



Academia Tehnică Militară "Ferdinand I"  
Facultatea de Sisteme Informaticе și Securitate Cibernetică

# PARADIGME DE PROGRAMARE (ÎN JAVA)

## CURS 2 - PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

COL(R) TRAIAN NICULA

# TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- **Programare orientată pe obiecte (OOP) (1/3)**
- Programare funcțională și asincronă (3)
- Programare reactivă (1)
- Servicii web REST (2)
- Quarkus Framework (4)

# CUPRINS CURS

- Tipuri de date și operatori
- Instructiuni de control
- Clase obiecte și metode
- Tipuri de date și operatori – revăzut
- Bibliografie



# TIPURI DE DATE ȘI OPERATORI

- Java este un limbaj **"strong typed"**, compilatorul verifică toate operațiile pentru compatibilitatea de tip. La compilare , tipul datelor determină operațiile permise
- Java suportă 2 tipuri de date: **orientate pe obiecte și primitive**

## Tipuri primitive

- Java suportă 8 tipuri de date primitive (slide următor)
- Tipurile nu sunt obiecte ci valori binare normale
- Există 3 tipuri întregi: **short** pe 16 biți, **int** pe 32 biți și **long** pe 64 biți
- **byte** pe 8 biți este util pentru procesarea datelor binare
- Întregii din Java sunt **big-endian** (bitul cel mai semnificativ este stocat primul)
- Java nu suportă tipul întreg pozitiv (**unsigned integer**)
- Există 2 tipuri în virgulă mobilă: **float** pe 32 biți și **double** pe 64 biți
- Tipul **char** în Java folosește Unicode, fiind stocat pe 16 biți ca întreg pozitiv (**unsigned integer**)



# TIPURI DE DATE ȘI OPERATORI

## TIPURI PRIMITIVE

Tip	Lungime în biți	Domeniu
<b>boolean</b>	1	true/false
<b>byte</b>	8	-128 la 127
<b>char</b>	16	0 la 65,536
<b>double</b>	64	$1.7976931348623157 \times 10^{308}$ , $4.9406564584124654 \times 10^{-324}$
<b>float</b>	32	$3.40282347 \times 10^{38}$ , $1.40239846 \times 10^{-45}$
<b>int</b>	32	-2,147,483,648 la 2,147,483,647
<b>long</b>	64	-9,223,372,036,854,775,808 la 9,223,372,036,854,775,807
<b>short</b>	16	-32,768 la 32,767

# TIPURI DE DATE ȘI OPERATORI

## TIPURI PRIMITIVE - EXEMPLE

```
src > main > java > atm > paradigms > ❶ Inches.java > ...
1  package atm.paradigms;
2
3  public class Inches {
4      Run | Debug
5      public static void main(String[] args) {
6          long im; // inches in a mile
7          long ci; // cubic inches in a mile
8          im = 5280 * 12;
9          ci = im * im * im;
10         System.out.println("There are " + ci + " inches in a cubic mile");
11     }
12 }
```

Rezultat: There are 254358061056000  
inches in a cubic mile

```
src > main > java > atm > paradigms > ❷ Pythagora.java > ...
1  package atm.paradigms;
2
3  public class Pythagora {
4      Run | Debug
5      public static void main(String[] args) {
6          double x, y, z;
7          x = 3;
8          y = 4;
9          z = Math.sqrt(x*x + y*y);
10         System.out.println("Hypotenuse is " + z);
11     }
12 }
```

Rezultat: Hypotenuse is 5.0



# TIPURI DE DATE ȘI OPERATORI

## TIPURI PRIMITIVE - EXEMPLE

```
src > main > java > atm > paradigms > CharDeno.java > ...
1 package atm.paradigms;
2
3 public class CharDeno {
4     Run | Debug
5     public static void main(String[] args) {
6         char chr = 'X'; // assign value
7         System.out.println("chr value is "
8                             + chr + " and int value is " + (int)chr);
9
10        chr++; // increment char value
11        System.out.println("chr is now " + chr);
12
13        chr = 90; // assign integer value
14        System.out.println("chr is now " + chr);
15    }
}
```

Rezultat:

chr value is X and int value is 88  
chr is now Y  
chr is now Z



# TIPURI DE DATE ȘI OPERATORI

## LITERALS

- Se referă la valori fixe ce pot fi citite de oameni. Se mai numesc și constante
- Pot fi orice tipuri primitive
- Constantele de tip **caracter** sunt marcate de apostrof: ‘a’, ‘%’
- Constantele de tip **întreg** sunt reprezentate ca numere fără parte fracționară: **123, 23\_456**
- Implicit, constantele întreg sunt de tip **int**. Pentru a specifica **long** după număr se pune litera **I** sau **L** (12 este *int*, iar 12L este *long*)
- **int** poate fi atribuit variabilelor de tip **char, byte, short** dacă valoare poate fi reprezentată de acest tip. Este necesară conversia de tip
- Constanta **int** poate fi atribuită și la tipul **long**
- Constantele de tip **virgulă mobilă** constau din partea zecimală, punct apoi partea fracționară: **11.234**. Se poate folosi și notația științifică
- Implicit, constantele în virgulă mobilă sunt de tip **double**. Pentru a specifica tipul **float** se adaugă **f** sau **F** după constantă (**10.19** este *double*, iar **10.19F** este *float*)



