



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

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 3 - PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

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

CUPRINS CURS

- Clase și metode – avansat
- Moștenirea
- Pachete și interfețe
- Tratarea exceptiilor
- Convenții de nume
- Ghid de structurare cod
- Bibliografie



CLASE ȘI METODE – AVANSAT

CONTROLUL ACCESULUI

- Sprijină încapsularea furnizând mijloace pentru controlul accesului la membrii clasei
- **Controlul accesului** la membrii clasei se realizează prin specifiatorii de acces: **public, private și protected**
- Permitând accesul la datele private printr-un set de metode se împiedică accesarea datelor sensibile dar și atribuirea de valori greșite
- Un membru **public** poate fi accesat de cod din afara clasei
- Un membru **private** poate fi accesat de cod din afara clasei numai prin alte metode publice definite de clasă
- Lipsa specifiatorilor de acces înseamnă acces implicit, respectiv la nivel pachet
- Specifiatorul **protected** este implicat în moștenirea obiectelor

CLASE ȘI METODE – AVANSAT

CONTROLUL ACCESULUI

```
public class AccessControl {  
    private int prv;  
    public int pub;  
    // acces methods  
    public void setPrv(int var1) {  
        this.prv = var1;  
    }  
    public int getPrv() {  
        return prv;  
    }  
}
```

```
public class AccessControlTest {  
    public static void main(String[] args) {  
        AccessControl ac = new AccessControl();  
        ac.prv = 10; // WRONG! private member  
        ac.setPrv(10); // OK - public method  
        // OK - public method  
        System.out.println(ac.getPrv());  
        ac.pub = 20; // OK - public member  
    }  
}
```

CLASE ȘI METODE – AVANSAT

OBIECTE CA PARAMETRII PENTRU METODE

```
public class Circle {
    private int radius;
    public Circle(int radius){
        this.radius = radius;
    }
    public int getRadius() {
        return radius;
    }
}
```

```
public class CircleMath {
    public double getPerimeter(Circle c){
        return 2 * Math.PI * c.getRadius();
    }
    public double getArea(Circle c){
        return Math.PI * Math.pow(c.getRadius(), 2.0);
    }
}
```

```
public class CircleAreaTest {
    public static void main(String[] args) {
        Circle c = new Circle(5);
        CircleMath areaCalc = new CircleMath();
        System.out.println("Perimeter: " + areaCalc.getPerimeter(c));
        System.out.println("Area: " + areaCalc.getArea(c));
    }
}
```

Metoda primește ca parametru o referință de tip Circle

Se trece ca argument metodei o referință la o instanță a clasei Circle



CLASE ȘI METODE – AVANSAT

METODE - TRECEREA ARGUMENTELOR

- **Tipurile primitive** sunt trecute metodei prin **copierea valorii**
- Modificările făcute asupra parametrului în metodă nu afectează argumentul original
- **Obiectele** sunt trecute metodei ca argumente prin **copierea referinței** la acesta
- Parametrul de tip referință este utilizat în codul metodei pentru accesarea obiectului
- **Modificările** făcute asupra parametrului **vor afecta obiectul** trimis ca argument prin referință
- Referința în sine este trecută prin valoare metodei, dar chiar dacă este o copie referă același obiect

CLASE ȘI METODE – AVANSAT

METODE - TRECEREA ARGUMENTELOR

```
public class PrimitiveArgs {
    private int a;
    private int b;
    public int getA() {
        return a;
    }
    public void setA(int a) {
        this.a = a;
    }
    public int getB() {
        return b;
    }
    public void setB(int b) {
        this.b = b;
    }
    public void modifyAB(){
        this.a++;
        this.b--;
    }
}
```

```
public class PrimitiveArgsTest {
    public static void main(String[] args) {
        PrimitiveArgs pa = new PrimitiveArgs();
        int a=10, b=20;
        pa.setA(a);
        pa.setB(b);
        pa.modifyAB();
        System.out.println("a: " + a + " b: " + b);
        System.out.println("pa.a: " + pa.getA()
                           + " pa.b: " + pa.getB());
    }
}
```

