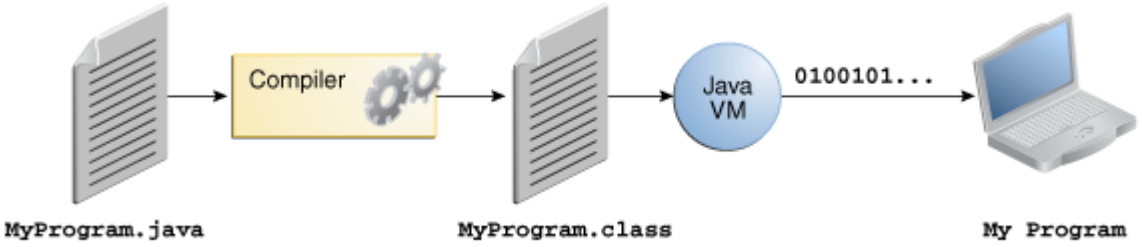
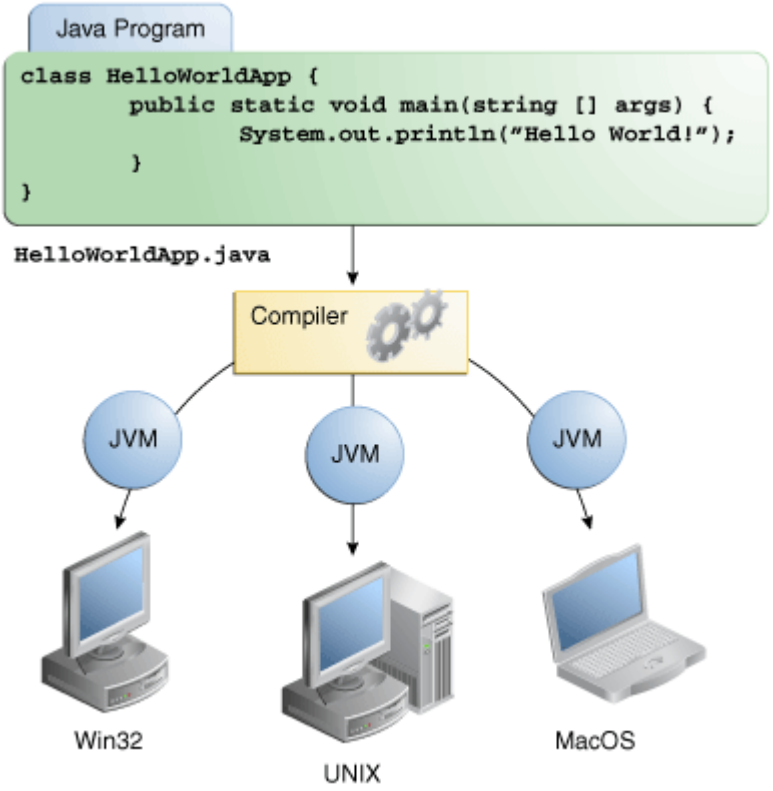
Features of Java

- Simple
- Architecture neutral (java follows 'Write-once-run-anywhere' approach)
- Object oriented
- Portable
- Distributed (the program can be design to run on computer networks)
- High performance
- Multi-threaded
- Robust (pointers removed, automatic memory management, garbage-collector)
- Dynamic (OOP – inheritance, reuse the code)
- Secure

Software development process – overview
(native code vs. byte-code)



Portability (Java VM)

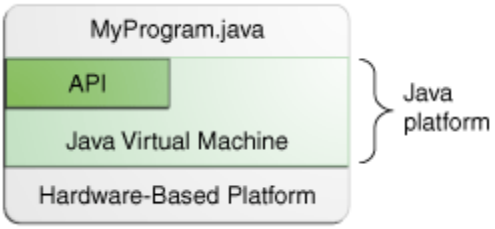


Software-only platform (Java) vs. Software-Hardware platforms

Java Platform Components:

Java Virtual Machine
Java application Programming Interface (API)

Program insulated from the hardware



Java Platforms:

- Java SE – Standard Edition
- Java EE – Enterprise Edition
- Java ME – Micro Edition

Other characteristics:

- C and C++ related (+ other languages)
- production language and not research language
- compiled and interpreted
- fully object oriented (no procedural programming)

Lexical structure

Case Sensitivity

Character set

- Unicode

  Divizat in subintervale – blocuri
  ex:
  - \u0030 - \u0039 : cifre ISO-Latin 0 – 9
  - \u03B1 - \u03C9 : simboluri grecesti

Keywords (reserved words)

Abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
Boolean	do	if	private	this
Break	double	implements	protected	throw
Byte	else	import	public	throws
Case	enum****	instanceof	return	transient
Catch	extends	int	short	try
Char	final	interface	static	void
Class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Not keywords (part of the language syntax) but literals, but not allowed
- true, false, null

Identifiers

- nu pot sa fie keywords sau reserved-words
- sir nelimitat de litere si cifre Unicode, incepand cu o litera, `'$'` sau `'_'`
- este de evitat `'$'` sau `'_'` la inceput (Oracle – documentation)
- “White space” nu se admite
- Case sensitivity (Name, name or NAME – identificatori diferiti)

Comments

1. `//` single-line comments
2. `/*` multi-line
   comments `*/`
3. `/**` multi-line
   documentation comments `*/`

Imbricarea nu e posibila.
`/*` sau `*/` pot apare in secventa `//` - dar nu mai au semnificatie
`//` poate apare in secventa `/*`, `*/` dar nu mai are semnificatie

Literals

Java literals
- Integer Literals
- Floating-point Literals
- Boolean Literals
- Character Literals
- String Literals

Integer Literals (normali 32b sau lungi 64b - L,l)

```
int octLit = 0400;    // octal equivalent of decimal 256
int hexLit = 0x100;   // decimal 256, hexadecimal equivalent
int decLit = 256;     // decimal 256
```

Floating-point Literals

```
float float1 = 89.0;      // Type mismatch: cannot convert
                          // from double to float
float float2 = 89.0f;     //OK
double double1 = 89.0D;   //OK
double double2 = 89.0d;   //OK
```

```
double double3 = 89.0;    // OK, by default floating point
                          // literal is double
```

Boolean Literals

```
boolean boolFalse = false;
boolean boolTrue = true;
```

0 / diferit de 0 nu mai au semnificatia de false / true din C, C++

Character Literals

```
char charLit = 'a';
```

Secventele escape predefinite in Java sunt:

- `'\b'` : Backspace (BS)
- `'\t'` : Tab orizontal (HT)
- `'\n'` : Linie nou (LF)
- `'\f'` : Pagina noua (FF)
- `'\r'` : Inceput de rand (CR)
- `'\"'` : Ghilimele
- `'\''` : Apostrof
- `'\\'` : Backslash

String Literals

`String` stringLit = "String Literal";

Separators

( ) [ ] { } ; , .

```
String language = "Java";
```

" - The double quotes are used to mark the beginning and the end of a string.

; - The semicolon is used to end each Java statement.

```
System.out.println("Java language");
```

() Parentheses (round brackets) always follow a method name. Between the parentheses we declare the input parameters. The parentheses are present even if the method does not take any parameters.

. The dot character separates the class name (System) from the member (out) and the member from the method name (println()).

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
```

The square brackets `[]` are used to denote an array type. They are also used to access or modify array elements. The curly brackets `{}` are also used to initiate arrays. The curly brackets are also used enclose the body of a method or a class.

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

, The comma character separates variables in a single declaration.

White Space

White-space in Java consists of the ASCII space character (SP), the ASCII horizontal tab character (HT), the ASCII form-feed character (FF), and line terminators.

White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code.

```
int i = 0;
```

White spaces are required in some places (for example between the **int** keyword and the variable name). In other places, white spaces are forbidden. They cannot be present in variable identifiers or language keywords.

```
int a=1;
int b = 2;
int c  =  3;
```

The amount of space put between tokens is irrelevant for the Java compiler.

Coding Conventions

Conventions are best practices followed by programmers when writing source code. Each language can have its own set of conventions. Conventions are not strict rules; they are merely recommendations for writing good quality code. A few conventions that are recognized by Java programmers:

- Class names begin with an uppercase letter.
- Method names begin with a lowercase letter

- The public keyword precedes the static keyword when both are used.
- The parameter name of the **main()** method is called **args**.
- Constants are written in uppercase.
- Each subsequent word in an identifier name begins with a capital letter.

Operators

Simple Assignment Operator

=	Simple assignment operator

Arithmetic Operators

+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Unary Operators

+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
- -	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

Equality and Relational Operators

`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

Conditional Operators

`&&`	Conditional-AND
`||`	Conditional-OR
`?:`	Ternary (shorthand for if-then-else statement)

Type Comparison Operator

`instanceof`	Compares an object to a specified type

Bitwise and Bit Shift Operators

`~`	Unary bitwise complement
`<<`	Signed left shift
`>>`	Signed right shift
`>>>`	Unsigned right shift
`&`	Bitwise AND

^ Bitwise exclusive OR

| Bitwise inclusive OR

More about operators

- operatori matematici:

  Este permisa notatia prescurtata de forma

  ```
  leftval operatie = rightval
  ```

  ```
  x += 2
  x -= 2
  ```

  operatori pentru autoincrementare:

  x++ (post), ++x (pre)

  operatori pentru autodecrementare:

  x-- (post), --x (pre)

Evaluarea expresiilor logice se face prin “metoda scurtcircuitului”: evaluarea se opreste in momentul in care valoarea de adevar a expresiei este sigur determinata.

- operatorul if-else:

expresie-logica ? valoare-true : valoare-false

- operatorul , (virgula) folosit pentru evaluarea secventiala a operatiilor:

  ```
  int x = 0, y = 1, z = 2;
  ```

- operatorul + pentru concatenarea sirurilor:

  ```
  String s1 = "Ana";

  String s2 = "mere";

  int x = 10;

  System.out.println(s1 + " are " + x + " " + s2);
  ```

- operatori pentru conversii (cast) : (tip-de-data)

```
int a = (int)'a';

char c = (char)96;

int i = 200;

long l = (long)i; //widening conversion

long l2 = (long)200;

int i2 = (int)l2; //narrowing conversion
```

Precedence and associativity of operators:

Precedence	Operator	Description	Associativity

1	`[]`
	`()`
	`.`
		Array index
		method call
		member access
			Left -> Right

2	`++`
	`--`
	`+ -`
	`~`
	`!`
		pre or postfix increment
		pre or postfix decrement
		unary plus, minus
		bitwise NOT
		logical NOT
			Right -> Left

3	`(type cast)` `new`	type cast object creation	Right -> Left

4	`*`
	`/`
	`%`
		multiplication
		division
		modulus (remainder)
			Left -> Right

5	`+ -` `+`	addition, subtraction string concatenation	Left -> Right

6	`<<`
	`>>`
	`>>>`
		left shift
		signed right shift
		unsigned or zero-fill right shift
			Left -> Right

7	`<`
	`<=`
	`>`
	`>=`
	`instanceof`
		less than
		less than or equal to
		greater than
		greater than or equal to
		reference test
			Left -> Right

8	== !=	equal to not equal to	Left -> Right
9	&	bitwise AND	Left -> Right
10	^	bitwise XOR	Left -> Right
11	|	bitwise OR	Left -> Right
12	&&	logical AND	Left -> Right
13	||	logical OR	Left -> Right
14	? :	conditional (ternary)	Right -> Left
15	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment and short hand assignment operators	Right -> Left

Ex:

```
a = b = c = 8
```

```
System.out.println("1 + 2 = " + 1 + 2);     // out: 1 + 2 = 12
System.out.println("1 + 2 = " + (1 + 2));   // out: 1 + 2 = 3

System.out.println(1 + 2 + " = 1 + 2");     // out: 3 = 1 + 2
```

1. Data Types

1.1. **Primitive Data Types**

The Java programming language is statically-typed (all variables must first be declared before they can be used) => Type and name

8 primitive data types:

- **byte** — 8 bit signed two's complement integer (range min: -128, max:+127). Useful for saving memory in large arrays, where the memory savings actually matters
- **short** — 16-bit signed two's complement integer (range min: -32,768, max: +32,767) Useful for saving memory in large arrays, where the memory savings actually matters
- **int** — 32-bit signed two's complement integer (range min: -2^{31}, max: $+2^{31}-1$)
- **long** — 64-bit signed two's complement integer (range min: -2^{63}, max: $+2^{63}-1$)
- **float** — single precision 32-bit (N = 24, K = 8)
- **double** — double precision 64-bit (N = 53, K = 11)
- **boolean** — 2 values: **true** and **false** (represents one bit of information, its "size" isn't something that's precisely defined)
- **char** — single 16-bit Unicode character (range min: \u0000' (or 0), max: '\uffff' (or 65535))

Not Primitive Data Type – but special:
Java programming language provides special support for character strings via the **java.lang.String** class. Enclosing your character string within double quotes will automatically create a **new String object**; for example, String s = "this is a string"; String objects are ***immutable***, which means that once created, their values cannot be changed.

Default values for Data Types

Data Type	**Default Value**
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	***null***
boolean	false

- Formatul si dimensiunea tipurilor primitive de date nu depinde de platforma (ca in alte limbaje). Java elimina dependenta de platforma.

1.2. Tipuri de Date Referinta

Vectorii, Clasele si Interfetele – Tipuri referinta.

Valoarea unei variable de tip referinta este o adresa de memorie (referinta) catre date sau set de date ale variabilei

Java a eliminat (din C): pointer, struct si union.

Pointer – inlocuit de referent (considerat sursa de erori)
Struct si union – tipurile compuse de date sunt formate prin intermediul claselor

2. Variables:

2.1.

Instance Variables (Non-Static Fields) Objects store their individual states in "non-static fields", that is, fields declared without the static keyword. *Instance variables* - their values are unique to each *instance* of a class (to each object)

Class Variables (Static Fields) *Class variable* is any field declared with the **static** modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. Keyword **final** could be added

Local Variables Methods store their individual states in local variable. Visible to the methods in which they are declared; they are not accessible from the rest of the class `(int count = 0;)`.

Parameters `public static void main(String[] args).` Variable args is parameter

2.2. Declararea Variabilelor Locale

Declarare

type name;

```
int count;
```

Declarare cu initializare

type name = value;

```
int count = 7;
```

Declararea constantelor

final type NAME = value;

```
final int MAX_COUNT = 10;
```

Declarare si initializare mai multe variabile de acelasi tip, intr-o instructiune:

```
int count = 7, speed = 30, count1 = 53;
char C1 = 'a', c2 = 'b', c3 = 'c';
final int MIN_COUNT = 4, MAX_COUNT = 100;
```

3. Expressions, Statements and Blocks

Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

3.1. Expressions:

An **expression** is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

```
1 * 2 * 3;          // easy to read (precedence not relevant)

x + y / 100;        // ambiguous
(x + y) / 100;      // unambiguous, recommended
x + (y / 100);      // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

3.2. Statements:

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution.

Types of statements:

3.2.1. Expression statements

3.2.2. Declaration statements

3.2.3. Control flow statements

3.2.1. Expression statements

The following types of expressions can be made into a statement by terminating the expression with a semicolon (;)

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

```
// assignment statement
aValue = 8933.234;

// increment statement
aValue++;

// method invocation statement
System.out.println("Hello World!");

// object creation statement
Bicycle myBike = new Bicycle();
```

3.2.2. Declaration statements

A declaration statement declares a variable.

```
// declaration statement
double aValue = 8933.234;
```

3.2.3. Control flow statements

A control flow statements regulate the order in which statements get executed (after Blocks)

3.3. Blocks:

A **block** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

```
if (condition) { // begin block 1
     System.out.println("Condition is true.");
} // end block 1
```

```
else { // begin block 2
     System.out.println("Condition is false.");
} // end block 2
```

4. Control flow statements (Controlul executiei)

(allow the program to conditionally execute particular blocks of code)

4.1. Decision-making statements (Instructiuni de decizie)

4.1.1. if-then
4.1.2. if-then-else
4.1.3. switch

4.2. Looping statements (Instructiuni de salt)

4.2.1. while
4.2.2. do-while
4.2.3. for

4.3. Branching statements

4.3.1. break
4.3.2. continue
4.3.3. return

4.1. Decision-making statements (Instructiuni de decizie)

4.1.1. if-then statement

```
int testScore = 75;
char grade;

if (testScore >= 90) {
     grade = 'A';
}
```

```
// single statement in block – braces removed (not recommended)
if (testScore >= 90)
     grade = 'A';
```

4.1.2. if-then-else statement

```
int testScore = 75;
char grade;

if (testScore >= 90) {
     grade = 'A';        // for (testScore >= 90)
} else {
     grade = 'B';        // for (testScore < 90)
}
```

```
int testScore = 75;
char grade;

if (testScore >= 90) {
     grade = 'A';        // for (testScore >= 90)
} else if (testScore >= 80) {
     grade = 'B';        // for (testScore < 90
                         // and(testScore >= 80)
} else {
     grade = 'C';        // for (testScore < 90)
                         // and(testScore < 80)
}
```

Note: Condition satisfied -> the remaining conditions are not evaluated.