# TIPURI DE DATE ȘI OPERATORI

## LITERALS

- Constantele hexazecimale încep cu 0x (0xFF). Constante octale încep cu 0 (011)

Codul Escape	Descriere
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\'	single quote ('')
\"	double quote (")
\?	question mark (?)
\\\	backslash (\ )

# TIPURI DE DATE ȘI OPERATORI

## LITERALS - EXEMPLE

```
src > main > java > atm > paradigms > TypeCast.java > ...
1 package atm.paradigms;
2
3 public class TypeCast {
4     Run | Debug
5     public static void main(String[] args) {
6         int i = 100;
7         short s = (short) i; // cast needed
8         byte b = (byte) i; // cast needed
9         char c = (char) i; // cast needed
10        long l = i;
11
12        double d = 12.123;
13        float f = (float) d; // cast needed
14        f = 1.456f;
15    }
16 }
```

# TIPURI DE DATE ȘI OPERATORI

## SCOPUL ȘI DURATA DE VIAȚĂ A VARIABILELOR

- În majoritatea limbajelor există **scop global** și **local**
- În Java o clasificare mai bună a scopului variabilelor este la nivel **de clasă** și la nivel **de metodă**
- Scopul definit de metodă începe și se termină cu paranteze accolade
- **Parametrii** sunt inclusi în scopul metodei
- Un bloc delimitat de paranteze accolade reprezintă un scop
- Ca o regulă generală, **variabilele definite într-un scop nu sunt vizibile pentru codul din afara acestuia**
- Scopurile pot fi imbricate
- **Variabilele declarate în scopul extern sunt vizibile pentru scopul intern**
- Reciproca este falsă
- Regulile privind scopul stau la baza **încapsulării**

# TIPURI DE DATE ȘI OPERATORI

## SCOPUL ȘI DURATA DE VIAȚĂ A VARIABILELOR

- Variabile sunt **create la intrarea în scop și șterse la ieșirea din acesta**
- Variabile definite în metode sunt disponibile pentru codul din acestea
- Timpul de viață al unei variabile ține de scopul acesteia
- Variabilele declarate într-un scop intern **nu pot avea același nume** cu variabilele din scopul extern
- Într-un bloc, **variabilele pot fi declarate oriunde dar pot fi utilizate numai după declarare**
- Variabilele vor fi reinitializate de fiecare dată la intrarea în blocul în care sunt declarate

# TIPURI DE DATE ȘI OPERATORI

## SCOPUL ȘI DURATA DE VIAȚĂ A VARIABILELOR - EXEMPLU

```
App.java 3 ScopeDemo.java X
src > main > java > atm > paradigms > ScopeDemo.java > ...
1 package atm.paradigms;
2
3 public class ScopeDemo {
4     Run | Debug
5     public static void main(String[] args) {
6         int x = 10; // visible to all code in main method
7         // new scope
8         {
9             int y = 20; // y is not visible outside the block
10            // x is visible here
11            System.out.println("x is " + x + "\ny is " + y);
12        }
13        System.out.println("x is " + x );
14    }
15 }
```

y nu mai există la ieșirea din scop

Rezultat:

x is 10

y is 20

x is 10



# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ARITMETICI

- Java are 4 clase de operatori: aritmetici, operații pe biți, relationali și logici

Operator	Meaning
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ARITMETICI - EXEMPLE

```
src > main > java > atm > paradigms > ArithmOperDemo.java > ArithmOperDemo
 1 package atm.paradigms;
 2
 3 public class ArithmOperDemo {
 4     Run | Debug
 5     public static void main(String[] args) {
 6         int ires, irem;
 7         double dres, drem;
 8
 9         ires = 10 / 3;
10         irem = 10 % 3;
11         System.out.print("Result is " + ires);
12         System.out.println(" Reminder is " + irem);
13
14         dres = 10.0 / 3.0;
15         drem = 10.0 % 3.0;
16         System.out.print("Result is " + dres);
17         System.out.println(" Reminder is " + drem);
18     }
}
```

Rezultat:

*Result is 3 Reminder is 1*

*Result is 3.333333333333335 Reminder is 1.0*



# TIPURI DE DATE ȘI OPERATORI

## OPERATORI RELAȚIONALI ȘI LOGICI

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>&gt;</code>	Greater than
<code>&lt;</code>	Less than
<code>&gt;=</code>	Greater than or equal to
<code>&lt;=</code>	Less than or equal to

Operator	Meaning
<code>&amp;</code>	AND
<code> </code>	OR
<code>^</code>	XOR (exclusive OR)
<code>  </code>	Short-circuit OR
<code>&amp;&amp;</code>	Short-circuit AND
<code>!</code>	NOT

Operanții operatorilor `&` și `|` sunt evaluări  
întotdeauna

Operanții operatorilor `&&` și `||` sunt  
evaluați la nevoie



# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ATRIBUIRE

***var = expression;***

- Tipul variabilei trebuie să fie compatibil cu tipul expresiei
- Operatorul de atribuire ia valoarea expresiei din partea dreaptă
- Cod valabil