Variabilele a și b trecute ca argumente (prin valoare) rămân neschimbate, chiar dacă valorile câmpurilor clasei se modifică

Rezultat:
a: 10 b: 20
pa.a: 11 pa.b: 19

CLASE ȘI METODE – AVANSAT

METODE - TRECEREA ARGUMENTELOR

```
public class RefArgs {
    private int a;
    private int b;
    public RefArgs(int a, int b){
        this.a = a;
        this.b = b;
    }
    public int getA() {
        return a;
    }
    public int getB() {
        return b;
    }
    public void swap(RefArgs ra){
        // private but same class
        ra.a = this.a;
        ra.b = this.b;
    }
}
```

```
public class RefsArgsTest {
    public static void main(String[] args) {
        RefArgs o1 = new RefArgs(10, 20);
        RefArgs o2 = new RefArgs(80, 90);
        o1.swap(o2);
        System.out.println("Obj1: a= " + o1.getA()
            + ", b=" + o1.getB());
        System.out.println("Obj2: a= "
            + o2.getA() + ", b=" + o2.getB());
    }
}
```

Metoda **swap** atribuie câmpurilor obiectului de același tip, primit ca referință, valorile corespunzătoare proprii (fiind o instanță a aceleasi clase poate accesa chiar și câmpurile private). Aplicarea metodei pentru 2 instanțe diferite produce aceleași valori pentru câmpuri

Rezultat:
Obj1: a= 10, b=20
Obj2: a= 10, b=20

CLASE ȘI METODE – AVANSAT

METODE – RETURNAREA DE OBIECTE

- O metodă poate returna orice tip de date inclusiv referințe la obiecte

```
public class Error {  
    private String msg;  
    private int severity;  
    public Error(String msg, int severity){  
        this.msg = msg;  
        this.severity = severity;  
    }  
    @Override  
    public String toString(){  
        return "Info: " + this.msg  
        + " Severity: " + this.severity;  
    }  
}
```

```
public class ErrorInfo {  
    private String msgs[] = {  
        "Output Error",  
        "Input Error",  
        "Disk Full",  
        "Index Out-Of-Bounds"  
    };  
    private int severities[] = {3, 3, 2, 4};  
    public Error getErrorInfo(int i){  
        if (i >= 0 && i < msgs.length){  
            return new Error(msgs[i], severities[i]);  
        } else {  
            return new Error("Invalid Error Code", 0);  
        }  
    }  
}
```

Metoda returnează o referință la un obiect de tip Error

CLASE ȘI METODE – AVANSAT

METODE – RETURNAREA DE OBIECTE

- Când referința la obiect este returnată din metodă, obiectul va exista atât timp cât există referințe la el
- Chiar dacă execuția metodei s-a terminat obiectul va continua să existe

```
public class ErrorInfoTest {  
    public static void main(String[] args) {  
        ErrorInfo ei = new ErrorInfo();  
        Error e = ei.getErrorInfo(2);  
        System.out.println(e);  
        e = ei.getErrorInfo(19);  
        System.out.println(e);  
    }  
}
```

Referința la obiectul de tip Error va exista până la reatribuirea altei referințe, sau ieșirea din metodă. GC va șterge ulterior obiectul din memorie.

Rezultat:

Info: Disk Full Severity: 2

Info: Invalid Error Code Severity: 0

CLASE ȘI METODE – AVANSAT

SUPRAÎNCĂRCAREA METODELOR

- Două sau mai multe metode dintr-o clasă pot avea același nume atât timp cât declararea parametrilor lor este diferită
- **Supraîncărcarea metodelor (overloading)** este o modalitate de **implementare a polimorfismului** în Java
- **Tipul și/sau numărul de parametrii ai fiecărei metode supraîncărcate trebuie să difere**
- Tipurile de date returnate pot să difere pentru metode dar nu constituie un criteriu pentru supraîncărcare
- La executarea unei metode supraîncărcate se alege cea a cărei număr și tip de parametrii corespunde cu argumentele furnizate

