALSE + TRUE = TRUE
FALSE + FALSE = FALSE

* AND Gate &&

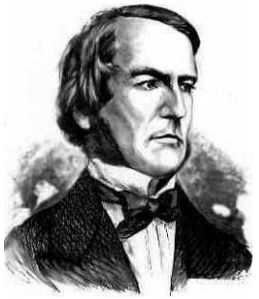
TRUE * TRUE = TRUE
TRUE * FALSE = FALSE
FALSE * TRUE = FALSE
FALSE * FALSE = FALSE

- XOR...

De Morgan's laws:
NOT(A + B) = (NOT(A) * NOT(B))
NOT(A * B) = (NOT(A) + (NOT(B))

In Java, use connectors || and && for creating boolean expressions

II.1. Boolean expressions



Boole, George
The mathematical analysis of logic, 1847,
A Treatise on the Calculus of Finite Differences, 1860

```
boolean a=true, b=false;
boolean or,and;
boolean expression;

or=a||b;
System.out.println(or);
and=a&&b;
System.out.println(and);

expression=(a||b)&&a||!a;

System.out.println(expression);//true
```

Priority order: var=true||Expression;
=> Expression is not evaluated

II.1. Conditional structure if

```
if (BooleanExpression)
        {BlockA}
     else
        {BlockB}
```

```
if (!BooleanExpression)
        {BlockB}
     else
        {BlockA}
```

Getting the minimum of a and b:

```
if (a<b)
        c=a; // Block1
     else
        c=b; // Block2
```


II.2. Conditional structure: nested if

```
if (BooleanExpression1)
        {BlockA}
     else
        if (BooleanExpression2)
           {BlockB}
           else
              if BooleanExpression2)
                    {BlockC}
                    else
                    {BlockD}
```


II.3. Conditional structure: switch

```
switch(n)
     {
     case 0: InstructionSequence0;
             break;
     case 1: InstructionSequence1;
             break;
     case 2: InstructionSequence2;
             break;
     case 3: InstructionSequence3;
              break;
     default: InstructionDefault;
             break;
```


II.3. Conditional structure: switch

```
class SwitchTest
{
public static void main(String[] args)
{char c='a';int code;

switch(c)
{
case 'a': case 'A':
       code=1;
       break;

case 'b': case 'B':
       code=2;
       break;

default:
       code=0;
       break;
}

System.out.println("Code="+code);
}
}
```

The Fundamentals.

II.4. While loop

```
while (boolean_expression)
  { block_instruction;}
```

- Evaluate boolean_expression
- Execute the block of instruction if and only if it is true

```
while (boolean_expression)
  Single_instruction;
```

Forever running program...

```
while (true);
```


The Fundamentals.

II.4. While loop

```
int i=0, res=0;

while(i<10)
   {
      res=res+i*i;
      i++;
   }
//res=285 i=10
```

Unrolling the while loop

```
int i=0, res=0;
res=res+i*i;//res=0
i++;//i=1 (i<10) is true
res=res+i*i;// res=1
i++;//i=2 (i<10) is true
res=res+i*i;// res=5
i++;//i=3 (i<10) is true
res=res+i*i;
i++;//i=4 (i<10) is true
res=res+i*i;
i++;//i=5 (i<10) is true
res=res+i*i;
i++;//i=6 (i<10) is true
res=res+i*i;
i++;//i=7 (i<10) is true
res=res+i*i;
i++;//i=8 (i<10) is true
res=res+i*i;
i++;//i=9 (i<10) is true
res=res+i*i;// res=285
i++;//i=10  (i<10) is false
```


The Fundamentals.

II.5. For loop: Convenient for iterating

```
for(instruction1; boolean_condition; instruction2)
              block_instructions;
```

Equivalence with While construction

```
instruction1;
while (boolean_condition)
  {
       block_instructions;
       instruction2;
  }
```


The Fundamentals.

II.5. For loop: Convenient for iterating

```
int i, n=10;
  int cumulLoop=0;

  for(i=0;i<n;i++)
       {cumulLoop+=i;}
```

```
int cumulLoop=0;
i=0; // Initialization
cumulLoop+=i;
i++; // i=1 now
// i<n so we continue...
cumulLoop+=i;
i++; // i=2 now
// i<n so we continue...
cumulLoop+=i;
...etc...
cumulLoop+=i; // i=n-1
i++; // i=n now
// i is not i<n so we stop...
```

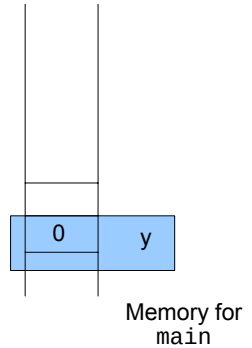
III.1. Defining a static function

```
public static  typeF F(type1 arg1, ..., typeN argN)
            {
            // Description
            Block of instructions;
            }
```

- `typeF` is the type of the return value
- `type1... typeN` are the types of arguments
- Java passes arguments through value
- If `typeF` is `void` then it is a procedure

III.2. Function calls and memory stack

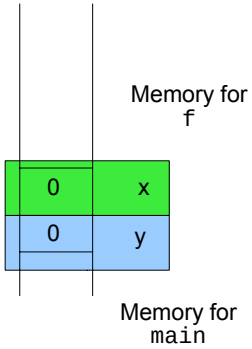
```
class FunctionCallSample
{

public static void f(double x)
{
x=1;
System.out.println(x);//1
return;
}

public static void main(String[] args)
{
int y=0;

f(y);
System.out.println(y);//0

}
}
```



Java is pass by value (function arguments)