***int x, y, z;***

***X = y = z = 100;***

- Scurtături la atribuire

<b><i>+ =</i></b>	<b><i>- =</i></b>	<b><i>* =</i></b>	<b><i>/ =</i></b>
<b><i>% =</i></b>	<b><i>&amp; =</i></b>	<b><i>  =</i></b>	<b><i>^ =</i></b>

# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ATRIBUIRE – CONVERSIE DE TIP

- Când tipurile sunt compatibile, valoarea din dreapta este atribuită tipului din stânga
- O valoare de tip **int** poate fi atribuită unei variabile de tip **float**

**int i = 10;**  
**float f = i;**

**Conversia automată** are loc când:

- Cele 2 tipuri sunt compatibile
- Tipul destinație este mai larg (widening) decât tipul sursă
- Exemplu: **int** este suficient de larg pentru a conține tipul **byte** și ambele sunt de același tip (întreg)
- Tipurile întregi pot fi convertite automat la tipurile virgulă mobilă (**long** la **double**)
- Tipul boolean nu este compatibil cu tipurile numerice și tipul **char**



# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ATRIBUIRE – CONVERSIE DE TIP

- Cazurile în care conversia automată nu este posibilă se face conversia de tip implicit folosind instrucțiunea **cast**
- Conversia explicită se fac: **(target-type)expression**
- Exemplu:

```
double x,y;
```

```
int z;
```

```
z = (int)(x/y);
```

- **Conversia explicită** se numește **conversie de îngustare** (narrowing)
- Informația poată să se piardă

# TIPURI DE DATE ȘI OPERATORI

## OPERATORI ATRIBUIRE – CONVERSIE DE TIP

```
src > main > java > atm > paradigms > CastDemo.java > CastDemo > main(String[])
1 package atm.paradigms;
2
3 public class CastDemo {
4     Run | Debug
5     public static void main(String[] args) {
6         double x=10.0, y=3.0;
7         byte b;
8         int i;
9         char ch;
10
11         i = (int) (x/y); // truncation due to double to int cast
12         System.out.println("x/y = " + i);
13
14         i = 100;
15         b = (byte) i; // no loss because a byte can hold 100
16         System.out.println(" b = " + b);
17         i = 257;
18         b = (byte) i; // info loss because a byte cannot hold 257
19         System.out.println(" b = " + b);
20
21         b = 88; // ASCII for X
22         ch = (char) b;
23         System.out.println("ch = " + ch);
24     }
}
```

Rezultat:

$x/y = 3$

$b = 100$

$b = 1$

$ch = X$

# INSTRUCȚIUNI DE CONTROL

## IF

- Formă

```
if (condition) {  
    statement sequence  
}  
else {  
    statement sequence  
}
```

- **condition** este expresie booleană. Dacă este adevărată se execută **if** altfel se execută **else**

# INSTRUCȚIUNI DE CONTROL

## IF - EXEMPLU

```
src > main > java > atm > paradigms > ❶ GuessLetter.java > ❷ GuessLetter > ❸ main(String[])
1 package atm.paradigms;
2
3 import java.io.IOException;
4
5 public class GuessLetter {
6     Run | Debug
7     public static void main(String[] args) throws IOException {
8         char ch, answer = 'k';
9
10        System.out.println("Pick a letter between a and z:");
11        ch = (char) System.in.read();
12
13        if (ch == answer) System.out.println("Very good!!!"); // Line 13
14        else {
15            System.out.print("Sorry, you are ");
16            if (ch < answer) System.out.println("too low");
17            else
18                System.out.println("too high");
19        }
20    }
}
```

Citește de la consolă un caracter, îl compară lexicografic cu litera k și afișează rezultatul.

# INSTRUCȚIUNI DE CONTROL IF-ELSE-IF

- Formă

**if (condition)**

**statement;**

**else if (condition)**

**statement;**

**else if (condition)**

**statement;**

**else**

**statement;**

- Se evaluatează de sus în jos până se găsește o condiție adevărată

- Apoi se execută instrucțiunea sau blocul de instrucțiuni, iar restul de condiții nu vor mai fi evaluate

# INSTRUCȚIUNI DE CONTROL

## SWITCH

- Alternativă mai eficientă la **if** în anumite situații
- Valoarea unei expresii este testată succesiv față de o listă de constante
- Când se găsește o potrivire instrucțiunile asociate sunt executate până la instrucțiunea **break**
- Expresia din **switch** poate fi de tip **char**, **byte**, **short**, **int**, **enum**, **String**
- Constantele din **case** trebuie să aibă același tip cu expresia
- Secvența de instrucțiuni din **default** se execută dacă nu se găsește nicio potrivire
- Dacă secvența de instrucțiuni din **case** nu se termină cu **break**, atunci și instrucțiunile următoarelor case se vor executa până la primul **break**

# INSTRUCȚIUNI DE CONTROL

## SWITCH

```
package atm.paradigms;
import java.io.IOException;

public class SwitchDemo {
    public static void main(String[] args) throws IOException {
        int i = (char) System.in.read() - '0';
        switch (i) {
            case 0:
                System.out.println("i is zero");
                break;
            case 1:
                System.out.println("i is one");
                break;
            case 2:
                System.out.println("i is two");
                break;
            default:
                System.out.println("i is three or more");
        }
    }
}
```

Citește de la consolă un întreg, îl compară cu câteva valori întregi și afișează rezultatul.