CLASE ȘI METODE – AVANSAT

SUPRAÎNCĂRCAREA METODELOR

```
public class Overload {  
    public void method() {  
        System.out.println("No params");  
    }  
    public void method(int a) {  
        System.out.println("One param: " + a);  
    }  
    public int method(int a, int b) {  
        System.out.print("Result of: " + a  
                        + " + " + b + " = ");  
        return a + b;  
    }  
    public double method(double a, double b){  
        System.out.print("Result of: " + a  
                        + " + " + b + " = ");  
        return a + b;  
    }  
}
```

```
public class OverloadTest {  
    public static void main(String[] args) {  
        Overload ov = new Overload();  
        ov.method();  
        ov.method(2);  
        System.out.println(ov.method(4, 5));  
        System.out.println(ov.method(5.6, 7.4));  
    }  
}
```

Rezultat:
No parameters
One parameter: 2
Result of: 4 + 5 = 9
Result of: 5.6 + 7.4 = 13.0

CLASE ȘI METODE – AVANSAT

SUPRAÎNCĂRCAREA CONSTRUCTORILOR

- Permite construirea de obiecte în mai multe feluri
- Diferența se face prin numărul și tipul de parametrii

```
public class ConsOvl {  
    private int x;  
    public ConsOvl() {  
        this.x = 0;  
    }  
    public ConsOvl(int i) {  
        this.x = i;  
    }  
    public ConsOvl(double d) {  
        this.x = (int) d;  
    }  
    public ConsOvl(int i, int j) {  
        this.x = i * j;  
    }  
    public int getX() {  
        return x;  
    }  
}
```

```
public class ConsOvlTest {  
    public static void main(String[] args) {  
        ConsOvl c1 = new ConsOvl();  
        System.out.println("ConsOvl(): " + c1.getX());  
        ConsOvl c2 = new ConsOvl(60);  
        System.out.println("ConsOvl(int): " + c2.getX());  
        ConsOvl c3 = new ConsOvl(12.4);  
        System.out.println("ConsOvl(double): " + c3.getX());  
        ConsOvl c4 = new ConsOvl(12, 20);  
        System.out.println("ConsOvl(int,int): " + c4.getX());  
    }  
}
```

Rezultat:

ConsOvl(): 0
ConsOvl(int): 60
ConsOvl(double): 12
ConsOvl(int,int): 240



CLASE ȘI METODE – AVANSAT

STATIC

- Uneori este util să se definească membrii ai clasei care pot fi folosiți independent de instanțe
- Declarația acestor membrii este precedată de cuvântul cheie **static**
- Membrul clasei declarat **static**, poate fi accesat înainte ca orice obiect al clasei să fie creat
- Un exemplu comun de membru static este metoda **main()**
- Membrul static poate fi accesat folosind numele clasei urmat de operatorul punct apoi numele membrului static
- În esență **variabile statice** sunt **echivalentul variabilelor globale** din alte limbiage
- Variabilele statice nu sunt copiate, **toate instanțele clasei folosesc aceeași variabilă**



CLASE ȘI METODE – AVANSAT

STATIC

- Metodele declarate ca statice au câteva restricții: pot **folosi doar alte metode statice; acceseză numai date statice; nu au referința this**

```
public class Static {
    private int x;
    private static int y;
    public int getX() {
        return this.x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public static int getY() {
        return y;
    }
    public static void setY(int y) {
        Static.y = y;
    }
}
```

Metode publice statice pentru accesarea și modificarea variabilei statice y. Toate instanțele “văd” aceeași valoarea a lui y.

```
public class StaticTest {
    public static void main(String[] args) {
        Static obj1 = new Static();
        Static obj2 = new Static();
        // instance variable access
        obj1.setX(5);
        obj2.setX(10);
        System.out.println("obj1.x: " + obj1.getX() +
                           "\nobj2.x: " + obj2.getX());
        // access through class name
        Static.setY(500);
        System.out.println("Static.y: " + Static.getY());
    }
}
```

obj1.x: 5
obj2.x: 10
Static.y: 500

CLASE ȘI METODE – AVANSAT

BLOCURI STATICHE

- Blocul static este executat la încărcarea clasei. Este folosit pentru inițializări

```
public class StaticBlock {  
    private static double val;  
    static{  
        System.out.println("Static block");  
        val = Math.sqrt(3.0);  
    }  
    public static double getVal() {  
        return val;  
    }  
}
```

```
public class StaticBlockTest {  
    public static void main(String[] args) {  
        StaticBlock sb = new StaticBlock();  
        System.out.println("val value: "  
                           + StaticBlock.getVal());  
    }  
}
```