4.1.3. switch statement

Can have a number of possible execution paths

```
Switch (variable) {
     case value1:
          statement;
          ...
          statement;
          break;
```

```
     case value2:
          statement;
          ...
          statement;
          break;

     case value3:
          statement;
          ...
          statement;
          break;

     ...
     default:
          statement;
          ...
          statement;
          break;
}
```

Necessity of **break**; statement

> All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered.

Ultimul **break** nu e necesar – dar e recomandat sa fie lasat acolo pentru dezvoltari ulterioare
Sectiunea **default** pentru cazul in care nu toate valorile posibile sunt tratate explicit

Variable can be:

> // from primitive types
> **byte**
> **short**
> **int**
> **char**
>
> **String**

Enum Types (special data type that enables for a variable to be a set of predefined constants)

// special classes that wrap certain primitive types

Byte
Short
Integer
Character

4.2. **Looping statements** (Instructiuni de salt)

4.2.1. while statement

```
while (expression) {
     statement(s)
}
```

```
int count = 1;
while (count < 11) {
     System.out.println("Count is: " + count);
     count++;
}
```

Infinite loop

```
while (true) {
     statement(s)
}
```

4.2.2. do-while statement

```
do {
     statement(s)
} while (expression);
```

```
int count = 1;
do {
     System.out.println("Count is: " + count);
     count++;
} while (count < 11);
```

Executed at least one time

4.2.3. for statement

Provides a compact way to iterate over a range of values.

```
// general form
for (initialization; termination; increment) {
     statement(s)
}
```

The **initialization** expression initializes the loop; it's executed once, as the loop begins.
When the **termination** expression evaluates to false, the loop terminates.
The **increment** expression is invoked after each iteration through the loop

```
for (int i = 1; i < 11; i++) {
     System.out.println("Count is: " + i);
}
```

The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

Infinite loop

```
for ( ; ; ) {
     statement (s)
}
```

```
for(int i = 0, j = 100 ; i < 100 && j > 0; i++, j--) {
     statement (s)
}
```

Enhanced for statement

Is designed for iteration through **Collections** or **arrays**

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};

for (int item : numbers) {
     System.out.println("Count is: " + item);
}
```

item is the **value**

Enhanced for is the recommended form when possible.

4.3. **Branching statements**

4.3.1. break statement

Unlabeled or labeled

Unlabeled **break** used in **switch** statement

Unlabeled **break** to terminate a loop (**while**, **do-while** or **for**)

An **unlabeled break** statement terminates the **innermost switch**, **for**, **while**, or **do-while** statement, but a **labeled break** terminates an **outer** statement.

```
for (int i = 1; i < 11; i++) {
     System.out.println("Count is: " + i);
     if (i == 5) {
          break;
     }
     System.out.println("Count +1 is: " +(i + 1));
}


breakLabel:

for (int i = 1; i < 11; i++) {
     for (int j = 1; j < 11; j++) {
          System.out.println("i is: " + i);
          if (i == 5) {
               break breakLabel;
          }
          System.out.println("j is: " + j);
     }
}
```

4.3.2. continue statement

Unlabeled or labeled

Unlabeled **continue** skips the current iteration of a loop (**while**, **do-while** or **for**)

An **unlabeled continue** statement skips the current iteration of the **innermost for**, **while**, or **do-while** statement, but a **labeled continue** skips the current iteration of an **outer** statement.

```
for (int i = 1; i < 11; i++) {
     System.out.println("Count is: " + i);
     if (i == 5) {
          continue;
     }
     System.out.println("Count +1 is: " +(i + 1));
}

continueLabel:

for (int i = 1; i < 11; i++) {
     for (int j = 1; j < 11; j++) {
          System.out.println("i is: " + i);
          if (i == 5) {
               continue continueLabel;
          }
          System.out.println("j is: " + j);
     }
}
```

4.3.3. return statement

The **return** statement exits from the current method, and control flow returns to where the method was invoked.

Two forms of **return**:

return with returned value. The value returned has to match the type required

```
return count;       // return a value
return ++count;     // return expression
```

return with no value returned

```
return;             // no value returned
```

1. Arrays

An **array** is a container object that holds a **fixed number of values** of a **single type**.
The length of an array is established when the array is created.
After creation its length is fixed.

Zero-based numbering (numbering or index origin = 0)

1.1. Declarare
1.2. Creare (instantiere)
1.3. Initializare
1.4. Declarare + initializare
1.5. Multidimensional array
1.6. Array Dimension
1.7 Copy Array
1.8 Array Manipulations

```
// declares an array of integers
int[] myArray;

// allocates memory for 10 integers
myArray = new int[10];


// initialize first element
myArray[0] = 10;
// initialize second element
myArray[1] = 20;

...

// initialize last element
myArray[9] = 100;
```

1.1. Declarare

Array declaration: 2 components: the array's type and the array's name.
An array's type is written as **type[]**, where type is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty).

Similar putem avea:

```
byte[] myArrayOfBytes;
short[] myArrayOfShorts;
long[] myArrayOfLongs;
float[] myArrayOfFloats;
double[] myArrayOfDoubles;
boolean[] myArrayOfBooleans;
char[] myArrayOfChars;
String[] myArrayOfStrings;
```

Array declaration – another (discouraged) form

```
// this form is discouraged
float myArrayOfFloats[];
```

1.2. Creare (instantiere)

```
// create an array of integers
myArray = new int[10];
```

Alocarea de memorie este realizata doar prin intermediul operatorului **new**:

```
int[10] myArray     // eroare
int myArray[10]     // eroare
```

1.3. Initializare

```
// initialize first element
myArray[0] = 10;
// initialize second element
myArray[1] = 20;

...

// initialize last element
myArray[9] = 100;
```

```
// Accesare
System.out.println("Element 1, index 0: " + anArray[0]);
```

1.4. Declarare + initializare

Length of the array determined by the number of elements

```
int[] myArray = {
                10, 20, 30,
                40, 50, 60,
                70, 80, 90, 100
};
```

1.5. Multidimensional Array

```
// array 5 rows – 10 columns
int[][] myArrayM = new int [5][10];
```

In the Java programming language, a multidimensional array is an array whose components are themselves arrays.

Consecinta afirmatiei de mai sus este ca liniile (rows) din array pot varia in lungime:

```
String[][] names = {
                {"Mr. ", "Mrs. ", "Ms. "},
                {"Smith", "Jones"}
};

// Mr. Smith
System.out.println(names[0][0] + names[1][0]);
// Ms. Jones
System.out.println(names[0][2] + names[1][1]);
```

1.6. Array Dimension

Variabila publica a clasei – **length**.

```
myArray = new int[10];
// myArray.lenght are valoarea 10

int[][] myArrayM = new int [5][10];
// myArray[0].lenght are valoarea 10
```

1.7. Copy array

```
int[] myArray = {
               10, 20, 30,
               40, 50, 60,
               70, 80, 90, 100
};

myDestArray = new int[10];

// 1
for (int i = 0; i < myArray.lenght; i++)
     myDestArray[i] = myArray[i];
```

The **System** class has an **arraycopy** method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
          Object dest, int destPos, int length)

int[] myArray = {
               10, 20, 30,
               40, 50, 60,
               70, 80, 90, 100
};

myDestArray = new int[7];

// 2
System.arraycopy(myArray, 2, myDestArray, 0, 7);
```

1.8. Array Manipulations

Methods to provide some common manipulation tasks (copying, sorting and searching) in **java.utils.Arrays** class

```
char[] myDest Array =
                java.util.Arrays.copyOfRange(copyFrom, 2, 9);
```

The second parameter of the **copyOfRange** method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, exclusively.

Some other useful operations provided by methods in the **java.util.Arrays** class, are:

Searching an array for a specific value to get the index at which it is placed (the **binarySearch** method).

Comparing two arrays to determine if they are equal or not (the **equals** method).

Filling an array to place a specific value at each index (the **fill** method).

Sorting an array into ascending order. This can be done either sequentially, using the **sort** method, or concurrently, using the **parallelSort** method - Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

```
java.util.Arrays.sort(myArray);
```

2 Siruri de caractere

Un sir de caractere poate fi reprezentat printr-un **vector** format din elemente de tip **char**, un obiect de tip **String** sau un obiect de tip **StringBuffer**.
Daca un sir de caractere este constant (nu se doreste schimbarea continutului pe parcursul executiei programului) atunci el va fi declarat de tipul **String**, altfel va fi declarat de tip **StringBuffer**. Diferenta principala intre aceste clase este ca **StringBuffer** pune la dispozitie metode pentru modificarea continutului sirului, cum ar fi: **append**, **insert**, **delete**, **reverse**.

Uzual, cea mai folosita modalitate de a lucra cu siruri este prin intermediul clasei String, care are si unele particularitati fata de restul claselor menite sa simplifice cat mai mult folosirea sirurilor de caractere. Clasa StringBuffer va fi utilizata predominant in aplicatii dedicate procesarii textelor cum ar fi editoarele de texte.

```
String s = "abc";        // missing new (ok)

String s = new String("abc");

char data[] = {'a', 'b', 'c'};
String s = new String(data);
```

In Java, operatorul de concatenare **+** este foarte flexibil: permite concatenarea sirurilor cu obiecte de orice tip care au o reprezentare de tip sir de caractere.

```
System.out.print("Vectorul v are " + v.length +
                          " elemente");
String x = "a" + 1 + "b";
```

String x = new StringBuffer().append("a").append(1).append("b").toString()

Nota: Ordinea de efectuare a operatiilor.

```
S = 1 + 2 + "a" + 1 + 2; //"3a12"
```

3 Classes

3.1. Declarare (definire)

Clasele sunt definite in felul urmator (minimal)

```
class MyClass {
     // field, constructor, and
     // method declarations
}
```

This is a **class declaration**. The **class body** (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: **constructors** for initializing new objects, declarations for the **fields** that provide the state of the class and its objects, and **methods** to implement the behavior of the class and its objects.

```
class MyClass extends MySuperClass implements MyInterface {
     // field, constructor, and
     // method declarations
}
```

In general, class declarations can include these components, in order:

Modifiers such as public, private, and a number of others.

The class name, with the initial letter capitalized by convention.

The name of the class's parent (superclass), if any, preceded by the keyword **extends**. A class can only extend (subclass) one parent.

A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword **implements**. A class can **implement** more than one interface.

The class body, surrounded by braces, {}.

3.2. Declaring Member Variables

Kindes of variables:

Member variables in a class—these are called **fields**.
Variables in a method or block of code—these are called **local variables**.
Variables in method declarations—these are called **parameters**.

Field declarations are composed of **three components**, in order:

Zero or more modifiers, such as **public** or **private**.
The field's **type**.
The field's **name**.

3.2.1. Access modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field.

public modifier—the field is accessible from all classes.
private modifier—the field is accessible only within its own class.

In the spirit of encapsulation – metode de accesare (**getter, setter**):

```
private int myField;

//Constructor
public MyClass(int startField) {
     myField = startField;
}

//Getter
public int getField() {
     return myField;
}

//Setter
public void setField(int newValue) {
     myField = newValue;
}
```

Codul de mai sus ruleaza correct, dar genereaza neclaritate pentru dezvoltarea acestei clase deoarece campul “myField” este incapsulat cu getField / setField (ca pentru un nume de camp “field”). Incapsularea potrivita pentru “myField” este getMyField / setMyField. Neclaritatea generata nu se manifesta si in exterior, deoarece myField este invizibil in exteriorul clasei (are modificatorul de acces **private**).

Codul de mai sus scris fara neclaritati si usor de utilizat din exteriorul clase este:

```
private int myField;

//Constructor
public MyClass(int myField) {
     setMyField(myField);
}

//Getter
public int getMyField() {
     return myField;
}

//Setter
public void setMyField(int myField) {
     this.myField = myField;
}
```

3.2.2. Variable Types

All variables **must have a type**. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

3.2.2. Names

The same naming rules and conventions are used for method and class names, except that

- The first letter of a class name should be capitalized

- The first (or only) word in a method name should be a verb.

3.3. Methods

Example of a typical method declaration:

```
public double calculateAnswer(double wingSpan,
                              int numberOfEngines,
                              double length,
                              double grossTons) {

     //do the calculation here
}
```

The only required elements of a method declaration are the **method's return type**, **name**, a pair of parentheses, **()**, and a body between braces, **{}**.

More generally, method declarations have six components, in order:

Modifiers—such as public, private, and others you will learn about later.
The return type—the **data type of the value returned** by the method, or **void** if the method does not return a value.
The method name—the rules for field names apply to method names as well
The parameter list in parenthesis—a **comma-delimited list** of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
An exception list— later.
The method body, enclosed between braces—the method's code, including the declaration of local variables.

Definition: Two of the components of a method declaration comprise the **method signature**—the **method's name** and the **parameter types**.

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

3.3.1. Names

Although a method name can be any legal identifier, code conventions restrict method names. By convention, **method names should be a verb in lowercase** or a **multi-word name that begins with a verb in lowercase**, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

```
run
runFast
```

```
getBackground
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. Exception: **method overloading**.

3.3.2. Overloading Methods

The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures.

```
public void draw(String s) {
     ...
}

public void draw(int i) {
     ...
}

public void draw(double f) {
     ...
}

public void draw(int i, double f) {
     ...
}
```

Alternativa (greoaie) era sa definim

```
drawString
drawInt
drawDouble
```

Nota: **return Type** nu e parte a semnaturii (compilatorul nu il considera la diferentierea metodelor)

3.3.3. Constructors

Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

```
public MyClass(int startField1,
               int startField2,
```

```
                int startField3) {
        myField1 = startField1;
        myField2 = startField2;
        myField3 = startField3;
    }
```

To create a new MyClass object called myInstance, a constructor is called by the **new** operator:

```
MyClass myInstance = new MyClass(10, 20, 0);

new MyClass(10, 20, 0)   // creates space in memory for the
                         //object and initializes its fields
```

A class can have many constructors, including a no-argument constructor:

```
public MyClass() {
     myField1 = 1;
     myField2 = 5;
     myField3 = 2;
}


MyClass myInstance = new MyClass();
```

Multiple Constructors
Both constructors could have been declared in MyClass because they have **different argument lists**. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

No Constructors:
The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

Using access modifiers in a constructor's declaration: to control which other classes can call the constructor.

Note: If another class cannot call a MyClass constructor, it cannot directly create MyClass objects

3.5. Passing Information to a Method or a Constructor

Parameters - refers to the list of variables in a method declaration. **Arguments** - are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

3.5.1 Parameter Types

Any data type for a parameter of a method or a constructor: including primitive data types, such as doubles, floats, and integers, and reference data types, such as objects and arrays.

3.5.2 Arbitrary Number of Arguments

You can use a construct called **varargs** to pass an arbitrary number of values to a method. You use **varargs** when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually.
To use **varargs**, you follow **the type of the last parameter** by: **three dots** (**...**), then **a space**, and the **parameter name**. The method can then be called with **any number of that parameter**, including **none**.

```
public Polygon polygonFrom(Point... corners) {

     (corners[1].x - corners[0].x)
     ...
     (corners[3].x - corners[1].x)
}
```

You can see that, inside the method, **corners** is treated like an array. The method can be called either with **an array** or with **a sequence of arguments**. The code in the method body will treat the parameter as an array in either case.

Most common **varargs** with the printing methods; for example, this **printf** method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects.

3.5.3 Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.
The name of a parameter must be unique in its scope. It cannot be the same as the **name of another parameter** for the same method or constructor, and it cannot be the **name of a local variable** within the method or constructor.
A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to **shadow the field**. **Shadowing fields** can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field.

```
public class Circle {
     private int x, y, radius;

     public void setOrigin(int x, int y) {
          ...
     }
}
```

Parametri x si y **umbresc** campurile x si y (pentru utilizarea campurilor x si y in metoda se foloseste **this**)

3.5.4 Passing Primitive Data Type Arguments

Primitive arguments are passed into methods **by value**. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.

3.5.5 Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods **by value**. This means that when the method returns, the passed-in reference still **references the same object as before**. However, the values of the object's fields can be changed in the method, if they have the proper access level.