# INSTRUCȚIUNI DE CONTROL

## BUCLA FOR

- Formă pentru o singură instrucțiune

***for(initialization; condition; iteration) statement;***

- Formă pentru bloc

***for(initialization; condition; iteration)***

{

***statement sequence***

}

- ***Initialization*** setează valoarea inițială a variabilei de control a buclei
- ***Condition*** este o expresie booleană ce controlează execuția buclei
- ***Iteration*** stabilește cu cât se schimbă variabila de control la repetarea buclei

# INSTRUCȚIUNI DE CONTROL

## BUCLA FOR - EXEMPLE

- Diverse opțiuni pentru inițializare, condiție și iteratie

```
for (int i = 0; i < 10; i++){  
    double sqrt = Math.sqrt(i);  
    System.out.println(sqrt);  
}
```

```
for (double d = 10.0; d >= 0.0; d-- ){  
    double sqrt = Math.sqrt(d);  
    System.out.println(sqrt);  
}
```

```
for (int x = 100; x > -100; x -= 5) {  
    System.out.println(x);  
}
```

```
for (int i = 0, j = 10; i < j; i++, j--)  
    System.out.println(i + " " + j);
```

```
for (int i = 0; (char) System.in.read()  
!= 'S'; i++)  
    System.out.println("Pass: " + i);
```

```
for (int i = 0; i < 10;) {  
    System.out.println("Pass #" + i);  
    i++; // increment loop control var  
}
```

# INSTRUCȚIUNI DE CONTROL

## BUCLA WHILE

- Formă

***while(condition) statement;***

- ***Statement*** poate fi o singură instrucțiune sau un grup
- ***Condition*** este o expresie booleană
- Bucla se repetă câtă atâta timp cât condiția este adevărată

```
char ch = 'a';
while (ch <= 'z'){
    System.out.println(ch);
    ch++;
}
```



# INSTRUCȚIUNI DE CONTROL

## BUCLA DO-WHILE

- Spre deosebire de **for** și **while**, **do-while** verifică condiția la sfârșitul buclei
- Formă

```
do {  
    statements;  
} while(condition);
```

- Se execută cel puțin o dată

```
char ch;  
do {  
    ch = (char) System.in.read();  
} while (ch == '\n' | ch == '\r');
```



# INSTRUCȚIUNI DE CONTROL

## BREAK - IEȘIREA DIN BUCLA

- Cu instrucția **break** se poate ieși din orice buclă și se execută instrucțiunile de după acesta
- Cu **break** se ieșe doar din bucla interioară în care se află
- Cu **break** în **switch** se ieșe doar din acesta nu și din eventuala buclă în care se află instrucția **switch**

```
int t = 0;
while (t < 100) {
    if (t == 10)
        break;
    System.out.print(t + " ");
    t++;
}
```

# INSTRUCȚIUNI DE CONTROL CONTINUE

- Forțează următoarea iterare a buclei sărind peste execuția codului care îi urmează
- Trece direct la expresia condițională a buclei și continuă procesarea normală

```
for (int i = 0; i <= 100; i++) {  
    if ((i % 2) != 0)  
        continue; // iterate  
    System.out.println(i);  
}
```



# CLASE OBIECTE ȘI METODE

## CLASE

- Programarea Java are loc în clase
- Toate exemplele anterioare au folosit clase și metoda **main**
- **Clasa este un șablon** care definește forma unui obiect
- Conține atât datele cât și codul ce operează pe acestea
- Clasele sunt abstracțiuni logice
- Obiectele sunt instanțe ale clasei și sunt create în memorie pe baza specificațiilor clasei
- Metodele și variabilele care alcătuiesc o clasă poartă numele de membri ai acestea

# CLASE OBIECTE ȘI METODE

## CLASE

- Forma generală a definiției clasei

```
class classname {  
    // declare instance variables  
    type var1;  
    type var2;  
    // ...  
    type varN;  
    // declare methods  
    type method1(parameters) {  
        // body of method  
    }  
    type method2(parameters) {  
        // body of method  
    }  
    // ...  
    type methodN(parameters) {  
        // body of method  
    }  
}
```

# CLASE OBIECTE ȘI METODE

## DEFINIREA UNEI CLASE

- Ca exemplu definim clasa **Vehicle** care stochează ca date: numărul de pasageri, capacitatea rezervorului și consumul mediu

```
public class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity  
    int mpg; // fuel consumption  
}
```

- Prin definirea unei clase se creează un nou tip de date – **Vehicle**
- Definirea unei clase nu creează nici un obiect
- Pentru a crea un obiect de tip **Vehicle** se folosește sintaxa

```
Vehicle minivan = new Vehicle();
```