Rezultat:
Static block
val value: 1.7320508075688772



CLASE ȘI METODE – AVANSAT

CLASE CUIBĂRITE (NESTED) ȘI INTERIOARE (INNER)

- **Clasa cuibărită este o clasă definită în interiorul altei clase**
- Scopul clasei cuibărite se limitează la clasa care o conține
- **Clasa cuibărită are acces la membrii (inclusiv privați) ai clasei care o conține**
- Clasa exterioară nu are acces la membrii privați ai clasei cuibărite
- **Clasele cuibărite care nu sunt statice se numesc și clase interioare**
- Clasa interioară este folosită pentru a furniza un set de servicii utilizate doar de clasa care o conține

CLASE ȘI METODE – AVANSAT

CLASE CUIBĂRITE (NESTED) ȘI INTERIOARE (INNER)

```
public class Outer {
    private int[] vals;
    public Outer(int[] vals) {
        this.vals = vals;
    }
    class Inner {
        int avg() {
            int a = 0;
            for (int i = 0; i < vals.length; i++)
                a += vals[i];
            return a / vals.length;
        }
    }
    public void compute(){
        Inner in = new Inner();
        System.out.println("Average: " + in.avg());
    }
}
```

Clasa interioară este folosită pentru a calcula media aritmetică a întregilor conținuți în vectorul `vals` al clasei exterioare

```
public class OuterTest {
    public static void main(String[] args) {
        int [] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
        Outer out = new Outer(x);
        out.compute();
    }
}
```

Rezultat:
Average: 5

CLASE ȘI METODE – AVANSAT

VARARGS: ARGUMENTE CU LUNGIME VARIABILĂ

- Permite crearea de **metode care primesc un număr variabil de argumente**
- Lista variabilă de argumente este **definită prin 3 puncte (...)**
- Parametrul cu lungime variabilă trebuie să fie ultimul declarat**

```
public class VarArgs {
    public void vaTest(int... v) {
        System.out.println("Number of args: "
            + v.length);
        System.out.print("Contents: ");
        for (int i = 0; i < v.length; i++)
            System.out.print(" " + v[i] + " ");
        System.out.println();
    }
}
```

În scopul metodei parametrul de tipul **varargs** este văzut ca vector

```
public class VarArgsTest {
    public static void main(String[] args) {
        VarArgs va = new VarArgs();
        va.vaTest(10);
        va.vaTest(1, 2, 3);
        va.vaTest();
    }
}
```

Number of args: 1

Contents: 10

Number of args: 3

Contents: 1 2 3

Number of args: 0

Contents:

CLASE ȘI METODE – AVANSAT

VARARGS ȘI AMBIGUITATEA

```
public class VarArgs {  
    public void vaTest(int... v) {  
        System.out.println("Number of args: " + v.length);  
    }  
    public void vaTest(boolean... v) {  
        System.out.println("Number of args: " + v.length);  
    }  
    public void vaTest(int a, int... v) {  
        System.out.println("Number of args: " + v.length);  
    }  
}
```

```
public class VarArgsTest {  
    public static void main(String[] args) {  
        VarArgs va = new VarArgs();  
        va.vaTest(true, true, false); // OK  
        va.vaTest(10);  
        // Error: Ambiguous!  
        va.vaTest(1, 2, 3);  
        // Error: Ambiguous!  
        va.vaTest();  
        // Error: Ambiguous!  
    }  
}
```