```
public void moveCircle(Circle circle, int deltaX, int deltaY){

     // code to move origin of circle to x + deltaX, y + deltaY
     circle.setX(circle.getX() + deltaX);
     circle.setY(circle.getY() + deltaY);

     // code to assign a new reference to circle – no permanence
     circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56);
```

4. Objects

4.1 Creating Objects

Trei parti:

Declaration	(Declarare)	Type Name
Instantiation	(Instatiere)	new operator
Initialization	(Initializare)	constructor

4.1.1 Declaring a Variable to Refer to an Object

```
type name;
```

Daca este o variabila primitiva atunci este alocata memoria. Daca nu este o variabila primitiva, atunci aceasta referinta e de valoarea **null** (doar operatorul **new** aloca, altfel – eroare compilare).

4.1.2 Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.
The phrase "**instantiating a class**" means the same thing as "**creating an object**." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The **new operator** requires a single, postfix argument: a call to a constructor.

The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created.

4.1.3 Initializing an Object

```
public class Point {
     public int x = 0;
     public int y = 0;

     //constructor
     public Point(int a, int b) {
          x = a;
          y = b;
     }
}
```

```
Point originOne = new Point(23, 94);
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's **no-argument constructor**, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

4.2 Using Objects

4.2.1 Referencing an Object's Fields

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name as:

```
objectReference.fieldName

//for:
Point originOne = new Point(23, 94);

originOne.x


originX = new Point (23, 94).x;    //Point scope ended
```

This statement creates a new Point object and immediately gets its height. In essence, the statement calculates the default x of a Point. Note that after this statement has been executed, the program no longer has a reference to the created Point, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

4.2.2 Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);

objectReference.methodName();
```

4.2.3 The Garbage Collector

The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called **garbage collection**.

An **object is eligible for garbage collection** when there are no more references to that object. References that are held in a variable are usually dropped when the **variable goes out of scope**. Or, you can explicitly drop an object reference by setting the variable to the special value **null**. A program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

5. More on Classes

5.1. Returning a Value from a Method

A method returns to the code that invoked it when it:

- completes all the statements in the method,
- reaches a **return** statement, or
- throws an exception,

whichever occurs first.

Methods declared **void** – **return** not needed (or **return;** for branch out the method)

```
return;

return returnValue;

return expression;

return reference;
```

5.2. Using the **this** Keyword

Within an instance method or a constructor, **this** is a reference to the **current object** — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using **this**.

5.2.1 Using **this** with a Field

The main reason to use it is the **shadowing**

```
public class Point {
     public int x = 0;
     public int y = 0;

     //constructor
     public Point(int x, int y) {
          this.x = x;
          this.y = y;
     }
}
```

5.2.2 Using **this** with a Constructor

From within a constructor, you can also use the **this** keyword to call **another constructor in the same class**. Doing so is called an explicit constructor invocation.

```
public class Rectangle {
     private int x, y;
     private int width, height;

     public Rectangle() {
          this(0, 0, 1, 1);
     }
```

```
    public Rectangle(int width, int height) {
         this(0, 0, width, height);
    }

    public Rectangle(int x, int y, int width, int height) {
         this.x = x;
         this.y = y;
         this.width = width;
         this.height = height;
    }

    ...

}
```

If present, the invocation of another constructor **must be the first line** in the constructor.

5.3. Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

Top level—**public**, or **package-private** (no explicit modifier).

Member level—**public**, **private**, **protected**, or **package-private** (no explicit modifier).

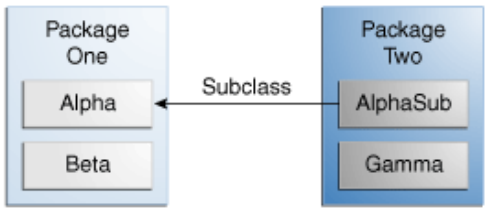
The **private** modifier - the member can only be accessed in its own class
The **protected** modifier - the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Coloane:

1. Clasa proprie
2. Oricare Clase din pachet
3. Subclasa exterioara pachetului
4 Clase exterioara pachetului si nu subclasa a clase



Visibility				
Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

Use the most restrictive access level that makes sense for a particular member. Use **private** unless you have a good reason not to.

Avoid public fields except for constants. Public fields – not recommended for production code. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

5.4 Static members

The use of the **static** keyword to create fields and methods that belong to the class, rather than to an instance of the class.

5.4.1 Class Variables

Target: variables common to all objects. The fields with the **static** modifier in their declaration are called **static fields** or **class variables**. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated **without creating an instance** of the class.

Static variable initialization

Static variables are initialized **when class is loaded**.
Static variables in a class are initialized **before any object of that class can be created.**
Static variables in a class are initialized **before any static method of the class runs.**

```
public class ClassName {
     // instance variable
     private int instanceIntVariable;
     ...

     // class variable
     private static int classIntVariable;
     ...
}
```

```
ClassName.classIntVariable ...

instanceName.classIntVariable ...  // discouraged not clear
```

5.4.2 Class Methods

The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in:

```
ClassName.methodName(args)

instanceName.methodName(args) // discouraged – not clear
```

Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods **cannot** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the **this** keyword as there is no instance for **this** to refer to.

5.4.3 Constants

The **static** modifier, in combination with the **final** modifier, is also used to define **constants**. The **final** modifier indicates that the value of this field cannot change.
For example, the following variable declaration defines a constant named **PI**, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

```
static final double PI = 3.141592653589793;
```

Naming:
By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a **compile-time constant**. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

5.5 Initializing Fields

```
public static int variableName = ...
```

It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

This form of initialization has limitations because of its simplicity

If initialization requires some logic, instance variables can be initialized in **constructors**. For **class variables**, the Java programming language includes **static initialization blocks**

5.5.1 Static Initialization Blocks

A **static initialization block** is a normal block of code enclosed in braces, **{ }**, and proceeded by the **static** keyword.

```
static {
     // initialization code here
}
```

A class can have **any number of static initialization blocks**, and they **can appear anywhere** in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks — you can write a private static method:

```
class ClassName {
     public static varType myVar = initializeClassVariable();

     private static varType initializeClassVariable() {

          // initialization code here
     }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

```
public class StaticExample{
     static {
          System.out.println("This is first static block");
     }

     public StaticExample(){
          System.out.println("This is constructor");
     }

     public static String staticString = "Static Variable";

     static {
          System.out.println("This is second static block and "
                         + staticString);
     }

     public static void main(String[] args){
          StaticExample statEx = new StaticExample();
          StaticExample.staticMethod2();
     }

     static {
          staticMethod();
          System.out.println("This is third static block");
     }

     public static void staticMethod() {
          System.out.println("This is static method");
     }

     public static void staticMethod2() {
          System.out.println("This is static method2");
     }
}
```

Output:

```
This is first static block
This is second static block and Static Variable
This is static method
This is third static block
This is constructor
This is static method2
```

First all static blocks are positioned in the code and they are executed when the class is loaded into JVM. Since the static method **staticMethod()** is called inside third static block, its executed before calling the main method. But the **staticMethod2()** static method is executed after the class is instantiated because it is being called after the instantiation of the class.

5.5.2 Initializing Instance Members

Constructor or: there are two alternatives to using a constructor to initialize instance variables**: initializer blocks** and **final methods**.

Initializer blocks for instance variables look just like static initializer blocks, but **without** the **static** keyword:

```
{
     // initialization code here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors. A **final method** cannot be overridden in a subclass

```
class ClassName {
     private varType myVar = initializeInstanceVariable();

     protected final varType initializeInstanceVariable() {

          // initialization code here
     }
}
```

The advantage: possibility to reuse the initialization method.

1. OOP concepts in Java

- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**

1.1. Encapsulation

Enforcing Data Encapsulation

- We must design our objects so that they have state and behaviors. We create private fields that hold the state and public methods that are the behaviors.

- **Accessor methods (getter)**. For every private field we can create a public method that will return its value.
- **Mutator methods (setter)**. For every private field we can create a public method that will set its value. If you want a private field to be read-only, do not create a mutator method for it.

Reasons for Data Encapsulation

Keeping the state of an object legal. By forcing a private field of an object to be modified by using a public method, we can add code into the mutator or constructor methods to make sure the value is legal.
We can change the implementation of an object. As long as we keep the public methods the same we can change how the object works without breaking the code that uses it. The object is essentially a "black box" to the code that calls it.
Re-use of objects. We can use the same objects in different applications because we have combined the data and how it's manipulated in one place.
The independence of each object. If an object is incorrectly coded and causing errors it's easy to test and fix because the code is in one place. In fact, the object can be tested independently from the rest of the application. The same principle can be used in large projects where different programmers can be assigned the creation of different objects.

1.2. Abstraction

Abstract Classes and Abstract Methods

A class that is declared using “abstract” keyword is known as abstract class. It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with abstract methods (without an implementation, without braces, and followed by a semicolon). An abstract class can not be **instantiated** (you are not allowed to create **object** of Abstract class).

Abstract class declaration

Specifying **abstract keyword** before the class during declaration makes it abstract. Have a look at below code:

```
// Declaration using abstract keyword
abstract class AbstractDemo{
     // Concrete method: body and braces
     public void myMethod(){
     //Statements here
     }

     // Abstract method: without body and braces
     abstract public void anotherMethod();

}
```

Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction.
Interfaces are used for **100% abstraction** (full abstraction)

Two rules:

1) If the class is having few abstract methods and few concrete methods: declare it as abstract class.
2) If the class is having only abstract methods: declare it as interface.

Abstract vs. Concrete

A class which is not abstract is referred as **Concrete class**.

Key Points:

- An abstract class is of no use until it is extended by another class.
- If you declare an **abstract method** in a class then you must declare the **class abstract** as well. You can't have abstract method in a non-abstract class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- Abstract class can have non-abstract method (concrete) as well.

Abstract methods

An **abstract method** is a method that is declared without an implementation (without braces, and followed by a semicolon)

Syntax:

public abstract void display();

Points to remember about abstract method:

1) Abstract method has no body.
2) Always end the declaration with a **semicolon(;)**.
3) It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
4) Abstract method must be in an abstract class.

Note: When an abstract class is subclassed, the subclass usually **provides implementations for all of the abstract methods in its parent class**. However, if it does not, then the subclass must also be declared abstract.

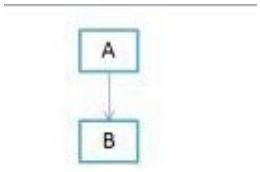
1.3. Inheritance

Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance give rise to a hierarchy.

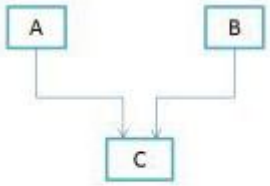
Subclasses and Superclasses

A subclass is a modular, derivative class that inherits one or more properties from another class (called the superclass). The properties commonly include class data variables, properties, and methods or functions. The superclass establishes a common interface and foundational functionality, which specialized subclasses can **inherit, modify, and supplement**. The software inherited by a subclass is considered **reused** in the subclass.
In some cases, a subclass **may customize or redefine a method inherited** from the superclass. A superclass method which can be redefined in this way is called a **virtual method**.

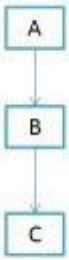
1.3.1 Types of inheritance in Java: Single, Multiple, Multilevel & Hybrid



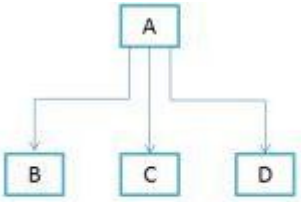
(a) Single Inheritance



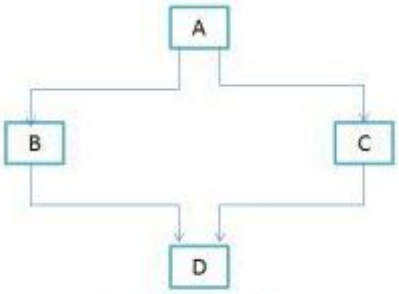
(b) Multiple Inheritance



(d) Multilevel Inheritance



(c) Hierarchical Inheritance



(e) Hybrid Inheritance

1.4. Polymorphism

Polymorphism means **one name, many forms**. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality.

There are 2 basic types of polymorphism.

> **Overridding**, also called **run-time polymorphism**. Method will be used for method overriding is determined at runtime based on the dynamic type of an object.
> **Overloading**, this is referred to as **compile-time polymorphism**. For method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled.

1.4.1 Runtime Polymorphism (Dynamic polymorphism)

Method overriding is a perfect example of **runtime polymorphism**. In this kind of polymorphism, reference of **class X** can hold **object of class X** or an **object of any sub classes of class X**. For e.g. if **class Y extends class X** then both of the following statements are valid:

```
Y obj = new Y();
//Parent class reference can be assigned to child object
X obj = new Y();
```

Since in method overriding, both the classes (base class and child class) have same method, compile doesn't figure out which method to call at compile-time. In this case JVM (Java Virtual Machine) decides which method to call at runtime that's why it is known as runtime or dynamic polymorphism.

```
public class X {
     public void methodA() { //Base class method
          System.out.println ("hello, I'm methodA of class X");
     }
}

public class Y extends X {
     public void methodA() { //Derived Class method
          System.out.println ("hello, I'm methodA of class Y");
     }
}
```

```
public class Z {
     public static void main (String args []) {
          X obj1 = new X(); // Reference and object X
          X obj2 = new Y(); // X reference but Y object
          obj1.methodA();
          obj2.methodA();
     }
}
```

Output:
hello, I'm methodA of class X
hello, I'm methodA of class Y

- methodA has different - 2 forms in child and parent class - methodA here is polymorphic.

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class (base class).

Rules of method overriding in Java

Argument list: The argument list of overriding method must be the same as that of the method in parent class. The data types of the arguments and their sequence should be maintained as it is in the overriding method.

Access Modifier: The Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of base class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier as all of the three are more restrictive than public.

private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.

If a class **is extending an abstract class** or **implementing an interface** then it has to override all the abstract methods unless the class itself is a abstract class.

Super keyword in Overriding

super keyword is used for calling the parent class method/constructor:

super.methodname() calling the specified method of base class
super() calls the constructor of base class.

1.4.2 Compile time Polymorphism (Static polymorphism)

Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one method with same name but with different number of arguments or different types of arguments or both. **Method overloading**

Argument lists could differ in

1. Number of parameters.
2. Data type of parameters
3. Sequence of Data type of parameters.

Return type not relevant

```
class X {
     void methodA(int num) {
          System.out.println ("methodA:" + num);
     }

     void methodA(int num1, int num2) {
          System.out.println ("methodA:" + num1 + "," + num2);
     }

     double methodA(double num) {
          System.out.println("methodA:" + num);
          return num;
     }
}


class Y {
     public static void main (String args []) {
          X obj = new X();
          double result;
```

```
            obj.methodA(20);
            obj.methodA(20, 30);
            result = obj.methodA(5.5);
            System.out.println("Answer is:" + result);
        }
}
```

Output:
methodA:20
methodA:20,30
methodA:5.5
Answer is:5.5

As you can see in the above example that the class has **three variance of methodA** or we can say **methodA is polymorphic in nature** since it is having three different forms. In such scenario, compiler is able to figure out the method call at compile-time that's the reason it is known as compile time polymorphism.

2. Interfaces

2.1. What is an Interface

Interfetele duc conceptul de clasa abstracta cu un pas inainte prin eliminarea oricaror implementari de metode, punand in practica unul din conceptele programarii orientate obiect si anume cel de separare a modelului unui obiect (interfata) de implementarea sa. Asadar, o interfata poate fi privita ca un **protocol de comunicare intre obiecte** (**contract**)

O interfata Java **defineste un set de metode** dar **nu specifica nici o implementare** pentru ele. O **clasa care implementeaza** o interfata trebuie obligatoriu sa specifice implementari **pentru toate metodele interfetei**.

Definitie

O interfata este o colectie de metode fara implementare si declaratii de constante.

Interfetele permit, alaturi de clase, definirea unor noi tipuri de date.

Interface looks like class but it is not a class. An interface can have methods and

variables just like the class but the methods declared in interface are **by default abstract** (**only method signature, no body**). Also, **the variables declared in an interface are public, static & final by default**.

Used for **abstraction**. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. **The class that implements interface must implement all the methods of that interface**. Also, java programming language does not support multiple inheritance, using interfaces we can achieve this as a class can implement more than one interfaces, however it cannot extend more than one classes.

2.2. Interface Declaration

Definirea unei interfete se face prin intermediul cuvantului cheie interface:

```
[public] interface NumeInterfata
     [extends SuperInterfata1, SuperInterfata2...] {
     /* Corpul interfetei:
     Declaratii de constane
     Declaratii de metode abstracte
     */
}
```

O interfata poate avea un singur modificator si anume public. O interfata publica este accesibila tuturor claselor, indiferent de pachetul din care fac parte, implicit nivelul de acces fiind doar la nivelul pachetului din care face parte interfata.

O interfata poate extinde oricate interfete. Acestea se numesc superinterfete si sunt separate prin virgula.
Corpul unei interfete poate contine:

- **constante**: acestea pot fi sau nu declarate cu modificatorii public, static si final care sunt impliciti, nici un alt modificator neputand aparea in declaratia unei variabile dintr-o interfata. **Constantele unei interfete trebuie obligatoriu initializate**.

```
interface Exemplu {
     int MAX = 100;

     // Echivalent cu:
     public static final int MAX = 100;

     int MAX;
     // Incorect, lipseste initializarea

     private int x = 1;
     // Incorect, modificator nepermis
```

```
}
```

- **metode fara implementare**: acestea pot fi sau nu declarate cu modificatorul public, care este implicit; nici un alt modificator nu poate aparea in declaratia unei metode a unei interfete.