# CLASE OBIECTE ȘI METODE

## DEFINIREA UNEI CLASE

```
public class VehicleDemo {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 60;  
        minivan.mpg = 12;  
        // compute the range assuming a full tank of gas  
        int range = minivan.fuelcap * minivan.mpg;  
        System.out.println("Minivan can carry " + minivan.passengers +  
                           " with a range of " + range);  
    }  
}
```

- La compilare rezultă 2 fișiere cu extensia .class



# CLASE OBIECTE ȘI METODE

## CREAREA OBIECTELOR

- Crearea unui obiect

```
Vehicle minivan = new Vehicle();
```

- Instrucțiunea efectuează 2 funcții:
  - Declară variabila **minivan** de tip **Vehicle**. Variabila declarată referă obiectul nu îl conține
  - Creează o copie fizică a obiectului și atribuie variabilei **minivan** o referință la acesta cu operatorul **new**
- Operatorul **new** alocă dinamic memorie pentru obiect și returnează o **referință** la acesta
- În Java toate obiectele sunt alocate dinamic
- Referința este adresa de memorie a obiectului alocat cu **new**



# CLASE OBIECTE ȘI METODE

## CREAREA OBIECTELOR

- Pentru **tipurile primitive**, când se atribuie o variabilă la alta, variabila din stânga primește o copie a valorii variabilei din dreapta

```
int a, b;  
a = 10;  
b = a;  
a = 20;  
System.out.println("a: " + a + " b: " + b);
```

Rezultat: a: 20 b: 10

- La atribuirea unei variabile referință de obiect, valoarea alteia, poate duce la rezultate neintuitive

```
Vehicle car1 = new Vehicle();  
Vehicle car2 = car1;
```

- Variabile `car1` și `car2` referă același obiect, iar obiectul poate fi modificat prin ambele variabile

# CLASE OBIECTE ȘI METODE

## CREAREA OBIECTELOR

```
Vehicle minivan1 = new Vehicle();
Vehicle minivan2 = minivan1;

// assign values to fields in minivan1
minivan1.passengers = 7;
minivan1.fuelcap = 60;
minivan1.mpg = 12;
// assign values for minivan2
minivan2.passengers = 9;
minivan2.fuelcap = 80;
minivan2.mpg = 13;
System.out.println("Minivan 1 can carry " + minivan1.passengers +
    " with a range of " + (minivan1.fuelcap * minivan1.mpg));
System.out.println("Minivan 2 can carry " + minivan2.passengers +
    " with a range of " + (minivan2.fuelcap * minivan2.mpg));
```

Variabilele minivan1 și minivan2 referă și modifică același obiect. Tipărirea proprietăților asignate prin cele două variabile sunt identice.

Minivan 1 can carry 9 with a range of 1040  
Minivan 2 can carry 9 with a range of 1040



# CLASE OBIECTE ȘI METODE

## METODE

- Metodele alături de **variabilele instanței** alcătuiesc clasele
- Metodele sunt subroutines care permit accesarea datelor (getters și setters) dar și manipularea acestora
- În Java o metodă ar trebui să execute o singură sarcină
- Forma generală a unei metode

```
ret-type name( parameter-list ) {  
    // body of method  
}
```

- **ret-type** specifică tipul de date returnat de metodă
- Dacă metoda nu returnează o valoare atunci returnează tipul **void**

# CLASE OBIECTE ȘI METODE

## METODE

```
public class VehicleDemo {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportcar = new Vehicle();  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 60;  
        minivan.mpg = 12;  
        // assign values for sportcar  
        sportcar.passengers = 2;  
        sportcar.fuelcap = 80;  
        sportcar.mpg = 13;  
        System.out.println("Minivan 1 can carry " + minivan.passengers +  
                           " with a range of " + minivan.range());  
        System.out.println("Minivan 2 can carry " + sportcar.passengers +  
                           " with a range of " + sportcar.range());  
    }  
}
```

```
public class Vehicle {  
    int passengers; // no. of passengers  
    int fuelcap; // fuel capacity  
    int mpg; // fuel consumption  
  
    // Calculate the range  
    int range() {  
        return fuelcap * mpg;  
    }  
}
```

# CLASE OBIECTE ȘI METODE

## METODE

- O metodă **void** poate returna la închiderea parantezelor accolade sau la execuția unei instrucțiuni **return**
- Într-o metodă **void** pot exista mai multe instrucțiuni **return** dacă există 2 sau mai multe rute de ieșire din metodă

```
void myMethod() {  
    // ...  
    if(done) return;  
    // ...  
    if(error) return;  
}  
  
• Metodele care returnează valori sunt folosite în multe situații (rezultatul unui calcul,  
succes sau eșec, coduri de stare etc.)  
  
    return value;
```



# CLASE OBIECTE ȘI METODE

## METODE

- Metodele pot primi valori ca parametrii
- **Parametrii** sunt declarați în parantezele rotunde care apar după numele metodei
- Valoarea transmisă metodei prin parametru poartă numele de **argument**
- Sintaxa pentru declararea parametrilor este aceeași cu cea pentru variabile
- **Parametrul este vizibil în scopul metodei** și acționează ca o variabilă locală