Metodele sunt definite ambiguu iar compilatorul nu știe pe care să o cheue



MOȘTENIREA

- Principiu fundamental al OOP
- **Clasa care este moștenită** poartă numele de **superclasă**
- **Clasa care moștenește** poartă numele de **subclasă**
- Subclasa este o versiune specializată a superclasei
- Moștenește variabilele și metodele superclasei și adaugă pe cele proprii specifice
- Moștenirea în Java se realizează permitând unei clase să încorporeze o altă clasă (doar una) în declarația ei prin cuvântul cheie **extends**
- Forma generală

```
class subclass-name extends superclass-name {  
    // body of class  
}
```



MOȘTENIREA ACCESUL LA MEMBRII SUPERCLASEI

- Membrii privați ai unei superclase nu pot fi accesati direct din subclasă

```
public class Shape {  
    private double width;  
    private double height;  
    public double getWidth() {  
        return width;  
    }  
    public double getHeight() {  
        return height;  
    }  
    public void setWidth(double width) {  
        this.width = width;  
    }  
    public void setHeight(double height) {  
        this.height = height;  
    }  
    public void showDims(){  
        System.out.println("Width: " + this.width  
        + " Height: " + this.height);  
    }  
}
```

```
public class Triangle extends Shape {  
    private String type;  
    public double area(){  
        // cannot access width and height directly  
        return getWidth() * getHeight() / 2;  
    }  
    public String getType() {  
        return type;  
    }  
    public void setType(String type) {  
        this.type = type;  
    }  
}
```



MOȘTENIREA ACCESUL LA MEMBRII SUPERCLASEI

```
public class TriangleTest {  
    public static void main(String[] args) {  
        Triangle t = new Triangle();  
        t.setWidth(4.0);  
        t.setHeight(4.0);  
        t.setType("isosceles");  
        System.out.println("Info for triangle: ");  
        System.out.println("Type: " + t.getType());  
        t.showDims();  
        System.out.println("Area is " + t.area());  
    }  
}
```

Clasa Triangle moștenește variabilele și metodele superclasei dar poate accesa direct doar pe cele publice

Rezultat:

Info for triangle:

Type: isosceles

Width: 4.0 Height: 4.0

Area is 8.0



MOȘTENIREA CONSTRUCTORII

- Superclasele și subclasele au proprii constructori
- **Constructorul superclasei construiește partea sa de obiect iar constructorul subclasei la fel**
- **Dacă doar superclasa definește un constructor fără argumente, la crearea unui obiect al subclasei, constructorul implicit cheamă automat constructorul superclasei**
- **Dacă atât superclasa cât și subclasa definesc constructori se utilizează cuvântul cheie *super***
- ***super()*** este prima instrucțiune din constructorul subclasei
- Forma generală
super(parameter-list);
- ***parameter-list*** reprezint parametrii ceruți de constructorul superclasei

MOȘTENIREA CONSTRUCTORII

```
public class Shape {  
    private double width;  
    private double height;  
    public Shape(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
    public double getWidth() {  
        return width;  
    }  
    public double getHeight() {  
        return height;  
    }  
    public void showDims(){  
        System.out.println("Width: " + this.width  
        + " Height: " + this.height);  
    }  
}
```

```
public class Triangle extends Shape {  
    private String type;  
    public Triangle( double width,  
                    double height, String type){  
        super(width, height); // first line  
        this.type = type;  
    }  
    public double area(){  
        // cannot access width and height directly  
        return getWidth() * getHeight() / 2;  
    }  
    public String getType() {  
        return type;  
    }  
}
```

Constructorul Triangle cheamă pe prima linie constructorul Shape cu cei 2 parametrii width și height

MOȘTENIREA CONSTRUCTORII

```
public class TriangleTest {  
    public static void main(String[] args) {  
        Triangle t = new Triangle(4.0, 4.0, "isosceles");  
        System.out.println("Info for triangle: ");  
        System.out.println("Type: " + t.getType());  
        t.showDims();  
        System.out.println("Area is " + t.area());  
    }  
}
```

Rezultat:
Info for triangle:
Type: isosceles
Width: 4.0 Height: 4.0
Area is 8.0

MOȘTENIREA CONSTRUCTORII

```
public class Shape {  
    private double width;  
    private double height;  
    public Shape(){  
        this.width = this.height = 0.0;  
    }  
    public Shape(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
    public double getWidth() {  
        return width;  
    }  
    public double getHeight() {  
        return height;  
    }  
    public void showDims(){  
        System.out.println("Width: " + this.width  
        + " Height: " + this.height);  
    }  
}
```