```
interface Exemplu {
     void metoda();

     // Echivalent cu:
     public void metoda();

     protected void metoda2();
     // Incorect, modificator nepermis

}
```

Nota:

- **Variabilele unei interfete** sunt **implicit publice** chiar daca nu sunt declarate cu modificatorul public.
- **Variabilele unei interfete** sunt **implicit constante** chiar daca nu sunt declarate cu modificatorii static si final.
- **Metodele unei interfete** sunt **implicit publice** chiar daca nu sunt declarate cu modificatorul public.

2.3. Interface Implementation

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface.
Note: **class implements interface** but an **interface extends another interface**.

```
interface MyInterface {
     public void method1();
     public void method2();
}

class XYZ implements MyInterface {
     public void method1() {
          System.out.println("implementation of method1");
     }
```

```
     public void method2() {
          System.out.println("implementation of method2");
     }

     public static void main(String arg[]) {
          MyInterface obj = new XYZ();
          obj.method1();
     }
}
```

Output:

```
implementation of method1
```

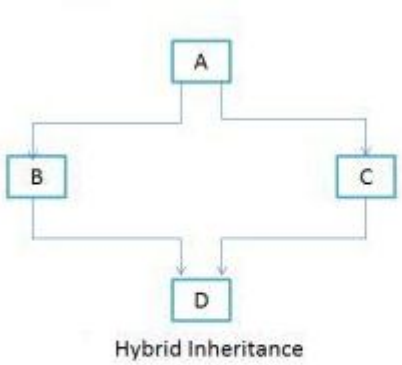
2.4. Interface and Inheritance

An interface can not implement another interface. It has to extend the other interface if required. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf1 and Inf2.

```
public interface Inf1 {
     public void method1();
}

public interface Inf2 extends Inf1 {
     public void method2();
}


public class Demo implements Inf2 {
     public void method1() {
     //Implementation of method1
     }

     public void method2(){
          //Implementation of method2
     }
}
```

“Demo” class is implementing only one interface “Inf2”. However it has to provide the implementation of all the methods of interface “Inf1” too, because interface Inf2 extends Inf1.

2.4. Hybrid Inheritance



Hybrid Inheritance

2.4.1. Using classes to form hybrid inheritance

```
public class A {
     public void methodA() {
          System.out.println("Class A methodA");
     }
}

public class B extends A {
     public void methodA() {
          System.out.println("Child class B is overriding
          inherited method A");
     }

     public void methodB() {
          System.out.println("Class B methodB");
     }
}

public class C extends A {
     public void methodA() {
          System.out.println("Child class C is overriding the
          methodA");
     }

     public void methodC() {
          System.out.println("Class C methodC");
     }
}
```

```
public class D extends B, C {
     public void methodD() {
          System.out.println("Class D methodD");
     }

     public static void main(String args[]) {
          D obj1= new D();
          obj1.methodD();
          obj1.methodA();
     }
}
```

Output:
Error!!

2.4.2. Using interfaces to form hybrid inheritance (correct)

```
interface A {
     public void methodA();
}

interface B extends A {
     public void methodB();
}

interface C extends A {
     public void methodC();
}

class D implements B, C {
     public void methodA() {
          System.out.println("MethodA");
     }

     public void methodB() {
          System.out.println("MethodB");
     }

     public void methodC() {
          System.out.println("MethodC");
     }

     public static void main(String args[]) {
          D obj1= new D();
          obj1.methodA();
          obj1.methodB();
          obj1.methodC();
     }
```

```
}
```

Output:
MethodA
MethodB
MethodC

Note: Even though class D didn't implement interface "A" still we have to define the methodA() in it. It is because interface B and C extends the interface A. The above code would work without any issues and that's how we implemented **hybrid inheritance** in java using interfaces.

3. Difference between Abstract Class and Interface

	abstract Classes	**Interfaces**
1	abstract class can extend only one class or one abstract class at a time	interface can extend any number of interfaces at a time
2	abstract class can extend from a class or from an abstract class	interface can extend only from an interface
3	abstract class can have both abstract and concrete methods	interface can have only abstract methods
4	A class can extend only one abstract class	A class can implement any number of interfaces
5	In abstract class keyword 'abstract' is mandatory to declare a method as an abstract	In an interface keyword 'abstract' is optional to declare a method as an abstract
6	Abstract class can have protected , public and public abstract methods	Interface can have only public abstract methods i.e. by default
7	abstract class can have static, final or static final variable with any access specifier	interface can have only static final (constant) variable i.e. by default

1. Nested Classes

Class within another class: it is called a **nested class**

O **clasa imbricata** este, prin definitie, o clasa membra a unei alte clase, numita si **clasa de acoperire**.

```
class OuterClass {
     ...

     class NestedClass {
          ...
     }
}
```

Terminology: Nested classes are divided into two categories: **static and non-static**. Nested classes that are declared **static** are called **static nested classes**. **Non-static** nested classes are called **inner classes**.

```
class OuterClass {
     ...

     static class StaticNestedClass {
          ...
     }

     class InnerClass {
          ...
     }
}
```

A **nested class** is a member of its **enclosing class**.

> **Non-static nested classes (inner classes)** have access to other members of the enclosing class, even if they are declared private.
>
> **Static nested classes** do not have access to other members of the enclosing class.
>
> As a member of the **outer class**, a nested class can be declared private, public, protected, or package private. Outer classes can only be declared public or package private.

1.1 Why Use Nested Classes?

Reasons for using nested classes include the following:

> **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
>
> **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
>
> **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

1.2 Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, **a static nested class cannot refer directly to instance variables or methods defined in its enclosing class**: it can use them only through an object reference.

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
             new OuterClass.StaticNestedClass();
```

1.3 Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist **within** an instance of the outer class.

Consider the following classes:

```
class OuterClass {
     ...

     class InnerClass {
          ...
     }
}
```

An instance of **InnerClass** can exist only **within an instance of OuterClass** and has direct access to the methods and fields of its enclosing instance.
To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject =
               outerObject.new InnerClass();
```

There are two **special kinds of inner classes**: **local classes** and **anonymous classes**

1.3.1 Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration **shadows** the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone.

```
public class ShadowTest {

     public int x = 0;

     class FirstLevel {

          public int x = 1;
```

```
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " +
                               ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

The output:

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

This example defines three variables named **x**: the member variable of the class **ShadowTest**, the member variable of the **inner class FirstLevel**, and the parameter in the method **methodInFirstLevel**. The variable **x** defined as a parameter of the method **methodInFirstLevel shadows** the variable of the inner class **FirstLevel**. Consequently, when you use the variable **x** in the method **methodInFirstLevel**, it refers to the method parameter. To refer to the member variable of the inner class **FirstLevel**, use the keyword **this** to represent the enclosing scope:

```
System.out.println("this.x = " + this.x);
```

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class **ShadowTest** from the method **methodInFirstLevel**:

```
System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
```

1.3.2 Local and Anonymous Classes

There are two additional types of inner classes. You can declare an **inner class within the body of a method**. These classes are known as **local classes**. You can also declare an **inner class within the body of a method without naming the class**. These classes are known as **anonymous classes**.

1.3.3 Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers **private**, **public**, and **protected** to restrict access to inner classes, just as you use them to restrict access do to other class members.

1.4 Local Classes

Local classes are classes that are defined in a **block**, which is **a group of zero or more statements between balanced braces**. You typically find local classes defined in the body of a method.

Topics:

- **Declaring Local Classes**
- **Accessing Members of an Enclosing Class**
    - **Shadowing and Local Classes**
- **Local Classes Are Similar To Inner Classes**

1.4.1 Declaring Local Classes

You can define a local class inside any block. For example, you can define a local class in a method body, a **for loop**, or an **if clause**.

The following example, LocalClassExample, validates two phone numbers. It defines the local class PhoneNumber in the method validatePhoneNumber:

```
public class LocalClassExample {

     static type memberOfEnclosingClass = ...;

     public static void method1(...) {

          final type localVariable =...;

          class LocalClass {

               ...

               LocalClass(...){        //constructor

                    ...
               }
```

```
            public ... oneMethodOfLocalClass(...) {
                 ...//return something (or not)
            }

            ...

        }

        LocalClass myObject1 = new LocalClass (...);
        LocalClass myObject2 = new LocalClass (...);


        ...
        //call oneMethodOfLocalClass
        myObject1.oneMethodOfLocalClass (...) ...
        ...

    }

    public static void main(String... args) {
        method1(...);
    }
}
```

1.4.2 Accessing Members of an Enclosing Class

A local class has access to the members of its enclosing class. In the previous example, **LocalClass (…)** constructor accesses the member **LocalClassExample.memberOfEnclosingClass**.
In addition, a **local class has access to local variables**. However, a local class can only access local variables that are **declared final**. When a local class accesses a local variable or parameter of the enclosing block, it **captures** that variable or parameter. For example, the **LocalClass (…)** constructor can access the local variable **localVariable** because it is declared final; **localVariable** is a **captured variable**.
However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are **final or effectively final**. A variable or parameter whose value is never changed after it is initialized is effectively final.

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters.

1.4.2.1 Shadowing and Local Classes

Declarations of a type (such as a variable) in a local class, shadow declarations in the enclosing scope that have the same name.

1.4.3 Local Classes Are Similar To Inner Classes

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods, such as the class **LocalClass**, which is defined in the static method **method1**, can only refer to static members of the enclosing class. For example, if you do not define the member variable **memberOfEnclosingClass** as **static**, then the **Java compiler generates an error** similar to "non-static variable **memberOfEnclosingClass** cannot be referenced from a static context."
Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

1.5 Anonymous Classes

Anonymous classes enable you to make your **code more concise**. They enable you to **declare and instantiate a class at the same time**. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

Topics:

- **Declaring Anonymous Classes**
- **Syntax of Anonymous Classes**
- **Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

1.5.1 Declaring Anonymous Classes

While local classes are class declarations, **anonymous classes are expressions**, meaning that you **define the class in another expression**.

The following example, **HelloWorldAnonymousClasses**, uses anonymous classes in the initialization statements of the local variables **frenchGreeting** and **spanishGreeting**, but uses a local class for the initialization of the variable **englishGreeting**:

```
public class HelloWorldAnonymousClasses {

     interface HelloWorld {
          public void greet();
          public void greetSomeone(String someone);
     }

     public void sayHello() {

          class EnglishGreeting implements HelloWorld {
               String name = "world";
               public void greet() {
                    greetSomeone("world");
               }
               public void greetSomeone(String someone) {
                    name = someone;
                    System.out.println("Hello " + name);
               }
          }

          HelloWorld englishGreeting = new EnglishGreeting();

          HelloWorld frenchGreeting = new HelloWorld() {
               String name = "tout le monde";
               public void greet() {
                    greetSomeone("tout le monde");
               }
               public void greetSomeone(String someone) {
                    name = someone;
                    System.out.println("Salut " + name);
               }
          };

          HelloWorld spanishGreeting = new HelloWorld() {
               String name = "mundo";
               public void greet() {
                    greetSomeone("mundo");
               }
               public void greetSomeone(String someone) {
                    name = someone;
                    System.out.println("Hola, " + name);
               }
          };
          englishGreeting.greet();
```

```
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
                    new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

1.5.2 Syntax of Anonymous Classes

An anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the **frenchGreeting** object:

```
HelloWorld frenchGreeting = new HelloWorld() {
    String name = "tout le monde";

    public void greet() {
        greetSomeone("tout le monde");
    }

    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Salut " + name);
    }
};
```

The anonymous class expression consists of the following:

- The **new operator**

- The **name of an interface to implement or a class to extend**. In this example, the anonymous class is implementing the interface **HelloWorld**.

- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.

A body, which is a **class declaration body**. More specifically, in the body, **method declarations are allowed but statements are not**.

Because an anonymous class definition is an expression, **it must be part of a statement**. In this example, the anonymous class expression is part of the statement that instantiates the **frenchGreeting** object. (This explains why there is a semicolon after the closing brace.)

1.5.3 Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class

Like local classes, **anonymous classes can capture variables**; they have the same access to local variables of the enclosing scope:

- An anonymous class has **access to the members of its enclosing class**.
- An anonymous class **cannot access local variables** in its enclosing scope that are not declared as **final or effectively final**.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You **cannot declare static initializers or member interfaces** in an anonymous class.
- An anonymous class **can have static members** provided that they are constant variables.

Note that you can declare the following in anonymous classes:

- **Fields**
- **Extra methods** (even if they do not implement any methods of the supertype)
- **Instance initializers**
- **Local classes**

However, you cannot declare constructors in an anonymous class.

1.6 Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

1.7 When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

Nested classes enable you to logically group classes that are only used in one place, increase the use of encapsulation, and create more readable and maintainable code. Local classes, anonymous classes, and lambda expressions also impart these advantages; however, they are intended to be used for more specific situations:

- **Local class**: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).
- **Anonymous class**: Use it if you need to declare fields or additional methods.
- **Lambda expression**:
    - Use it if you are encapsulating a single unit of behavior that you want to pass to other code.
    - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).
- **Nested class**: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.
    - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

2. Enum Types

Incepand cu versiunea 1.5 a limbajului Java, exista posibilitatea de a defini **tipuri de date enumerare** prin folosirea cuvantului cheie **enum**. Acesta solutie simplifica manevrarea grupurilor de constante, dupa cum reiese din urmatorul exemplu:

```
public class CuloriSemafor {
public static final int ROSU = -1;
public static final int GALBEN = 0;
public static final int VERDE = 1;
}
...
// Exemplu de utilizare
if (semafor.culoare = CuloriSemafor.ROSU)
semafor.culoare = CuloriSemafor.GALBEN);
...
```

Clasa de mai sus poate fi rescrisa astfel:

```
public enum CuloriSemafor {ROSU, GALBEN, VERDE};
...
// Utilizarea structurii se face la fel
if (semafor.culoare = CuloriSemafor.ROSU)
     semafor.culoare = CuloriSemafor.GALBEN);
...
```

Compilatorul este responsabil cu transformarea unei astfel de structuri intr-o clasa corespunzatoare.

An **enum type** is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in **uppercase letters**.

In the Java programming language, you define an **enum type** by using the **enum keyword**. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
     THURSDAY, FRIDAY, SATURDAY
}
```

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Here is some code that shows you how to use the **Day enum** defined above:

```
public class EnumTest {
     Day day;

     public EnumTest(Day day) {
          this.day = day;
     }

     public void tellItLikeItIs() {
          switch (day) {
               case MONDAY:
                    System.out.println("Mondays are bad.");
                    break;

               case FRIDAY:
                    System.out.println("Fridays are better.");
                    break;

               case SATURDAY: case SUNDAY:
                    System.out.println("Weekends are best.");
                    break;

               default:
                    System.out.println("Midweek days are
                                        so-so.");
                    break;
          }
     }

     public static void main(String[] args) {
          EnumTest firstDay = new EnumTest(Day.MONDAY);
          firstDay.tellItLikeItIs();
          EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
          thirdDay.tellItLikeItIs();
          EnumTest fifthDay = new EnumTest(Day.FRIDAY);
          fifthDay.tellItLikeItIs();
          EnumTest sixthDay = new EnumTest(Day.SATURDAY);
          sixthDay.tellItLikeItIs();
```

```
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

The output is:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

Java programming language enum types are much more powerful than their counterparts in other languages. The **enum declaration defines a class** (**called an enum type**). The **enum class body can include methods and other fields**. The compiler automatically **adds some special methods** when it creates an enum. For example, they have a **static values method that returns an array containing all of the values of the enum in the order they are declared**. This method is commonly used in combination with the **for-each construct to iterate over the values** of an enum type. For example, this code from the **Planet** class example below iterates over all the planets in the solar system.