# CLASE OBIECTE ȘI METODE

## METODE

```
public class VehicleDemo {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportcar = new Vehicle();  
        int dist = 400;  
        // assign values in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 60;  
        minivan.mpg = 12;  
        // assign values for sportcar  
        sportcar.passengers = 9;  
        sportcar.fuelcap = 80;  
        sportcar.mpg = 13;  
        System.out.println("To go " + dist + " km");  
        System.out.println("Minivan needs " + minivan.fuelneeded(dist));  
        System.out.println("Sportcar needs " + sportcar.fuelneeded(dist));  
    }  
}
```

```
public class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity  
    int mpg; // fuel consumption  
    // Calculate the range  
    int range() {  
        return fuelcap * mpg;  
    }  
    // Fuel needed for a given distance  
    double fuelneeded(int distance) {  
        return Math.round((double) (distance *  
10 / mpg))/10.0;  
    }  
}
```



# CLASE OBIECTE ȘI METODE CONSTRUCTORI

- **Constructorul initializează obiectul când acesta este creat**
- **Are același nume cu clasa și este sintactic similar cu metoda**
- **Constructorii nu returnează un tip**
- Se folosesc pentru inițializarea variabilelor de instanță aparținând clasei sau pentru efectuarea altor acțiuni la crearea obiectului
- **Toate clasele au constructori chiar dacă au fost definiți sau nu**
  - Java automat furnizează un constructor implicit și initializează toate variabile membre la zero sau null
  - La definirea unui constructor, cel implicit nu se mai utilizează
  - Adesea constructorul acceptă unul sau mai mulți parametrii

# CLASE OBIECTE ȘI METODE CONSTRUCTORI

```
public class VehicleDemo {  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle(7, 60, 12);  
        Vehicle sportcar = new Vehicle(2, 80, 12);  
        int dist = 400;  
        System.out.println("To go " + dist  
                           + " km");  
        System.out.println("Minivan needs "  
                           + minivan.fuelneeded(dist));  
        System.out.println("Sportcar needs "  
                           + sportcar.fuelneeded(dist));  
    }  
}
```

```
public class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in liters  
    int mpg; // fuel consumption in km/l  
    // Constructor with params  
    Vehicle(int p, int f, int m){  
        passengers = p;  
        fuelcap = f;  
        mpg = m;  
    }  
    int range() {  
        return fuelcap * mpg;  
    }  
    // Fuel needed for a given distance  
    double fuelneeded(int distance) {  
        return (double) (distance / mpg);  
    }  
}
```

# CLASE OBIECTE ȘI METODE

## GARBAGE COLLECTION (GC)

- **Obiectele sunt alocate dinamic** în zona de memorie liberă folosind operatorul **new**
- Memoria nu este infinită, o componentă critică a alocării dinamice este eliberarea memoriei ocupate obiecte nefolosite
- **Sistemul GC** al Java funcționează automat fără intervenția programatorului
- Mod de lucru: când nu mai există nicio referință la un obiect din memorie se presupune că acesta nu mai este necesar și spațiul de memorie ocupat este eliberat, dar nu imediat
- Pentru eficiență sistemul GC rulează când sunt îndeplinite 2 condiții: există obiecte care pot fi reciclate și este nevoie să fie eliberat spațiul ocupat
- Programatorul nu știe exact când GC intră în acțiune



# CLASE OBIECTE ȘI METODE

## METODA FINALIZE

- În clasa obiectului se poate defini metoda **finalize()** care se execută înaintea distrugerii acestuia de către sistemul GC
- Forma generală a metodei **finalize()**

```
protected void finalize()  
{  
    // finalization code here  
}
```

- **protected** este specificator de acces care nu permite accesarea codului metodei din afara clasei
- **finalize()** se execută doar atunci când GC intră în acțiune și nu la ieșirea din scop a obiectului

# CLASE OBIECTE ȘI METODE

## THIS

- La execuția unei metode Java trece automat acesteia o referință la obiectul pe care se cheamă metoda
- Referința poartă numele de **this**
- Prin folosirea **this** numele parametrului metodei să aibă același nume cu variabilele de instanță

```
public class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
    // Constructor with params  
    Vehicle(int passengers, int fuelcap, int mpg){  
        this.passengers = passengers;  
        this.fuelcap = fuelcap;  
        this.mpg = mpg;  
    }  
    int range(){  
        return fuelcap * mpg;  
    }  
    double fuelneeded(int distance) {  
        int mpg = 20; // useless local variable  
        return (double) (distance / this.mpg);  
    }  
}
```

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## VECTORI (ARRAYS)

- Sunt colecții de variabile de același tip referite printr-un singur nume
- Pot avea una sau mai multe dimensiuni
- În Java vectorii sunt implementați ca obiecte iar crearea lor se face în 2 etape:
  - Declararea unei variabile referințe la vector
  - Alocarea memoriei pentru vector și atribuirea unei referințe la acesta variabilei declarate anterior
- Declarație vector unidimensional