Constructorii Triangle cheamă pe prima linie constructorii Shape corespunzători

```
public class Triangle extends Shape {  
    private String type;  
    public Triangle(){  
        super();  
        this.type = "";  
    }  
    public Triangle( double width,  
                    double height, String type){  
        super(width, height); // first  
        this.type = type;  
    }  
    public double area(){  
        return getWidth() * getHeight() / 2;  
    }  
    public String getType() {  
        return type;  
    }  
}
```



MOȘTENIREA ACCESUL LA MEMBRII SUPERCLASEI

- Există o formă de **super** (asemănătoare cu **this**) care referă membrii superclasei
- Nu putem accesa cu **super** membrii privați ai superclasei
- Forma generală
super.member
- **member** poate fi o metodă sau o variabilă a instanței
- Ca în cazul **this** putem accesa membrii din superclasă cu același nume cu cei din subclasă

```
public class A {  
    public int i;  
}
```

```
public class B extends A{  
    private int i;  
    public B(int i){  
        super.i = i;  
        this.i = i;  
    }  
}
```



MOȘTENIREA SUPRASCRIEREA METODELOR

- Când o metodă din subclasă returnează același tip și are aceeași semnătură cu o metodă din superclasă, atunci metoda din subclasă **suprascrîe** (override) metoda din superclasă
- Metoda din superclasă va fi ascunsă

```
public class A {  
    private int i, j;  
    public A(int i, int j){  
        this.i = i;  
        this.j = j;  
    }  
    public void show(){  
        System.out.println("i: " + this.i  
                           + " j: " + this.j);  
    }  
}
```

```
public class B extends A{  
    private int k;  
    public B(int i, int j, int k){  
        super(i, j);  
        this.k = k;  
    }  
    // overrides show() in A  
    public void show(){  
        System.out.println("k: " + k);  
    }  
}
```



MOȘTENIREA SUPRASCRIEREA METODELOR

- Se va utiliza versiunea metodei `show()` definită în B care suprascrie metoda `show()` din A

```
public class OverrideTest {  
    public static void main(String[] args) {  
        B b = new B(1, 2, 3);  
        b.show();  
    }  
}
```

Rezultat:
k: 3

```
public class B extends A{  
    private int k;  
    public B(int i, int j, int k){  
        super(i, j);  
        this.k = k;  
    }  
    // overrides show() in A  
    public void show(){  
        super.show();  
        System.out.println("k: " + k);  
    }  
}
```

Rezultat:
i: 1 j: 2
k: 3

MOȘTENIREA SUPRASCRIEREA METODELOR

- Prin **suprascriere** Java implementează polimorfismul la **rulare (runtime)**
- O variabilă **referință cu tipul superclasei** poate **referi un obiect al subclasei**
- La chemarea unei metode pe referința superclasei Java **determină în momentul execuției** pe baza obiectului referit ce variantă de metodă folosește
- Prin **combinarea moștenirii cu suprascrierea metodelor** o superclasă poate defini forma generală a metodelor folosite de toate subclasele ei

```
public class A {  
    private int i, j;  
    public A(int i, int j){  
        this.i = i;  
        this.j = j;  
    }  
    public void show(){  
        System.out.println("A.i: " + this.i  
                           + " A.j: " + this.j);  
    }  
}
```

MOȘTENIREA SUPRASCRIEREA METODELOR

```
public class B extends A{  
    private int k;  
    public B(int i, int j, int k){  
        super(i, j);  
        this.k = k;  
    }  
    // overrides show() in A  
    public void show(){  
        System.out.println("B.k: " + k);  
    }  
}
```

```
public class OverrideTest {  
    public static void main(String[] args) {  
        A a = new A(1,2);  
        a.show();  
        A b = new B(3, 4, 3);  
        b.show();  
    }  
}
```