```
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f%n",
                      p, p.surfaceWeight(mass));
}
```

All enums implicitly extend java.lang.Enum

In the following example, **Planet** is an enum type that represents the planets in the solar system. They are defined with constant mass and radius properties.
Each enum constant is declared with values for the mass and radius parameters. These values are **passed to the constructor** when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of **enum constants must end with a semicolon**.

Note: The **constructor** for an enum type must be **package-private or private access**. It automatically creates the constants that are defined at the beginning of the enum body. **You cannot invoke an enum constructor yourself**.

In addition to its **properties and constructor**, **Planet** has **methods** that allow you to retrieve the surface gravity and weight of an object on each planet. Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public enum Planet {
     MERCURY (3.303e+23, 2.4397e6),
     VENUS   (4.869e+24, 6.0518e6),
     EARTH   (5.976e+24, 6.37814e6),
     MARS    (6.421e+23, 3.3972e6),
     JUPITER (1.9e+27,   7.1492e7),
     SATURN  (5.688e+26, 6.0268e7),
     URANUS  (8.686e+25, 2.5559e7),
     NEPTUNE (1.024e+26, 2.4746e7);

     private final double mass;   // in kilograms
     private final double radius; // in meters

     Planet(double mass, double radius) {
          this.mass = mass;
          this.radius = radius;
     }

     private double mass() { return mass; }
     private double radius() { return radius; }

     // universal gravitational constant (m3 kg-1 s-2)
     public static final double G = 6.67300E-11;

     double surfaceGravity() {
          return G * mass / (radius * radius);
     }
     double surfaceWeight(double otherMass) {
          return otherMass * surfaceGravity();
     }
     public static void main(String[] args) {
          if (args.length != 1) {
               System.err.println("Usage: java Planet
                                        <earth_weight>");
               System.exit(-1);
          }

          double earthWeight = Double.parseDouble(args[0]);
          double mass = earthWeight/EARTH.surfaceGravity();

          for (Planet p : Planet.values())
               System.out.printf("Your weight on %s is %f%n",
                                 p, p.surfaceWeight(mass));
     }
}
```

If you run **Planet.class** from the command line with an argument of 175, you get this

output:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

1. Threads

1.1 Processes and Threads (concurrent programming)

There are two basic units of execution: **processes** and **threads**

Firele de executie fac trecerea de la programarea secventiala la programarea concurenta. Un program secvential reprezinta modelul clasic de program: are un inceput, o secventa de executie a instructiunilor sale si un sfarsit (la un moment dat programul are un singur punct de executie). Un program aflat in executie se numeste **proces**.

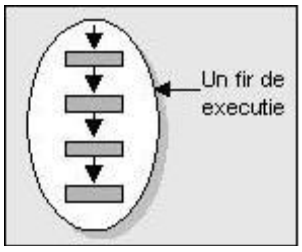
Un sistem de operare monotasking, cum ar fi fi MS-DOS, nu este capabil sa execute decat un singur process la un moment dat. Un sistem de operare multitasking, cum ar fi UNIX sau Windows, poate rula oricate procese in acelasi timp (concurent), folosind diverse strategii de alocare a procesorului fiecaruia dintre acestea.
Notiunea de fir de executie nu are sens decat in cadrul unui sistem de operare multitasking.

Un fir de executie este similar unui proces secvential, in sensul ca are un inceput, o secventa de executie si un sfarsit. Diferenta dintre un fir de executie si un proces consta in faptul ca un fir de executie nu poate rula independent ci trebuie sa ruleze in cadrul unui proces.

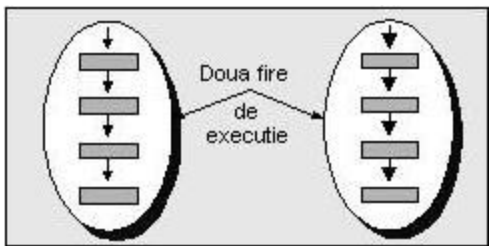
Definitie

Un fir de executie este o succesiune secventiala de instructiuni care se executa in cadrul unui proces.

Program (proces)



Program (proces)



Un program isi poate defini insa nu doar un fir de executie ci oricate, ceea ce inseamna ca in cadrul unui proces se pot executa simultan mai multe fire de executie, permitand efectuarea concurenta a sarcinilor independente ale acelui program.

Un fir de executie poate fi asemanat cu o versiune redusa a unui proces, ambele ruland simultan si independent pe o structura secventiala formata de instructiunile lor.

De asemenea, executia simultana a firelor in cadrul unui proces este similara cu executia concurenta a proceselor: sistemul de operare va aloca procesorul dupa o anumita strategie fiecarui fir de executie pana la terminarea lor. Din acest motiv firele de executie mai sunt numite si procese usoare (**lightweight processes**).

Care ar fi insa deosebirile intre un fir de executie si un proces? In primul, rand deosebirea majora consta in faptul ca firele de executie nu pot rula decat in cadrul unui proces. O alta deosebire rezulta din faptul ca fiecare proces are propria sa memorie (propriul sau spatiu de adrese) iar la crearea unui nou proces este realizata o copie exacta a procesului parinte: cod si date, in timp ce la crearea unui fir nu este copiat decat codul procesului parinte, toate firele de executie avand acces la aceleasi date, datele procesului original.

Asadar, un fir mai poate fi privit si ca un **context de executie in cadrul unui proces**.

Firele de executie sunt utile in multe privinte, insa uzual ele sunt folosite pentru executarea unor operatii consumatoare de timp fara a bloca procesul principal: calcule matematice, asteptarea eliberarii unei resurse, desenarea componentelor unei aplicatii GUI, etc. De multe ori firele isi desfasoara activitatea in fundal insa, evident, acest lucru nu este obligatoriu.

1.2. Crearea unui fir de executie

Crearea unei clase care sa defineasca fire de executie poate fi facuta prin doua modalitati:

- prin extinderea clasei **Thread**.
- prin implementarea interfetei **Runnable**.

Orice clasa ale carei instante vor fi executate separat intr-un fir propriu trebuie declarata ca fiind de tip **Runnable**. Aceasta este o interfata care contine o singura metoda si anume metoda **run**. Asadar, orice clasa ce descrie fire de executie va contine metoda **run** in care este implementat codul ce va fi rulat. Interfata **Runnable** este conceputa ca fiind un protocol comun pentru obiectele care doresc sa execute un anumit cod pe durata existentei lor.

Cea mai importanta clasa care implementeaza interfata **Runnable** este **Thread**. Aceasta implementeaza un fir de executie generic care, implicit, nu face nimic; cu alte cuvinte, metoda run nu contine nici un cod. Orice fir de executie este o instanta a clasei **Thread** sau a unei subclase a sa.

1.2.1 Extinderea clasei **Thread**

Cea mai simpla metoda de a crea un fir de executie care sa realizeze o anumita actiune este prin extinderea clasei **Thread** si supradefinirea metodei **run** a acesteia.

Formatul general al unei astfel de clase este:

```
public class FirExecutie extends Thread {
     public FirExecutie(String nume) {
          // Apelam constructorul superclasei
          super(nume);
     }

     public void run() {
          // Codul firului de executie
          ...
     }
}
```

Prima metoda a clasei este **constructorul**, care primeste ca argument un sir ce va reprezenta numele firului de executie. In cazul in care nu vrem sa dam nume firelor pe care le cream, atunci putem renunta la supradefinirea acestui constructor si sa folosim constructorul implicit, fara argumente, care creeaza un fir de executie fara nici un nume. Ulterior, acesta poate primi un nume cu metoda **setName**. Evident, se pot defini si alti constructori, acestia fiind utili atunci cand vrem sa trimitem diversi parametri de initializare firului nostru.

A doua metoda este metoda **run**, ”inima” oricarui fir de executie, in care scriem efectiv codul care trebuie sa se execute.

Un fir de executie creat nu este automat pornit, lansarea sa fiind realizeaza de metoda **start**, definita in clasa **Thread**.

```
// Cream firul de executie
FirExecutie fir = new FirExecutie("simplu");
// Lansam in executie
fir.start();
```

1.2.2 Implementarea interfetei **Runnable**

Ce facem insa cand dorim sa cream o clasa care instantiaza fire de executie dar aceasta are deja o superclasa (in Java nu este permisa mostenirea multipla) ?

```
class FirExecutie extends Parinte, Thread // incorect !
```

In acest caz, nu mai putem extinde clasa **Thread** ci trebuie sa implementam direct interfata **Runnable**. Clasa **Thread** implementeaza ea insasi interfata **Runnable** si, din acest motiv, la extinderea ei obtineam implementarea indirect a a interfetei. Asadar, interfata **Runnable** permite unei clase sa fie **activa**, fara a extinde clasa **Thread**.

Interfata **Runnable** se gaseste in pachetul java.lang si este definita astfel:

```
public interface Runnable {
     public abstract void run();
}
```

Prin urmare, o clasa care instantiaza fire de executie prin implementarea interfetei Runnable trebuie obligatoriu sa implementeze metoda run. O astfel de clasa se mai numeste **clasa activa** si are urmatoarea structura:

```
public class ClasaActiva implements Runnable {
     public void run() {
          //Codul firului de executie
          ...
     }
}
```

Spre deosebire de modalitatea anterioara, se pierde insa tot suportul oferit de clasa **Thread**.
Simpla instantiere a unei clase care implemeneaza interfata **Runnable** nu creeaza nici un fir de executie, **crearea acestora trebuind facuta explicit**. Pentru a realiza acest lucru trebuie sa instantiem un obiect de tip **Thread** ce va reprezenta firul de executie propriu zis al carui cod se gaseste in clasa noastra. Acest lucru se realizeaza, ca pentru orice alt obiect, prin instructiunea **new**, urmata de un apel la un constructor al clasei Thread, insa nu la oricare dintre acestia.
Trebuie apelat constructorul care sa primeasca drept argument o instanta a clasei noastre. Dupa creare, firul de executie poate fi lansat printr-un apel al metodei **start**.

```
ClasaActiva obiectActiv = new ClasaActiva();
Thread fir = new Thread(obiectActiv);
fir.start();
```

Aceste operatiuni pot fi facute chiar in cadrul clasei noastre:

```
public class FirExecutie implements Runnable {

     private Thread fir = null;

     public FirExecutie() {
          if (fir == null) {
               fir = new Thread(this);
               fir.start();
          }
     }

     public void run() {
          //Codul firului de executie
          ...
     }
}
```

Specificarea argumentului this in constructorul clasei **Thread** determina crearea unui fir de executie care, la lansarea sa, va apela metoda run din clasa curenta. Asadar, acest constructor accepta ca argument orice instanta a unei clase **Runnable**. Pentru clasa FirExecutie data mai sus, lansarea firului va fi facuta automat la instantierea unui obiect al clasei:

```
FirExecutie fir = new FirExecutie();
```

Nota: Metoda **run** nu trebuie apelata explicit, acest lucru realizandu-se automat la apelul metodei **start**. Apelul explicit al metodei **run** nu va furniza nici o eroare, insa aceasta va fi executata ca orice alta metoda, si nu separat intr-un fir.

1.3 Thread Methods

Following is the list of important methods available in the Thread class.

Sr.No.	Method & Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

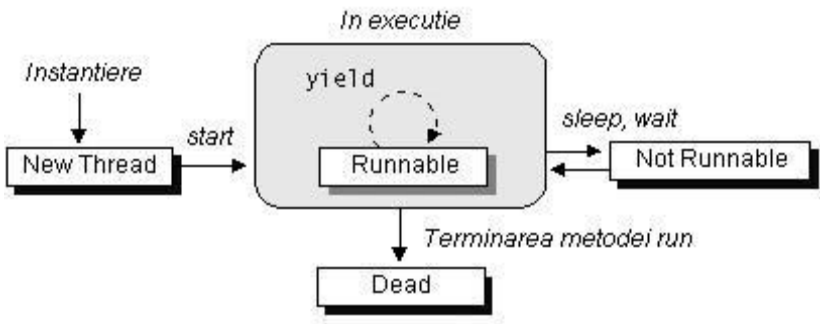
The previous methods are invoked on a particular **Thread object**. The following methods in the Thread class are **static**. Invoking one of the static methods performs the operation on the currently running thread.

Sr.No.	Method & Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.

5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

1.4. Ciclul de viata

Fiecare fir de executie are propriul sau ciclu de viata: este creat, devine active prin lansarea sa si, la un moment dat, se termina. In continuare, vom analiza mai indeaproape starile in care se poate gasi un fir de executie. Diagrama de mai jos ilustreaza generic aceste stari precum si metodele care provoaca tranzitia dintr-o stare in alta:



Un fir de executie se poate gasi in una din urmatoarele patru **stari**:

New Thread
Runnable
Not Runnable
Dead

New Thread

Un fir de executie se gaseste in aceasta stare imediat dupa crearea sa, cu alte cuvinte dupa instantierea unui obiect din clasa **Thread** sau dintr-o subclasa a sa.

```
Thread fir = new Thread(obiectActiv);
// fir se gaseste in starea "New Thread"
```

In aceasta stare firul este ”vid”, el nu are alocate nici un fel de resurse sistem si singura operatiune pe care o putem executa asupra lui este lansarea in executie, prin metoda **start**. Apelul oricarei alte metode in afara de **start** nu are nici un sens si va provoca o exceptie de tipul **IllegalThreadStateException**.

Runnable

Dupa apelul metodei start un fir va trece in starea **Runnable**, adica va fi **in executie**.

```
fir.start();
//fir se gaseste in starea "Runnable"
```

Metoda start realizeaza urmatoarele operatiuni necesare rularii firului de executie:

- Aloca resursele sistem necesare.
- Planifica firul de executie la procesor pentru a fi lansat.
- Apeleaza metoda run a obiectului activ al firului.

Un fir aflat in starea **Runnable** nu inseamna neaparat ca se gaseste efectiv in executie, adica instructiunile sale sunt interpretate de procesor.

Acest lucru se intampla din cauza ca majoritatea calculatoarelor au un singur procesor iar acesta nu poate rula simultan toate firele de executie care se gasesc in starea **Runnable**.

Pentru a rezolva aceasta problema exista o planificare care sa partajeze dinamic si corect procesorul intre toate firele de executie care sunt in starea **Runnable**. Asadar, un fir care ”ruleaza” poate sa-si astepte de fapt randul la procesor.

Not Runnable

Un fir de executie poate ajunge in aceasta stare in una din urmatoarele situatii:

- Este ”adormit” prin apelul metodei **sleep**;
- A apelat metoda **wait**, asteptand ca o anumita conditie sa fie satisfacuta;
- Este blocat intr-o operatie de intrare / iesire.

Metoda **sleep** este o metoda statica a clasei Thread care provoaca o pauza in timpul rularii firului curent aflat in executie, cu alte cuvinte il ”adoarme” pentru un timp specificat. Lungimea acestei pauze este specificata in **milisecunde** si chiar **nanosecunde**. Intrucat poate provoca exceptii de tipul **InterruptedException**, apelul acestei metode se face intr-un bloc de tip

```
try-cacth:

try {
     // Facem pauza de o secunda
     Thread.sleep(1000);
```

```
} catch (InterruptedException e) {
     ...
}
```

Observati ca **metoda** fiind **statica** apelul ei nu se face pentru o instanta anume a clasei **Thread**. Acest lucru este foarte normal deoarece, la un moment dat, un singur fir este in executie si doar pentru acesta are sens ”adormirea” sa.
In intervalul in care un fir de executie ”doarme”, acesta nu va fi executat chiar daca procesorul devine disponibil.
Dupa expirarea intervalului specificat firul revine in starea **Runnable** iar daca procesorul este in continuare disponibil isi va continua executia.
Pentru fiecare tip de intrare in starea **Not Runnable**, exista o secventa specifica de iesire din starea repectiva, care readuce firul de executie in starea **Runnable**. Acestea sunt:

- Daca un fir de executie a fost ”adormit”, atunci el devine **Runnable** doar dupa scurgerea intervalului de timp specificat de instructiunea **sleep**.
- Daca un fir de executie asteapta o anumita conditie, atunci un alt obiect trebuie sa il informeze daca acea conditie este indeplinita sau nu; acest lucru se realizeaza prin instructiunile **notify** sau **notifyAll**
- Daca un fir de executie este blocat intr-o operatiune de intrare / iesire atunci el redevine **Runnable** atunci cand acea operatiune s-a terminat.

Dead

Este starea in care ajunge un fir de executie la terminarea sa. Un fir nu poate fi oprit din program printr-o anumita metoda, ci trebuie sa se termine in mod natural la **incheierea metodei run** pe care o executa. Spre deosebire de versiunile curente ale limbajului Java, in versiunile mai vechi exista metoda stop a clasei **Thread** care termina fortat un fir de executie, insa aceasta a fost eliminata din motive de securitate. Asadar, un fir de executie trebuie sa-si cauzeze singur propria sa ”moarte”.

1.4.1 Terminarea unui fir de executie

Dupa cum am vazut, un fir de executie nu poate fi terminat fortat de catre program ci trebuie sa-si cauzeze singur terminarea sa. Acest lucru poate fi realizat in doua modalitati:

1. Prin scrierea unor metode run care sa-si **termine executia in mod natural**. La terminarea metodei **run** se va termina automat si firul de executie, acesta intrand in starea **Dead**.

```
// Primul exemplu
public void run() {
     for(int i = a; i <= b; i += pas)
          System.out.print(i + " ");
}
```

Dupa afisarea numerelor din intervalul specificat, metoda se termina si, odata cu ea, se va termina si firul de executie repsectiv.