`type array-name[ ] = new type[size];`

`type[ ] array-name= new type[size];`

- **type** determină tipul de date pentru fiecare element conținut în vector
- **size** determină numărul de elemente din vector
- Vectorii în Java sunt alocați dinamic cu operatorul **new**

# TIPURI DE DATE ȘI OPERATORI - REVĂZUT

## VECTORI (ARRAYS)

- Elementele individuale ale unui vector se accesează folosind **indexul**
- Indexul descrie poziția elementului, primul având poziția 0
- Exemplu:

```
public class ArrayDemo {  
    public static void main(String[] args) {  
        int array[] = new int[5];  
        // assign values  
        for (int i = 0; i < 5; i++)  
            array[i] = i;  
        // read array  
        for (int i = 0; i < 5; i++)  
            System.out.println("array[" + i + "]: " + i);  
    }  
}
```

Rezultat:  
array[0]: 0  
array[1]: 1  
array[2]: 2  
array[3]: 3  
array[4]: 4

# TIPURI DE DATE ȘI OPERATORI - REVĂZUT

## VECTORI MULTIDIMENSIONALI

- Vectorul multidimensional este un **vector de vectori**
- Declarație vector bidimensional

```
int table[][] = new int[10][20];  
  
public class Array2D {  
    public static void main(String[] args) {  
        int table[][] = new int[3][4];  
        for (int i = 0; i < 3; i++){  
            for (int j = 0; j < 4; j++){  
                table[i][j] = j + i * 4 + 1;  
                System.out.print(table[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

Creează și tipărește o matrice cu 3 rânduri și 4 coloane populată cu numere de la 1 la 12.

Rezultat:  
1 2 3 4  
5 6 7 8  
9 10 11 12

# TIPURI DE DATE ȘI OPERATORI - REVĂZUT

## VECTORI MULTIDIMENSIONALI

- La alocarea vectorilor multidimensionali este obligatoriu să se specifice doar prima dimensiune
- A 2-a dimensiune se stabilește manual

```
int table[][] = new int[3][];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];
```

- Când se alocă dimensiunile separat se pot specifica dimensiuni diferite pentru indecsii diferiti

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## INIȚIALIZARE VECTORI

- Vectori unidimensionali

*type-specifier array\_name[ ] = { val, val, val, ..., val };*

- Vectori multidimensionali

*type-specifier array\_name[ ][ ] = {*

*{ val, val, val, ..., val },*

*{ val, val, val, ..., val },*

*{ val, val, val, ..., val }*

*};*

- **val** indică o valoare pentru inițializare

- Blocul interior reprezintă un rând, iar blocurile se separă cu virgulă

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## VARIABILA LENGTH

- Vectorii fiind implementați ca obiecte aceștia au o variabilă de instanță numită **length** care stochează numărul de elemente

```
public class LengthDemo {  
    public static void main(String[] args) {  
        int array1[] = new int[10];  
        int[] array2 = { 1, 2, 3 };  
        int table[][] = { // a variable-length table  
            { 1, 2, 3 },  
            { 4, 5 },  
            { 6, 7, 8, 9 }  
        };  
        System.out.println("length of array1: " + array1.length);  
        System.out.println("length of array2: " + array2.length);  
        System.out.println("length of table: " + table.length);  
    }  
}
```

Rezultat:

*length of array1: 10  
length of array2: 3  
length of table: 3*

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## BUCLA FOR-EACH

- Bucla **for-each** trece printr-o colecție de obiecte, precum un vector, în mod secvențial de la început la sfârșit
- Forma generală

**for(*type* *itr-var* : *collection*) *statement-block***

- **type** specifică tipul variabilei de iterare și **itr-var** numele variabilei de iterare care primește elementele colecției unul câte unul de la început la sfârșit
- Suportă diferite tipuri de colecții
- Cu fiecare iterare a buclei următorul element al colecției este preluat și stocat în variabila **itr-var**

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## BUCLA FOR-EACH

- pentru vector alcătuit din tipuri primitive variabila de iterare **primește o copie a valorii elementului**

```
public class ForEachDemo {  
    public static void main(String[] args) {  
        int nums[] = { 1, 2, 3, 4, 5, 6};  
        int sum = 0;  
        for (int x : nums)  
            sum += x;  
        System.out.println(sum);  
    }  
}
```

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## BUCLA FOR-EACH

- Pentru vector de obiecte variabila de iterare primește o referință la obiectul stocat în elementul curent
- Referința nu se schimbă dar obiectul poate fi modificat

```
public class ForEachDemo {  
    public static void main(String[] args) {  
        // array of objects  
        MyInt nums[] = {new MyInt(1), new MyInt(2), new MyInt(3)};  
        for (MyInt o : nums) {  
            // o is reference  
            int x = o.getval();  
            // object instance variable can be altered  
            o.setval(++x);  
        }  
        for (int i = 0; i < nums.length; i++)  
            System.out.println(nums[i].getval());  
    }  
}
```

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## STRINGS

- Definește șiruri de caractere
- În Java **șirurile de caractere sunt obiecte**
- Șirurile de caractere pot fi construite cu operatorul **new**

```
String str = new String("Hello World!");
```

- O altă cale de a construi obiecte de tip **String**