Metoda show() chemată, se stabilește la rulare. În acest caz varianta din subclasă, deoarece variabila b primește o referință la un obiect de tip B

Rezultat:

A.i: 1 A.j: 2

B.k: 3



MOȘTENIREA CLASE ABSTRACTE

- Uneori este util ca o **superclasă să fie definită ca o formă generalizată** lăsând subclaselor să implementeze detaliile
- O astfel de clasă, **definește natura metodelor** pe care o subclasă trebuie să le implementeze **fără a le implementa ea însăși**
- **Metodele clasei abstracte sunt suprascrise de către metodele din subclase**
- O subclasă trebuie să suprascrie toate metodele superclasei definite ca metode **abstract**
- **Metoda abstractă nu are corp**
 - Forma generală a metodei abstracte
abstract type name(parameter-list);
 - O clasă care **conține una sau mai multe metode abstracte** se declară cu specificatorul **abstract**

MOȘTENIREA CLASE ABSTRACTE

```
public abstract class Shape {  
    private double width;  
    private double height;  
    private String name;  
    public Shape(double width,  
                double height, String name){  
        this.width = width;  
        this.height = height;  
        this.name = name;  
    }  
    public double getWidth() {  
        return width;  
    }  
    public double getHeight() {  
        return height;  
    }  
    public String getName() {  
        return name;  
    }  
    abstract public double area();
```

```
public class Triangle extends Shape {  
    public Triangle(double width, double height){  
        super(width, height, "triangle");  
    }  
    @Override  
    public double area() {  
        return super.getWidth() *  
               super.getHeight() / 2;  
    }  
}
```

Clasa abstractă **Shape** are metoda abstractă **area**, pe care subclasa **Triangle** trebuie să o suprascrie și implementeze

MOȘTENIREA CLASE ABSTRACTE

```
public class Rectangle extends Shape {  
  
    public Rectangle(double width, double height) {  
        super(width, height, "rectangle");  
    }  
  
    @Override  
    public double area() {  
        return super.getWidth() * super.getHeight();  
    }  
}
```

```
public class ShapeTest {  
    public static void main(String[] args) {  
        Shape t = new Triangle(4.0, 4.0);  
        System.out.println("Shape is a " + t.getName()  
                           + " and area is " + t.area());  
        Shape r = new Rectangle(4.0, 4.0);  
        System.out.println("Shape is a " + r.getName()  
                           + " and area is " + r.area());  
    }  
}
```

Rezultat:

Shape is a triangle and area is 8.0

Shape is a rectangle and area is 16.0



MOȘTENIREA CUVÂNTUL CHEIE FINAL

- Se folosește pentru:
 - a **împiedica moștenirea**
 - a **împiedica suprascrierea unei metode**
 - a **transforma variabilele în constante**

```
// final prevents inheritance
public final class A {
```

```
// ERROR Can't subclass A
public class B extends A {
```

```
// final prevents inheritance
public class A {
    // must initialize and cannot change
    private final int a = 5;
    // getter only
    public int getA() {
        return a;
    }
    public final void myMethod(){
        System.out.println("A final method.");
    }
}
```

```
public class B extends A {
    @Override
    public void myMethod(){
        // ERROR Cannot override
    }
}
```



MOȘTENIREA CLASA OBJECT

- Este o clasă specială care este implicit superclă pentru toate clasele din Java
- O variabilă referință de tip **Object** poate referi orice obiect al oricărei clase

Method	Purpose
<code>Object clone()</code>	Creates a new object that is the same as the object being cloned.
<code>boolean equals(Object object)</code>	Determines whether one object is equal to another.
<code>void finalize()</code>	Called before an unused object is recycled.
<code>Class<? extends Object> getClass()</code>	Obtains the class of an object at run time.
<code>int hashCode()</code>	Returns the hash code associated with the invoking object.
<code>void notify()</code>	Resumes execution of a thread waiting on the invoking object.
<code>void notifyAll()</code>	Resumes execution of all threads waiting on the invoking object.
<code>String toString()</code>	Returns a string that describes the object.
<code>void wait()</code> <code>void wait(long milliseconds)</code> <code>void wait(long milliseconds, int nanoseconds)</code>	Waits on another thread of execution.