2. Prin folosirea unei **variabile de terminare**. In cazul cand metoda **run** trebuie sa execute o bucla infinita atunci aceasta trebuie controlata printr-o variabila care sa opreasca ciclul atunci cand dorim ca firul de executie sa se termine. Uzual, vom folosi o variabila membra a clasei care descrie firul de executie, variabila care este fie publica, fie este asociata cu o metoda publica care permite schimbarea valorii sale.

```
import java .io .*;

class NumaraSecunde extends Thread {
     public int sec = 0;

     // Folosim o variabila de terminare
     public boolean executie = true;

     public void run () {
          while (executie) {
               try {
                    Thread.sleep (1000);
                    sec ++;
                    System.out.print (".");
               } catch (InterruptedException e){}
          }
     }
}

public class TestTerminare {
     public static void main (String args [])
                                        throws IOException {
          NumaraSecunde fir = new NumaraSecunde ();
          fir.start ();
          System.out.println (" Apasati tasta Enter ");
          System.in.read ();

          // Oprim firul de executie
          fir.executie = false;
          System.outprintln ("S-au scurs " + fir.sec +
                              " secunde ");
     }
}
```

Nu este necesara distrugerea explicita a unui fir de executie. Sistemul Java de colectare a ”gunoiului” se ocupa de acest lucru. Setarea valorii **null** pentru variabila care referea instanta firului de executie va usura insa activitatea procesului **gc**.

Pentru a testa daca un fir de executie a fost pornit dar nu s-a terminat inca putem folosi **metoda isAlive**. Metoda returneaza:

true - daca firul este in una din starile **Runnable** sau **Not Runnable**
false - daca firul este in una din starile **New Thread** sau **Dead**

Intre starile **Runnable** sau **Not Runnable**, respectiv **New Thread**, sau **Dead** nu se poate face nici o diferentiere.

```
NumaraSecunde fir = new NumaraSecunde();
// isAlive retuneaza false (starea este New Thread)

fir.start();
// isAlive retuneaza true (starea este Runnable)

fir.executie = false;
// isAlive retuneaza false (starea este Dead)
```

1.4.2 Fire de executie de tip ”daemon”

Un proces este considerat in executie daca contine cel putin un fir de executie activ. Cu alte cuvinte, la rularea unei aplicatii, masina virtuala Java nu se va opri decat atunci cand nu mai exista nici un fir de executie activ. De multe ori insa dorim sa folosim fire care sa realizeze diverse activitati, eventual periodic, pe toata durata de executie a programului iar in momentul terminarii acestuia sa se termine automat si firele respective. Aceste fire de executie se numesc **demoni**.
Dupa crearea sa, un fir de executie poate fi facut demon, sau scos din aceasta stare, cu **metoda setDaemon**.

1.4.3 Stabilirea prioritatilor de executie

Majoritatea calculatoarelor au un sigur procesor, ceea ce inseamna ca firele de executie trebuie sa-si imparta accesul la acel procesor. Executia intr-o anumita ordine

a mai multor fire de executie pe un numar limitat de procesoare se numeste **planificare (scheduling)**.

Sistemul Java de executie a programelor implementeaza un algoritm simplu, determinist de planificare, cunoscut sub numele de planificare cu prioritati fixate.

Fiecare fir de executie Java primeste la crearea sa o anumita prioritate.

O prioritate este de fapt un numar intreg cu valori cuprinse intre MIN PRIORITY si MAX PRIORITY. Implicit, prioritatea unui fir nou creat are valoarea NORM PRIORITY. Aceste trei constante sunt definite in clasa Thread astfel:

```
public static final int MAX_PRIORITY = 10;
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY= 5;
```

Schimbarea ulterioara a prioritatii unui fir de executie se realizeaza cu **metoda setPriority** a clasei Thread.

La nivelul sistemului de operare, exista doua modele de lucru cu fire de executie:

> **Modelul cooperativ**, in care firele de executie decid cand sa cedeze procesorul; dezavantajul acestui model este ca unele fire pot acapara procesorul, nepermitand si executia altora pana la terminarea lor.
> **Modelul preemptiv**, in care firele de executie pot fi intrerupte oricand, dupa ce au fost lasate sa ruleze o perioada, urmand sa fie reluate dupa ce si celelalte fire aflate in executie au avut acces la procesor; acest sistem se mai numeste cu ”cuante de timp”, dezavantajul sau fiind nevoia de a sincroniza accesul firelor la resursele comune.

Asadar, in modelul cooperativ firele de executie sunt responsabile cu partajarea timpului de executie, in timp ce in modelul preemptiv ele trebuie sa partajeze resursele comune.
Deoarece specificatiile masinii virtuale Java nu impun folosirea unui anumit model, programele Java trebuie scrise astfel incat sa functioneze corect pe ambele modele.

Planificatorul Java lucreaza in modul urmator: daca la un moment dat sunt mai multe fire de executie in starea **Runnable**, adica sunt pregatite pentru a fi rulate, planificatorul il va alege pe cel cu prioritatea cea mai mare pentru a-l executa.
Doar cand firul de executie cu prioritate maxima se termina, sau este suspendat din diverse motive, va fi ales un fir cu o prioritate mai mica. In cazul in care toate firele au aceeasi prioritate ele sunt alese pe rand, dupa un algoritm simplu de tip ”round-robin”.
De asemenea, daca un fir cu prioritate mai mare decat firul care se executa la un moment dat solicita procesorul, atunci firul cu prioritate mai mare este imediat trecut in executie iar celalalt trecut in asteptare. Planificatorul Java nu va intrerupe insa un fir de executie in favoarea altuia de aceeasi prioritate, insa acest lucru il poate face sistemul de operare in cazul in care acesta aloca procesorul in cuante de timp (un astfel de SO este Windows).

Un fir de executie Java cedeaza procesorul in una din situatiile:

- un fir de executie cu o **prioritate mai mare** solicita procesorul;
- **metoda** sa **run** se termina;
- face explicit acest lucru apeland **metoda yield**;
- **timpul alocat** pentru executia sa a **expirat** (pe SO cu cuante de timp).

Un fir de executie de lunga durata si care nu cedeaza explicit procesorul la anumite intervale de timp astfel incat sa poata fi executate si celelalte fire de executie se numeste **fir de executie egoist**.
Evident, trebuie evitata scrierea lor intrucat acapareaza pe termen nedefinit procesorul, blocand efectiv executia celorlalte fire de executie pana la terminarea sa.
Unele sistemele de operare combat acest tip de comportament prin metoda alocarii procesorului in cuante de timp fiecarui fir de executie, insa nu trebuie sa ne bazam pe acest lucru la scrierea unui program.
Un fir de executie trebuie sa fie ”corect” fata de celelalte fire si sa cedeze periodic procesorul astfel incat toate sa aiba posibilitatea de a se executa.

1.5 Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: **thread interference** and **memory consistency errors**. The tool needed to prevent these errors is **synchronization**.

1.5.1 Thread Interference

Consider a simple class called Counter

```
class Counter {
     private int c = 0;

     public void increment() {
          c++;
     }

     public void decrement() {
          c--;
     }

     public int value() {
          return c;
     }

}
```

Counter is designed so that each invocation of **increment will add 1 to c**, and each invocation of **decrement will subtract 1 from c**. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.
Interference happens when two operations, running in different threads, but acting on the same data, **interleave**. This means that the two operations consist of **multiple steps**, and the **sequences of steps overlap**.
It might not seem possible for operations on **instances of Counter to interleave**, since both operations on c are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single **expression c++ can be decomposed into three steps**:

> Retrieve the current value of c.
> Increment the retrieved value by 1.
> Store the incremented value back in c.

The expression c-- can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread **A invokes increment** at about the same time **Thread B invokes decrement**. If the initial value of c is 0, their interleaved actions might follow this sequence:

> Thread A: Retrieve c.
> Thread B: Retrieve c.
> Thread A: Increment retrieved value; result is 1.
> Thread B: Decrement retrieved value; result is -1.
> Thread A: Store result in c; c is now 1.
> Thread B: Store result in c; c is now -1.
>
> **Thread A's result is lost**, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are **unpredictable**, thread interference bugs can be difficult to detect and fix.

1.5.2 Synchronized Methods

The Java programming language provides two basic **synchronization techniques**: **synchronized methods** and **synchronized statements**.
To make a method synchronized, simply add the **synchronized keyword** to its declaration:

```
public class SynchronizedCounter {
     private int c = 0;

     public synchronized void increment() {
          c++;
     }

     public synchronized void decrement() {
          c--;
     }

     public synchronized int value() {
          return c;
     }
}
```

If **count** is an instance of **SynchronizedCounter**, then making these methods synchronized has two effects:
First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
Second, when a synchronized method exits, it automatically establishes a **happens-before relationship** with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.
Note that constructors cannot be synchronized — using the **synchronized keyword** with a constructor is a **syntax error**. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, **all reads or writes to that object's variables are done through synchronized methods**. (An important exception: **final fields**, which cannot be modified after the object is constructed, **can be safely read through non-synchronized methods**, once the object is constructed)

1.5.2 Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock**. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are

essential to visibility.
Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to **acquire the object's intrinsic lock** before accessing them, and then **release the intrinsic lock** when it's done with them. A thread is said to **own the intrinsic lock** between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a **happens-before relationship is established** between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an **uncaught exception**.
You might wonder what happens when a **static synchronized method is invoked**, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with **synchronized statements**. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
     synchronized(this) {
          lastName = name;
          nameCount++;
     }
     nameList.add(name);
}
```

Monitoare fine (**fine-grained synchronization)**

Adeseori, folosirea unui monitor pentru intreg obiectul poate fi prea restrictiva. De ce sa blocam toate resursele unui obiect daca un fir de executie nu doreste decat accesarea uneia sau a catorva dintre ele ? Deoarece orice obiect are un monitor, putem folosi obiecte fictive ca lacate pentru fiecare din resursele obiectului nostru, ca in exemplul de mai jos:

```
class MonitoareFine {
     //Cele doua resurse ale obiectului
     Resursa x, y;

     //Folosim monitoarele a doua obiecte fictive
     Object xLacat = new Object();
     Object yLacat = new Object();

     public void metoda() {
          synchronized(xLacat) {
               // Accesam resursa x
          }

          // Cod care nu foloseste resursele comune
          ...

          synchronized(yLacat) {
               // Accesam resursa y
          }
          ...

          synchronized(xLacat) {
               synchronized(yLacat) {
                    // Accesam x si y
               }
          }
          ...

          synchronized(this) {
               // Accesam x si y
          }
     }
}
```

Metoda de mai sus nu a fost declarata cu synchronized ceea ce ar fi determinat blocarea tuturor resurselor comune la accesarea obiectului respective de un fir de executie, ci au fost folosite monitoarele unor obiecte fictive pentru a controla folosirea fiecarei resursa in parte.

1. Exceptions (Topics)

The Java programming language uses exceptions to handle errors and other exceptional events.

What Is an Exception?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

The Catch or Specify Requirement

- How to **catch and handle exceptions**. (**try, catch, and finally blocks**, as well as **chained exceptions**).

How to Throw Exceptions

- The **throw statement** and the **Throwable class** and its subclasses.

The try-with-resources Statement

- The **try-with-resources** statement, which is a **try statement that declares one or more resources**. A resource is as an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement.

Unchecked Exceptions — The Controversy

- The correct and incorrect use of the unchecked exceptions indicated by subclasses of **RuntimeException**.

Advantages of Exceptions

The use of exceptions to manage errors has some advantages over traditional error-management techniques.

1.1 What Is an Exception?

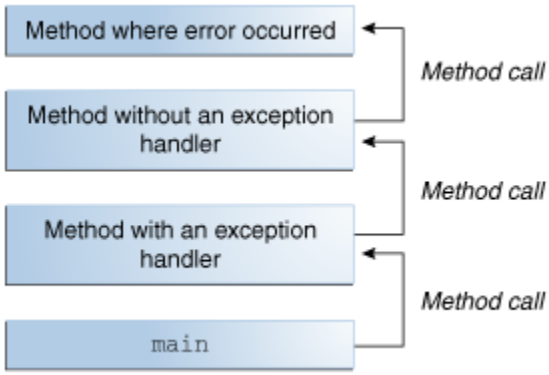
The term **exception** is shorthand for the phrase "exceptional event."

Definition: An exception is an event, which occurs during the execution of a program - that disrupts the normal flow of the program's instructions.

1.1.1. Throw the exception (aruncarea unei exceptii)

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. **Creating an exception object and handing it** to the runtime system is called **throwing an exception**.

After a method throws an exception, the runtime system attempts to find **something to handle it**. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the **call stack**.



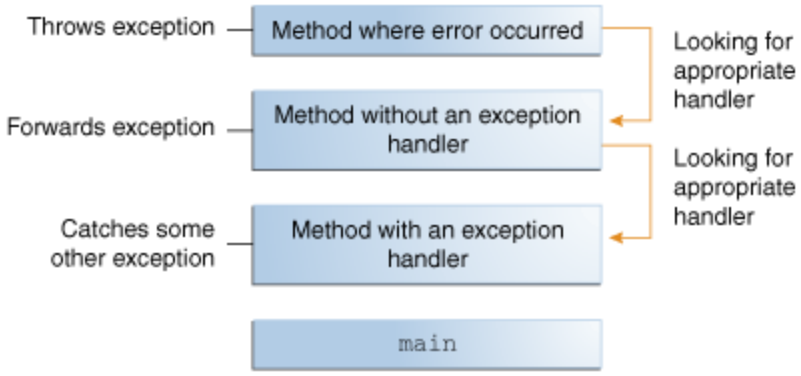
The call stack.

1.1.3 Exception handler (analizor de exceptie / al exceptiei)

The runtime system **searches** the call stack **for a method that contains a block of code that can handle the exception**. This block of code is called an **exception handler**. The search begins with the method in which the error occurred and **proceeds through the call stack in the reverse order in which the methods were called**. When an appropriate handler is found, the runtime system passes the exception to the handler. An **exception handler** is considered **appropriate** if the type of the exception object thrown matches the type that can be handled by the handler.

1.1.2 Catch the exception (prinderea exceptiei)

The exception handler chosen is said to **catch the exception**. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler.

Using exceptions to manage errors has some advantages over traditional error-management techniques.

Nota:
In Java tratarea erorilor nu mai este o optiune ci o constrangere. In aproape toate situatiile, o secventa de cod care poate provoca exceptii trebuie sa specifice modalitatea de tratare a acestora.

2. Exceptii (continuare)

Termenul exceptie este o prescurtare pentru ”eveniment exceptional” si poate fi definit ca un eveniment ce se produce in timpul executiei unui program si care provoaca intreruperea cursului normal al executiei acestuia.

Exceptiile pot aparea din diverse cauze si pot avea niveluri diferite de gravitate: de la erori fatale cauzate de echipamentul hardware pana la erori ce tin strict de codul programului, cum ar fi accesarea unui element din afara spatiului alocat unui vector.

In momentul cand o asemenea eroare se produce in timpul executiei va fi generat un obiect de tip exceptie ce contine:

- informatii despre exceptia respectiva;
- starea programului in momentul producerii acelei exceptii.