```
String str = "Java strings are powerful.;"
```

- **str** este inițializat cu secvența de caractere
- **Conținutul** obiectului String **nu mai poate fi modificat**
- Secvența de caractere care alcătuiesc șirul de caractere nu mai poate fi modificată

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## STRINGS

```
public class StringDemo {  
    public static void main(String[] args) {  
        // declare strings in various ways  
        String str1 = new String("Java strings are objects.");  
        String str2 = "They are constructed various ways.";  
        String str3 = new String(str2);  
        System.out.println(str1);  
        System.out.println(str2);  
        System.out.println(str3);  
    }  
}
```

Rezultat:

*Java strings are objects.  
They are constructed various ways.  
They are constructed various ways.*

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## CLASA STRING - METODE

Metode	Descriere
<b>boolean equals(String str)</b>	Returnează true dacă sirul conține aceeași secvență de caractere cu str
<b>int length( )</b>	Returnează lungimea sirului de caractere
<b>char charAt(int index)</b>	Returnează caracterul de la indexul specificat
<b>int compareTo(String str)</b>	Returnează mai mic ca zero dacă sirul este mai lexicografic înaintea str, mai mare ca zero dacă sirul este lexicografic după str și zero dacă sirurile sunt egale
<b>int indexOf(String str)</b>	Caută în sir subșirul specificat de str. Returnează indexul primei potriviri sau -1 dacă nu-l găsește
<b>int lastIndexOf(String str)</b>	Caută în sir subșirul specificat de str. Returnează indexul ultim ei potriviri sau -1 dacă nu-l găsește
<b>String format(String format, Object... Args)</b>	Returnează un String formatat pe baza formatului dat și argumentelor
<b>String[] split(String regex)</b>	Împarte un String într-un vector de substring-uri
<b>String substring(int startIndex)</b>	Returnează o parte a unui String

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## CLASA STRING - METODE

```
public class StringDemo {  
    public static void main(String[] args) {  
        String str1 = "abc";  
        String str2 = new String(str1);  
        String str3 = "defg";  
        System.out.println("Length of str1: " +  
                           str1.length());  
        System.out.println(str1.charAt(0));  
        if (str1.equals(str2))  
            System.out.println("str1 equals str2");  
        else  
            System.out.println("str1 does not equal str2");  
        int result = str1.compareTo(str3);  
        System.out.println(result);  
        int idx = str3.indexOf("fg");  
        System.out.println(idx);  
    }  
}
```

Rezultat:

Length of str1: 3

a

str1 equals str2

-3

2

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## ARGUMENTE ÎN LINIA DE COMANDĂ

- Metoda `main()` acceptă parametrul `args` de tip vector de `String`
- Argumentele din linia de comandă constituie informația aflată după numele programului la execuție

```
public class CLDemo {  
    public static void main(String[] args) {  
        System.out.println("There are " + args.length +  
                           " command-line arguments.");  
        System.out.println("They are: ");  
        for (int i = 0; i < args.length; i++)  
            System.out.println("arg[" + i + "]: " + args[i]);  
    }  
}
```

```
PS D:\> 'C:\Program Files\Java\jdk-11.0.12\bin\java.exe' 'atm.paradigms.CLDemo' one two three  
There are 3 command-line arguments.  
They are:  
arg[0]: one  
arg[1]: two  
arg[2]: three
```

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## OPERATORUL CONDIȚIONAL ?

- Operatorul ? instrucțiuni de tip ***if-else***
- Operatorul ? se numește **operator ternar** (are 3 operanzi)
- Operatorul ? are forma generală

***Exp1 ? Exp2 : Exp3;***

- ***Exp1*** este o expresie booleană
- ***Exp2*** și ***Exp3*** sunt expresii de orice tip dar să fie aceeași
- Valoarea operatorului ? Se determină astfel:
  - Dacă ***Exp1*** este evaluată la true atunci se evaluatează ***Exp2*** și valoarea returnată devine valoarea expresiei
  - Dacă ***Exp1*** este evaluată la false atunci se evaluatează ***Exp3*** și valoarea returnată devine valoarea expresiei

# TIPURI DE DATE ȘI OPERATORI – REVĂZUT

## OPERATORUL CONDIȚIONAL ?

```
public class NoZeroDiv {  
    public static void main(String[] args) {  
        for (int i = -3; i < 4; i++) {  
            int result = i != 0 ? 100 / i : 0; // avoids zero devide  
            if (i != 0)  
                System.out.println("100 / " + i + ": " + result);  
        }  
    }  
}
```

Rezultat:

100 / -3: -33  
100 / -2: -50  
100 / -1: -100  
100 / 1: 100  
100 / 2: 50  
100 / 3: 33

# BIBLIOGRAFIE

- **Java™: A Beginner's Guide, Third Edition**, Herbert Schildt, McGraw-Hill, 2005
- **Thinking in Java Fourth Edition** Bruce Eckel, Prentice Hall, 2006
- **Head First Java™, Second Edition**, Kathy Sierra and Bert Bates, O'Reilly, 2005
- **Using Java with 101 Examples**, Atiwong Suchato, First Printing, 2011