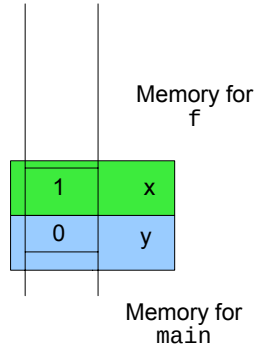
Pass by value y (=0) to variable x in function f

```
class FunctionCallSample
{

public static void f(double x)
{
x=1;
System.out.println(x);//1
return;
}

public static void main(String[] args)
{
int y=0;

f(y);
System.out.println(y);//0

}

}
```



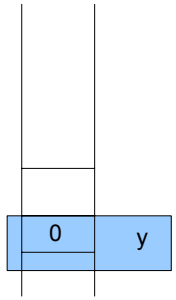
Assign in function f variable x to 1

```
class FunctionCallSample
{

public static void f(double x)
{
x=1;
System.out.println(x);//1
return;
}

public static void main(String[] args)
{
int y=0;

f(y);
System.out.println(y);//0

}

}
```



Return from function. Release memory in stack

```
class FunctionCallSample
{

public static void f(double x)
{
x=1;
System.out.println(x);//1
return;
}

public static void main(String[] args)
{
int y=0;

f(y);
System.out.println(y);//0

}

}
```



Memory for main

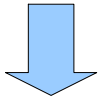
The Fundamentals.

III.3. Function and efficiency

Avoid to call many times a function with the **same** arguments

=> use **temporary** variables to store intermediate results

```
// for all j
g(f(t)[i]).[j]
```



```
// much better to do as
String [] tmp=f(t)[i];// computed once
g(tmp).[j]
```


The Fundamentals.

IV.1. Arrays

- Arrays of elements of type `ELEMENT_TYPE` are of type `ELEMENT_TYPE []`
- Declare array variables as `ELEMENT_TYPE [] MyTab;`
- Allocate memory for arrays with keyword `new`
- Size in `new` is an expression that is evaluated

```
boolean [ ] prime = new boolean[16];
double [] DblArray; DblArray=new double[3*n+1]
int [ ] prime={2, 3, 5, 7, 11, 13, 17, 19};
```


The Fundamentals.

IV.2. Arrays: Size

```
prime.length;
System.out.println(prime.length);
```

Beware: for Strings `s`, use method `length()`

```
s.length();
```

Strings are not arrays of characters!!!

Index of arrays *begin at 0... to length-1*:

```
for(int i=0;i<tab.length;++i)
....
for(int=tab.length-1;i>=0;--i)
...
```

IV.2. Arrays: Size and lazy evaluation

In loops, check whether the index goes out of bound first
Otherwise, you'll get an exception: `ArrayIndexOutOfBounds`

```
int n=tab.length;
while (tab[i]=='#' && i<n)
  {
  ...
  i++;
  }
```

If i=n, we do not check whether tab[n]='#' or not. Thus, we avoid the exception `ArrayIndexOutOfBounds`

```
int n=tab.length;
while (i<n && tab[i]=='#')
  {
  ...
  i++;
  }
```

Lazy evaluation of boolean expression
A && B
If A is false, do not evaluate B

IV.3. Arrays: Pass by reference

A variable that has a type array is a **reference** to the array
(the memory address of the first element)

Therefore an argument of type array **does not copy** all array elements in the memory allocated for the function, but rather allocate a **memory reference**:

```
static void MyFunction(int [ ] x)
MyFunction(v);
// the contents of v may thus be changed
// by some instructions
```

Only the *reference of v* is copied to the memory allocated for the function MyFunction.

IV.3. Arrays: Pass by reference

```
class ArrayRef{
  // Increment all elements by one
  public static void MyFunction(int [] tab)
  {
    for(int i=0;i<tab.length;i++)
      tab[i]++;
  }

  public static void main(String [] args)
  {
  int x [] ={0,1,2};
  MyFunction(x);
  System.out.println(x[0]+" "+x[1]+" "+x[2]);
  }
}
```

1 2 3

Memory allocation of arrays is in the heap, not stored in the memory stack of function calls

IV.3. Arrays: Pass by reference

```
class ArrayRef2{
  // Increment all elements by one
  public static void MyFunction(int [] tab)
  {
    int l=tab.length;
    tab=new int[l]; // Attn.: WRONG TO DO SO!!!!
    for(int i=0;i<l;i++) tab[i]=0;
  }

  public static void main(String [] args)
  {
  int x [] ={0,1,2};
  MyFunction(x);
  System.out.println(x[0]+" "+x[1]+" "+x[2]);
  }
}
```

0 1 2

Reference of x does not change after function call MyFunction

The Fundamentals.

IV.4. Arrays and linear search

Usual problem: Search if an element is already inside an array:

- Return the *index* of the position, if search is positive, or
- Return -1 if element is not found.

Beware: **Do not use == for comparing cell elements**.
Use instead a function (method), say `compareTo`

== of basic types (int, double) test for (physical) equality
== of array/object types test for equality of the references only
(not the contents)


```
class Point
{double x,y;
Point(double xx, double yy) {this.x=xx; this.y=yy;}
boolean isEqual(Point q) {return (this.x==q.x && this.y==q.y);}
}

 class PointSearch
 {
    public static int Inside(Point [] t, Point q)
    {for(int i=0;i<t.length;i++)
       {if (q.isEqual(t[i])) return i;  }   // do not use == here
    return -1;}

    public static void main(String[] tabofargs)
    {
    Point [] tab=new Point[10];

    for(int i=0;i<tab.length;i++)
       tab[i]=new Point(i,i*i);

    Point query1=new Point(2,4);
    System.out.println(Inside(tab,query1)); //2
    Point query2=new Point(3,8);
    System.out.println(Inside(tab,query2)); //-1
    }
 }
```