```
public class Exceptii {
     public static void main(String args[]) {
          int v[] = new int[10];
          v[10] = 0; //Exceptie !
          System.out.println("Aici nu se mai ajunge...");
     }
}
```

La rularea programului va fi generata o exceptie, programul se va opri la instructiunea care a cauzat exceptia si se va afisa un mesaj de eroare de genul:

"Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException :10
at Exceptii.main (Exceptii.java:4)"

Crearea unui obiect de tip exceptie se numeste aruncarea unei exceptii (”throwing an exception”). In momentul in care o metoda genereaza (arunca) o exceptie sistemul de executie este responsabil cu gasirea unei secvente de cod dintr-o metoda care sa o trateze.

Cautarea se face recursiv, incepand cu metoda care a generat exceptia si mergand inapoi pe linia apelurilor catre acea metoda.

Secventa de cod dintr-o metoda care trateaza o anumita exceptie se numeste analizor de exceptie (”exception handler”) iar interceptarea si tratarea ei se numeste prinderea exceptiei (”catch the exception”).
Cu alte cuvinte, la aparitia unei erori este ”aruncata” o exceptie iar cineva trebuie sa o ”prinda” pentru a o trata.

Daca sistemul nu gaseste nici un analizor pentru o anumita exceptie, atunci programul Java se opreste cu un mesaj de eroare (in cazul exemplului de mai sus mesajul ”Aici nu se mai ajunge...” nu va fi afisat).

2. 1”Prinderea” si tratarea exceptiilor

Tratarea exceptiilor se realizeaza prin intermediul blocurilor de instructiuni **try, catch** si **finally**.

O secventa de cod care trateaza anumite exceptii trebuie sa arate astfel:

```
try {
     // Instructiuni care pot genera exceptii
}
catch (TipExceptie1 variabila) {
     // Tratarea exceptiilor de tipul 1
}
catch (TipExceptie2 variabila) {
     // Tratarea exceptiilor de tipul 2
}
. . .
finally {
     // Cod care se executa indiferent
     // daca apar sau nu exceptii
}
```

Citirea unui fisier octet cu octet si afisarea lui pe ecran. Fara a folosi tratarea exceptiilor metoda responsabila cu citirea fisierului ar arata astfel:

```
public static void citesteFisier(String fis) {
     FileReader f = null;

     // Deschidem fisierul
     System.out.println("Deschidem fisierul " + fis);
     f = new FileReader(fis);

     // Citim si afisam fisierul caracter cu caracter
     int c;
     while ((c=f.read()) != -1)
          System.out.print((char)c);

     // Inchidem fisierul
     System.out.println("\\nInchidem fisierul " + fis);
     f.close();
}
```

Aceasta secventa de cod va furniza erori la compilare deoarece in Java tratarea erorilor este obligatorie. Folosind mecanismul exceptiilor metoda citeste isi poate trata singura erorile care pot surveni pe parcursul executiei sale.

Mai jos este codul complet si corect al unui program ce afiseaza pe ecran continutul unui fisier al carui nume este primit ca argument de la linia de comanda.

Tratarea exceptiilor este realizata complet chiar de catre metoda **citeste**.

Citirea unui fisier – correct

```
import java.io.*;

public class CitireFisier {
     public static void citesteFisier(String fis) {
          FileReader f = null;
          try {
               // Deschidem fisierul
               System.out.println("Deschidem fisierul " + fis);
               f = new FileReader(fis);
               // Citim si afisam fisierul caracter cu caracter
               int c;
               while ((c = f.read()) != -1)
                    System.out.print((char)c);
          } catch (FileNotFoundException e) {
               // Tratam un tip de exceptie
               System.out.println("Fisierul nu a fost gasit!");
               System.out.println("Exceptie : " + e.getMessage());
          } catch (IOException e) {
               // Tratam alt tip de exceptie
               System.out.println("Eroare la citirea din fisier!");
               e.printStackTrace();
          } finally {
               if (f != null) {
                    // Inchidem fisierul
                    System.out.println("\nInchidem fisierul");
                    try {
                         f. close ();
                    } catch (IOException e) {
                         System.out.println("Nu poate fi inchis!");
                         e.printStackTrace();
                    }
               }
          }
     }
}
```

2.2 Propagarea erorilor

Propagarea unei erori se face pana la un analizor de exceptii corespunzator.

Sa presupunem ca apelul la metoda citesteFisier este consecinta unor apeluri imbricate de metode:

```
int metoda1() {
     metoda2();
     ...
}

int metoda2() {
     metoda3;
...
}

int metoda3() {
     citesteFisier();
...
}
```

Java permite unei metode sa arunce exceptiile aparute in cadrul ei la un nivel superior, adica functiilor care o apeleaza sau sistemului. Cu alte cuvinte, o metoda poate sa nu isi asume responsabilitatea tratarii exceptiilor aparute in cadrul ei:

```
int metoda1() {
     try {
          metoda2();
     }
     catch (TipExceptie e) {
          //proceseazaEroare;
     }
     ...
}

int metoda2() throws TipExceptie {
     metoda3();
     ...
}

int metoda3() throws TipExceptie {
     citesteFisier();
     ...
}
```

2.3 Gruparea erorilor dupa tipul lor

In Java exista clase corespunzatoare tuturor exceptiilor care pot aparea la executia unui program. Acestea sunt grupate in functie de similaritatile lor intr-o ierarhie de clase.

De exemplu, clasa **IOException** se ocupa cu exceptiile ce pot aparea la operatii de intrare/iesire si diferentiaza la randul ei alte tipuri de exceptii, cum ar fi **FileNotFoundException**, **EOFException**, etc.

La randul ei, clasa **IOException** se incadreaza intr-o categorie mai larga de exceptii si anume clasa **Exception**.

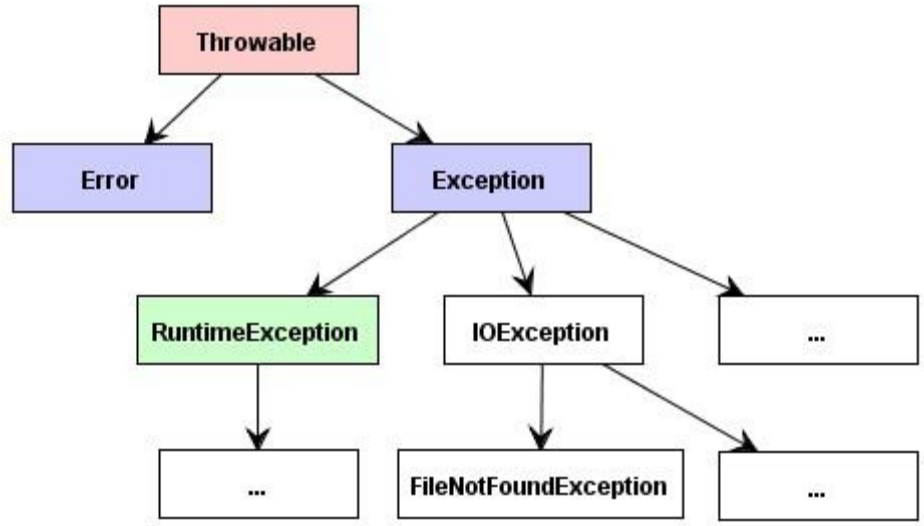
Radacina acestei ierarhii este clasa **Throwable** (vezi ”Ierarhia claselor ce descriu exceptii”).

Prinderea unei exceptii se poate face fie la nivelul clasei specifice pentru acea exceptie, fie la nivelul uneia din superclasele sale, in functie de necesitatile programului, insa, cu cat clasa folosita este mai generica cu atat tratarea exceptiilor programul isi pierde din flexibilitate.

```
try {
     FileReader f = new FileReader("input.dat");
     /* Acest apel poate genera exceptie
     de tipul FileNotFoundException
     Tratarea ei poate fi facuta in unul
     din modurile de mai jos:
     */
}
catch (FileNotFoundException e) {
     // Exceptie specifica provocata de absenta
     // fisierului ’input.dat’
} // sau

catch (IOException e) {
     // Exceptie generica provocata de o operatie IO
} // sau
catch (Exception e) {
     // Cea mai generica exceptie soft
} //sau
catch (Throwable e) {
     // Superclasa exceptiilor
}
```

2.4 Ierarhia claselor ce descriu exceptii

Radacina claselor ce descriu exceptii este clasa **Throwable** iar cele mai importante subclase ale sale sunt: **Error**, **Exception** si **RuntimeException**, care sunt la randul lor superclase pentru o serie intreaga de tipuri de exceptii.



Erorile, obiecte de tip **Error**, sunt cazuri speciale de exceptii generate de functionarea anormala a echipamentului hard pe care ruleaza un program Java si sunt invizibile programatorilor.

Un program Java nu trebuie sa trateze aparitia acestor erori si este improbabil ca o metoda Java sa provoace asemenea erori.

Exceptiile, obiectele de tip **Exception**, sunt **exceptiile standard** (soft) care trebuie tratate de catre programele Java. Dupa cum am mai zis tratarea acestor exceptii nu este o optiune ci o constrangere. Exceptiile care pot ”scapa” netratate descind din subclasa **RuntimeException** si se numesc **exceptii la executie**.

Metodele care sunt apelate uzual pentru un obiect exceptie sunt definite in clasa Throwable si sunt publice, astfel incat pot fi apelate pentru orice tip de exceptie.

Cele mai uzuale sunt:

- getMessage - afiseaza detaliul unei exceptii;
- printStackTrace - afiseaza informatii complete despre exceptie si localizarea ei;
- toString - metoda mostenita din clasa Object, care furnizeaza reprezentarea ca sir de caractere a exceptiei.

2.5 Exceptii la executie

In general, tratarea exceptiilor este obligatorie in Java. De la acest principiu se sustrag insa asa numitele exceptii la executie sau, cu alte cuvinte, exceptiile care provin strict din vina programatorului si nu generate de o anumita situatie externa, cum ar fi lipsa unui fisier.

Aceste exceptii au o superclasa comuna RuntimeException si in aceasta categorie sunt incluse exceptiile provocate de:

- operatii aritmetice ilegale (impartirea intregilor la zero);
  ArithmeticException
- accesarea membrilor unui obiect ce are valoarea null;
  NullPointerException
- accesarea eronata a elementelor unui vector.
  ArrayIndexOutOfBoundsException

Exceptiile la executie pot aparea oriunde in program si pot fi extrem de numeroase iar incercarea de ”prindere” a lor ar fi extrem de anevoioasa.

Din acest motiv, compilatorul permite ca aceste exceptii sa ramana netratate, tratarea lor nefiind insa ilegala. Reamintim insa ca, in cazul aparitiei oricarui tip de exceptie care nu are un analizor corespunzator, programul va fi terminat.

```
int v[] = new int[10];

try {
     v[10] = 0;
} catch (ArrayIndexOutOfBoundsException e) {
     System.out.println("Atentie la indecsi!");
     e.printStackTrace();
} // Corect, programul continua

v[11] = 0;
/* Nu apare eroare la compilare
dar apare exceptie la executie si
programul va fi terminat.
*/
System.out.println("Aici nu se mai ajunge...");
```

Impartirea la 0 va genera o exceptie doar daca tipul numerelor impartite este aritmetic intreg.

In cazul tipurilor reale (float si double) nu va fi generata nici o exceptie, ci va fi furnizat ca rezultat o constanta care poate fi, functie de operatie, **Infinity**, **-Infinity**, sau **Nan**.

```
int a=1, int b=0;
System.out.println(a/b); // Exceptie la executie !
```

```
double x=1, y=-1, z=0;
System.out.println(x/z); // Infinity
System.out.println(y/z); // -Infinity
System.out.println(z/z); // NaN
```

2.6 Crearea propriilor exceptii

Adeseori poate aparea necesitatea crearii unor exceptii proprii pentru a pune in evidenta cazuri speciale de erori provocate de metodele claselor unei biblioteci, cazuri care nu au fost prevazute in ierarhia exceptiilor standard Java.

O exceptie proprie trebuie sa se incadreze insa in ierarhia exceptiilor Java, cu alte cuvinte clasa care o implementeaza trebuie sa fie subclasa a uneia deja existente in aceasta ierarhie, preferabil una apropiata ca semnificatie, sau superclasa **Exception**.

```
public class ExceptieProprie extends Exception {
     public ExceptieProprie(String mesaj) {
     super(mesaj);
     // Apeleaza constructorul superclasei Exception
     }
}
```

1.6 Sincronizarea firelor de executie

Pana acum am vazut cum putem crea fire de executie independente si asincrone, cu alte cuvinte care nu depind in nici un fel de executia sau de rezultatele altor fire. Exista insa numeroase situatii cand fire de executie separate, dar care ruleaza concurent, trebuie sa comunice intre ele pentru a accesa diferite resurse comune sau pentru a-si transmite dinamic rezultatele ”muncii” lor. Cel mai elocvent scenariu in care firele de executie trebuie sa comunice intre ele este cunoscut sub numele de problema producatorului / consumatorului, in care producatorul genereaza un flux de date care este preluat si prelucrat de catre consumator.

Sa consideram de exemplu o aplicatie Java in care un fir de executie (producatorul) scrie date intr-un fisier in timp ce alt fir de executie (consumatorul) citeste date din acelasi fisier pentru a le prelucra. Sau sa presupunem ca producatorul genereaza niste numere si le plaseaza, pe rand, intr-un buffer iar consumatorul citeste numerele din acel buffer pentru a le procesa. In ambele cazuri avem de-a face cu fire de executie concurente care folosesc o resursa comuna: un fisier, respectiv o zona de memorie si, din acest motiv, ele trebuie sincronizate intr-o maniera care sa permita decurgerea normala a activitatii lor.

1.6.1 Scenariul producator / consumator

Pentru a intelege mai bine modalitatea de sincronizare a doua fire de executie sa implementam efectiv o problema de tip producator / consumator.
Sa consideram urmatoarea situatie:

- Producatorul genereaza numerele intregi de la 1 la 10, fiecare la un interval neregulat cuprins intre 0 si 100 de milisecunde. Pe masura ce le genereaza incearca sa le plaseze intr-o zona de memorie (o variabila intreaga) de unde sa fie citite de catre consumator.

- Consumatorul va prelua, pe rand, numerele generate de catre producator si va afisa valoarea lor pe ecran.

Pentru a fi accesibila ambelor fire de executie, vom incapsula variabila ce va contine numerele generate intr-un obiect descris de clasa Buffer si care va avea doua metode put (pentru punerea unui numar in buffer) si get (pentru obtinerea numarului din buffer).

Fara a folosi un mecanism de sincronizare clasa Buffer arata astfel:

```
//Clasa Buffer fara sincronizare

class Buffer {
     private int number = -1;
```

```
     public int get() {
          return number;
     }

     public void put(int number) {
          this.number = number;
     }
}
```

Vom implementa acum clasele Producator si Consumator care vor descrie cele doua fire de executie. Ambele vor avea o referinta comuna la un obiect de tip Buffer prin intermediul caruia isi comunica valorile.

```
//Clasele Producator si Consumator

class Producator extends Thread {
     private Buffer buffer;

     public Producator(Buffer b) {
          buffer = b;
     }

     public void run () {
          for(int i = 0; i < 10; i++) {
               buffer.put(i);
               System.out.println("Producatorul a pus :\t" + i);
               try {
                    sleep((int)(Math.random() * 100));
               } catch (InterruptedException e) {}
          }
     }
}

class Consumator extends Thread {
     private Buffer buffer;

     public Consumator (Buffer b) {
          buffer = b;
     }

     public void run () {
          int value = 0;
          for(int i = 0; i < 10; i++) {
               value = buffer.get();
               System.out.println("Consumatorul a primit :\t" +
               value);
          }
     }
}
```

```
public class TestSincronizare1 {
     public static void main (String[] args) {
          Buffer b = new Buffer();
          Producator p1 = new Producator(b);
          Consumator c1 = new Consumator(b);
          p1.start ();
          c1.start ();
     }
}
```

Dupa cum ne asteptam, rezultatul rularii acestui program nu va rezolva nici pe departe problema propusa de noi, motivul fiind lipsa oricarei sincronizari intre cele doua fire de executie.
Mai precis, rezultatul va fi ceva de forma:

```
Consumatorul a primit: -1
Consumatorul a primit: -1
Producatorul a pus: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Producatorul a pus: 1
Producatorul a pus: 2
Producatorul a pus: 3
Producatorul a pus: 4
Producatorul a pus: 5
Producatorul a pus: 6
Producatorul a pus: 7
Producatorul a pus: 8
Producatorul a pus: 9
```

Ambele fire de executie acceseaza resursa comuna, adica obiectul de tip Buffer, intr-o maniera haotica si acest lucru se intampla din doua motive:

- Consumatorul nu asteapta inainte de a citi ca producatorul sa genereze un numar si va prelua de mai multe ori acelasi numar.

- Producatorul nu asteapta consumatorul sa preia numarul generat inainte de a produce un altul, in felul acesta consumatorul va ”rata” cu siguranta unele numere (in cazul nostru aproape pe toate).

Problema care se ridica in acest moment este: cine trebuie sa se ocupe de sincronizarea celor doua fire de executie : clasele Producator si Consumator sau resursa comuna Buffer?
Raspunsul este evident: resursa comuna Buffer, deoarece ea trebuie sa permita sau nu accesul la continutul sau si nu firele de executie care o folosesc. In felul acesta efortul sincronizarii este transferat de la producator / consumator la un nivel mai jos, cel al resursei critice.
Activitatile producatorului si consumatorului trebuie sincronizate la nivelul resursei comune in doua privinte:

• Cele doua fire de executie nu trebuie sa acceseze simultan buffer-ul; acest lucru se realizeaza prin blocarea obiectului Buffer atunci cand este accesat de un fir de executie, astfel incat niciun alt fir de executie sa nu-l mai poata accesa (vezi ”Monitoare”).

• Cele doua fire de executie trebuie sa se coordoneze, adica producatorul trebuie sa gaseasca o modalitate de a ”spune” consumatorului ca a plasat o valoare in buffer, iar consumatorul trebuie sa comunice producatorului ca a preluat aceasta valoare, pentru ca acesta sa poata genera o alta. Pentru a realiza aceasta comunicare, clasa **Thread** pune la dispozitie metodele **wait**, **notify**, **notifyAll**. (vezi ”Semafoare”).

Folosind sincronizarea, clasa Buffer va arata astfel:

```
//Clasa Buffer cu sincronizare

class Buffer {
     private int number = -1;
     private boolean available = false;

     public synchronized int get() {
          while (!available) {
               try {
                    wait();
                    // Asteapta producatorul sa puna o valoare
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
          available = false;
          notifyAll ();
          return number;
     }

     public synchronized void put (int number) {
          while (available) {
               try {
                    wait();
                    // Asteapta consumatorul sa preia valoarea
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            this.number = number;
            available = true;
            notifyAll ();
        }
}
```

Rezultatul obtinut:

```
Producatorul a pus: 0
Consumatorul a primit: 0
Producatorul a pus: 1
Consumatorul a primit: 1
...
Producatorul a pus: 9
Consumatorul a primit: 9
```

1.6.2 Semafoare

Obiectul de tip Buffer din exemplul anterior are o variabila membra privata numita number, in care este memorat numarul pe care il comunica producatorul si pe care il preia consumatorul.

De asemenea, mai are o variabila privata logica available care ne da starea buffer-ului: daca are valoarea true inseamna ca producatorul a pus o valoare in buffer si consumatorul nu a preluat-o inca; daca este false, consumatorul a preluat valoarea din buffer dar producatorul nu a pus deocamdata alta la loc.

Deci, la prima vedere, metodele clasei Buffer ar trebui sa arate astfel:

```
public synchronized int get() {
     while (!available) {
          // Nimic - asteptam ca variabila sa devina true
     }
     available = false;
     return number;
}

public synchronized int put(int number) {
     while (available) {
          // Nimic - asteptam ca variabila sa devina false
     }
     available = true;
     this.number = number;
}
```

Varianta de mai sus, desi pare corecta, nu este. Aceasta deoarece implementarea metodelor este **”selfish”**, cele doua metode isi asteapta in mod egoist conditia de terminare. Ca urmare, corectitudinea functionarii va depinde de sistemul de operare pe care programul este rulat, ceea ce reprezinta o greseala de programare.

Punerea corecta a unui fir de executie in asteptare se realizeaza cu metoda **wait** a clasei **Thread**, care are urmatoarele forme:

void wait( )
void wait(long timeout)
void wait(long timeout, long nanos)

Dupa apelul metodei **wait**, firul de executie curent elibereaza monitorul asociat obiectului respectiv si asteapta ca una din urmatoarele conditii sa fie indeplinita:

• Un alt fir de executie informeaza pe cei care ”asteapta” la un anumit monitor sa se ”trezeasca” - acest lucru se realizeaza printr-un apel al metodei **notifyAll** sau **notify**.

• Perioada de asteptare specificata a expirat.

Metoda wait poate produce exceptii de tipul **InterruptedException**, atunci cand firul de executie care asteapta (este deci in starea **”Not Runnable”**) este intrerupt din asteptare si trecut fortat in starea **”Runnable”**, desi conditia asteptata nu era inca indeplinita.

Metoda **notifyAll** informeaza toate firele de executie care sunt in asteptare la monitorul obiectului curent indeplinirea conditiei pe care o asteptau. Metoda **notify** informeaza doar un singur fir de executie, specificat ca argument.

Reamintim varianta corecta a clasei Buffer:

```
//Folosirea semafoarelor
class Buffer {
     private int number = -1;
     private boolean available = false;

     public synchronized int get() {
          while (!available) {
               try {
                    wait();
                    // Asteapta producatorul sa puna o valoare
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
          available = false;
          notifyAll();
```

```
            return number ;
        }


        public synchronized void put (int number) {
            while (available) {
                try {
                    wait();
                    // Asteapta consumatorul sa preia valoarea
                } catch ( InterruptedException e) {
                    e.printStackTrace ();
                }
            }
            this.number = number;
            available = true;
            notifyAll();
        }
}
```

1.6.3 Probleme legate de sincronizare

Din pacate, folosirea monitoarelor ridica si unele probleme. Sa analizam cateva dintre ele si posibilele lor solutii:

Deadlock

Deadlock-ul este o problema clasica intr-un mediu in care ruleaza mai multe fire de executie si consta in faptul ca, la un moment dat, intreg procesul se poate bloca deoarece unele fire asteapta deblocarea unor monitoare care nu se vor debloca niciodata.

Exista numeroase exemple in acest sens, cea mai cunoscuta fiind ”Problema filozofilor”. Reformulata, sa ne imaginam doua persoane ”A” si ”B” (fire de executie) care stau la aceeasi masa si trebuie sa foloseasca in comun cutitul si furculita (resursele comune) pentru a manca. Evident, cele doua persoane doresc obtinerea ambelor resurse. Sa presupunem ca ”A” a obtinut cutitul si ”B” furculita. Firul ”A” se va bloca in asteptarea eliberarii furculitei iar firul ”B” se va bloca in asteptarea eliberarii cutitului, ceea ce conduce la starea de ”deadlock”. Desi acest exemplu este desprins de realitate, exista numeroase situatii in care fenomenul de ”deadlock” se poate manifesta, multe dintre acestea fiind dificil de detectat.

Exista cateva reguli ce pot fi aplicate pentru evitarea deadlock-ului:

- Firele de executie sa solicite resursele in aceeasi ordine. Aceasta abordare elimina situatiile de asteptare circulara.

• Folosirea unor monitoare care sa controleze accesul la un grup de resurse. In cazul nostru, putem folosi un monitor ”tacamuri” care trebuie blocat inainte de a cere furculita sau cutitul.

• Folosirea unor variabile care sa informeze disponibilitatea resurselor fara a bloca monitoarele asociate acestora.

• Cel mai importat, conceperea unei arhitecturi a sistemului care sa evite pe cat posibil aparitia unor potentiale situatii de deadlock.

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Variabile volatile

Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in its own memory.

The volatile keyword is used to say to the JVM "Warning, this variable may be modified in another Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword could be used on every kind of variable, either primitive or objects!
Cuvantul cheie volatile a fost introdus pentru a controla unele aspect legate de optimizarile efectuate de unele compilatoare.

Sa consideram urmatorul exemplu:

```
class TestVolatile {
     boolean test;

     public void metoda() {
          test = false;
          // *
          if (test) {
               // Aici se poate ajunge...
          }
     }
}
```

Un compilator care optimizeaza codul, poate decide ca variabila test fiind setata pe false, corpul if -ului nu se va executa si sa excluda secventa respectiva din rezultatul compilarii. Daca aceasta clasa ar fi insa accesata de mai multe fire de executie, variabile test ar putea fi setata pe true de un alt fir, exact intre instructiunile de atribuire si if ale firului curent.

Declararea unei variabile cu modificatorul volatile informeaza compilatorul sa nu optimizeze codul in care aceasta apare, previzionand valoarea pe care variabila o are la un moment dat.

Fire de executie inaccesibile

Uneori firele de executie sunt blocate din alte motive decat asteptarea la un monitor, cea mai frecventa situatie de acest tip fiind operatiunile de intrare / iesire (IO) blocante. Cand acest lucru se intampla celelalte fire de executie trebuie sa poata accesa in continuare obiectul. Dar daca operatiunea IO a fost facuta intr-o metoda sincronizata, acest lucru nu mai este posibil, monitorul obiectului fiind blocat de firul care asteapta de fapt sa realizeze operatia de intrare / iesire. Din acest motiv, operatiile IO nu trebuie facute in metode sincronizate.

1.6.4 Gruparea firelor de executie

Gruparea firelor de executie pune la dispozitie un mecanism pentru manipularea acestora ca un tot si nu individual. De exemplu, putem sa pornim sau sa suspendam toate firele dintr-un grup cu un singur apel de metoda.

Gruparea firelor de executie se realizeaza prin intermediul clasei ThreadGroup. Fiecare fir de executie Java este membru al unui grup, indiferent daca specificam explicit sau nu acest lucru.

Afilierea unui fir la un anumit grup se realizeaza la crearea sa si devine permanenta, in sensul ca nu vom putea muta un fir dintr-un grup in altul, dupa ce acesta a fost creat. In cazul in care cream un fir folosind un constructor care nu specifica din ce grup face parte, el va fi plasat automat in acelasi grup cu firul de executie care l-a creat.

La pornirea unui program Java se creeaza automat un obiect de tip ThreadGroup cu numele main, care va reprezenta grupul tuturor firelor de executie create direct din program si care nu au fost atasate explicit altui grup.

Cu alte cuvinte, putem sa ignoram complet plasarea firelor de executie in grupuri si sa lasam sistemul sa se ocupe cu aceasta, adunandu-le pe toate in grupul main.

Exista situatii insa cand gruparea firelor de executie poate usura substantial manevrarea lor.

Crearea unui fir de executie si plasarea lui intr-un grup (altul decat cel implicit) se realizeaza prin urmatorii constructori ai clasei Thread:

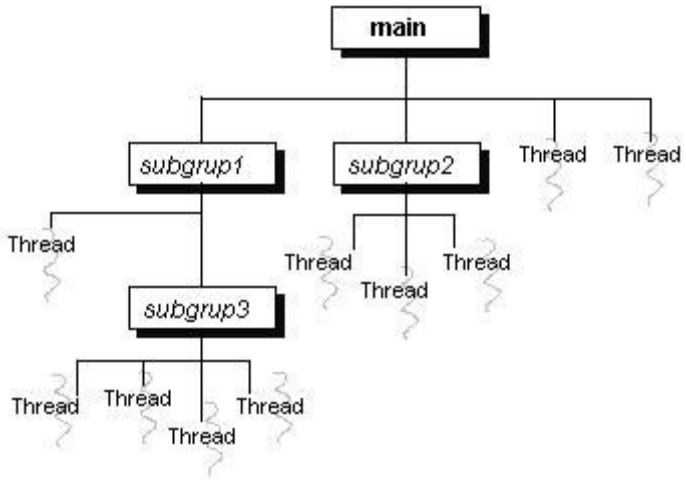
```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target, String name)
```

Fiecare din acesti costructori creeaza un fir de executie, il initializeaza si il plaseaza intr-un grup specificat ca argument. Pentru a afla carui grup apartine un anumit fir de executie putem folosi metoda getThreadGroup a clasei Thread.

In exemplul urmator vor fi create doua grupuri, primul cu doua fire de executie iar al doilea cu trei:

```
ThreadGroup grup1 = new ThreadGroup("Producatori");
Thread p1 = new Thread(grup1, "Producator 1");
Thread p2 = new Thread(grup1, "Producator 2");

ThreadGroup grup2 = new ThreadGroup("Consumatori");
Thread c1 = new Thread(grup2, "Consumator 1");
Thread c2 = new Thread(grup2, "Consumator 2");
Thread c3 = new Thread(grup2, "Consumator 3");
```

Un grup poate avea ca parinte un alt grup, ceea ce inseamna ca firele de executie pot fi plasate intr-o ierarhie de grupuri, in care radacina este grupul implicit main, ca in figura de mai jos:



1.6.5 Comunicarea prin fluxuri de tip ”pipe”

O modalitate deosebit de utila prin care doua fire de executie pot comunica este realizata prin intermediul canalelor de comunicatii (pipes). Acestea sunt implementate prin fluxuri descrise de clasele:

- PipedReader, PipedWriter - pentru caractere, respectiv
- PipedOutputStream, PipedInputStream - pentru octeti.

Fluxurile ”pipe” de iesire si cele de intrare pot fi conectate pentru a efectua transmiterea datelor. Acest lucru se realizeaza uzual prin intemediul constructorilor:

```
public PipedReader(PipedWriterpw)
public PipedWriter(PipedReaderpr)
```

In cazul in care este folosit un constructor fara argumente, conectarea unui flux de intrare cu un flux de iesire se face prin metoda connect:

```
public void connect(PipedWriterpw)
public void connect(PipedReaderpr)
```

Intrucat fluxurile care sunt conectate printr-un pipe trebuie sa execute simultan operatii de scriere / citire, folosirea lor se va face din cadrul unor fire de executie.

Fiecare capat al unui canal este utilizat dintr-un fir de executie separat. La un capat se scriu caractere, la celalalt se citesc. La citire, daca nu sunt date disponibile firul de executie se va bloca pana ce acestea vor deveni disponibile.

Se observa ca acesta este un comportament tipic producator / consumator asincron, firele de executie comunicand printr-un canal.

Realizarea conexiunii se face astfel:

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);

// sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);

// sau
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWirter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

Scrierea si citirea pe/de pe canale se realizeaza prin metodele uzuale read si write, in toate formele lor.
Sa reconsideram acum exemplul producator / consumator prezentat anterior, folosind canale de comunicatie.

Producatorul trimite datele printr-un flux de iesire de tip DataOutputStream catre consumator, care le primeste printr-un flux de intrare de tip DataInputStream.
Aceste doua fluxuri vor fi interconectate prin intermediul unor fluxuri de tip ”pipe”.

```
//Folosirea fluxurilor de tip ”pipe”
import java.io.*;

class Producator extends Thread {
     private DataOutputStream out;

     public Producator(DataOutputStream out) {
          this.out = out;
     }

     public void run() {
          for(int i = 0; i < 10; i++) {
               try {
                    out.writeInt(i);
               } catch (IOException e) {
                    e.printStackTrace ();
               }

               System.out.println("Producatorul a pus :\t" + i);
               try {
                    sleep((int)(Math.random () * 100));
               } catch (InterruptedException e) {}
          }
     }
}
```

```
class Consumator extends Thread {
     private DataInputStream in;

     public Consumator(DataInputStream in) {
          this.in = in;
     }

     public void run() {
          int value = 0;

          for(int i = 0; i < 10; i++) {
               try {
                    value = in.readInt();
               } catch (IOException e) {
                    e.printStackTrace();
               }
               System.out.println("Consumatorul a primit :\t" +
                                        value);
          }
     }
}

public class TestPipes {

     public static void main(String [] args)
                                   throws IOException {

          PipedOutputStream pipeOut = new PipedOutputStream();
          PipedInputStream pipeIn =
                              new PipedInputStream(pipeOut);

          DataOutputStream out = new DataOutputStream(pipeOut);
          DataInputStream in = new DataInputStream(pipeIn);

          Producator p1 = new Producator(out);
          Consumator c1 = new Consumator(in);

          p1.start();
          c1.start();
     }
}
```

1.6.6 Clasele Timer si TimerTask

Clasa Timer ofera o facilitate de a planifica diverse actiuni pentru a fi realizate la un anumit moment de catre un fir de executie ce ruleaza in fundal.

Actiunile unui obiect de tip Timer sunt implementate ca instante ale clasei TimerTask si pot fi programate pentru o singura executie sau pentru executii repetate la intervale regulate.

Pasii care trebuie facuti pentru folosirea unui timer sunt:

- Crearea unei subclase Actiune a lui TimerTask si supreadefinirea metodei run ce va contine actiunea pe care vrem sa o planificam. Pot fi folosite si clase anonime.

- Crearea unui fir de executie prin instantierea clasei Timer;

- Crearea unui obiect de tip Actiune;

- Planificarea la executie a obiectuluii de tip Actiune, folosind metoda schedule din clasa Timer;

Metodele de planificare pe care le avem la dispozitie au urmatoarele formate:

schedule(TimerTask task, Date time)
schedule(TimerTask task, long delay, long period)
schedule(TimerTask task, Date time, long period)
scheduleAtFixedRate(TimerTask task, long delay, long period)
scheduleAtFixedRate(TimerTask task, Date time, long period)

unde, task descrie actiunea ce se va executa, delay reprezinta intarzierea fata de momentul curent dupa care va incepe executia, time momentul exact la care va incepe executia iar period intervalul de timp intre doua executii.

Metodele de planificare se impart in doua categorii:

- schedule - planificare cu intarziere fixa: daca dintr-un anumit motiv actiunea este intarziata, urmatoarele actiuni vor fi si ele intarziate in consecinta

- scheduleAtFixedRate - planificare cu numar fix de rate: daca dintr-un anumit motiv actiunea este intarziata, urmatoarele actiuni vor fi executate mai repede, astfel incat numarul total de actiuni dintr-o perioada de timp sa fie tot timpul acelasi

Un timer se va opri natural la terminarea metodei sale run sau poate fi oprit fortat folosind metoda cancel. Dupa oprirea sa el nu va mai putea fi folosit pentru planificarea altor actiuni.

De asemenea, metoda System.exit va oprit fortat toate firele de executie si va termina aplicatia curenta.

```
//Folosirea claselor Timer si TimerTask

import java.util.*;
```

```
class Atentie extends TimerTask {
     public void run () {
          System.out.print (".");
     }
}

class Alarma extends TimerTask {
     public String mesaj ;

     public Alarma (String mesaj) {
          this.mesaj = mesaj;
     }

     public void run () {
          System.out.println (mesaj);
     }
}

public class TestTimer {

     public static void main (String args []) {

          // Setam o actiune repetitiva, cu rata fixa
          final Timer t1 = new Timer ();
          t1.scheduleAtFixedRate (new Atentie (), 0, 1*1000);

          // Folosim o clasa anonima pentru o alta actiune
          Timer t2 = new Timer ();
          t2.schedule (new TimerTask () {

               public void run () {
                    System.out.println("S-au scurs 10 secunde");

                    // Oprim primul timer
                    t1.cancel ();
               }
          }, 10*1000);

          // Setam o actiune pentru ora 22:30
          Calendar calendar = Calendar.getInstance ();
          calendar.set(Calendar.HOUR_OF_DAY , 22);
          calendar.set(Calendar.MINUTE , 30);
          calendar.set(Calendar.SECOND , 0);
          Date ora = calendar.getTime ();
          Timer t3 = new Timer ();
          t3.schedule (new Alarma ("Alarma !"), ora);
     }
}
```