PACHETE ȘI INTERFEȚE

PACHETE

- **Pachetele** (package) sunt alcătuite din grupuri de clase înrudite, permit organizarea codului și constituie un alt **mecanism de încapsulare**
- Clasele definite într-un pachet vor fi accesate prin numele pachetului
- **Pachetele contribuie la mecanismul de control al accesului**
- Clasele pot fi făcute private la nivel de pachet
- Numele unei clase reprezintă un **namespace**, clasele de până acum fiind create în namespace-ul atm.paradigms
- Deoarece namespace-ul global se aglomerează rapid soluția Java este crearea de pachete, deoarece numele pachetului devine parte a numelui clasei



PACHETE ȘI INTERFEȚE

PACHETE

- Pachetul se creează prin adăugarea pe prima linie a fișierului sursă a comenzi **package**
- Forma generală a declarației unui pachet
package pkg;
- **pkg** este numele pachetului
- **Java folosește sistemul de fișiere** pentru a gestiona pachetele
- Fiecare pachet este în directorul cu același nume (case sensitive)
- **Pachetele pot fi create ierarhic** prin separarea fiecărui nume de pachet cu punct
package pack1.pack2.pack3...packN;
- Evident trebuie create directoarele care să suporte ierarhia de pachete



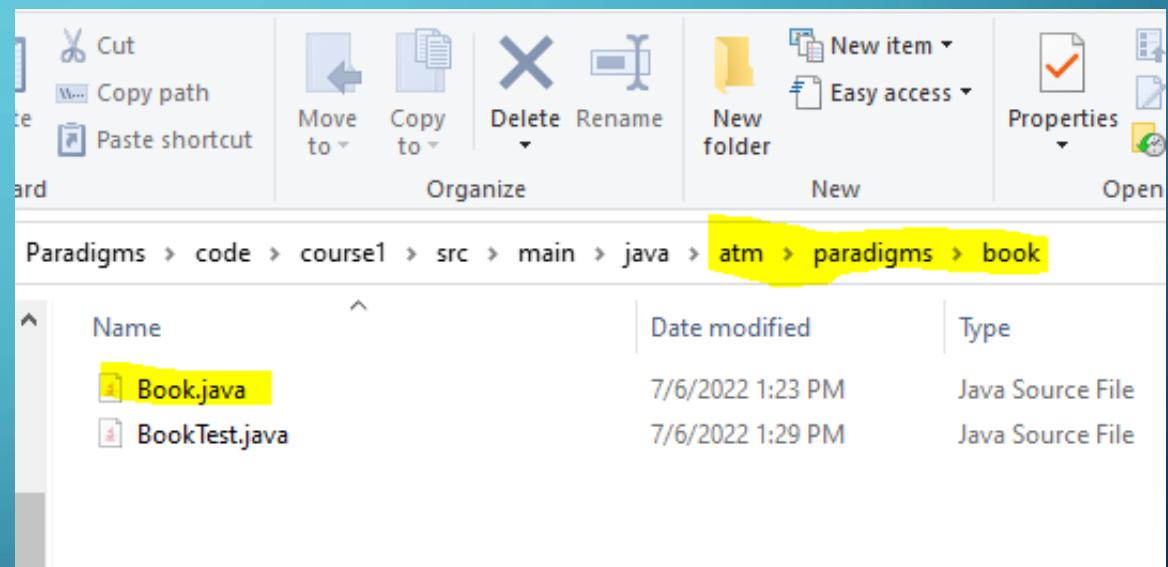
PACHETE ȘI INTERFEȚE PACHETE

```
package atm.paradigms.book;

public class Book {
    private String title;
    private String author;
    private int year;

    public Book(String title, String author,
               int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(year);
        System.out.println();
    }
}
```



PACHETE ȘI INTERFEȚE PACHETE

```
package atm.paradigms.book;

public class BookTest {
    public static void main(String[] args) {
        Book books[] = new Book[3];
        books[0] = new Book("Java: A Beginner's Guide",
                           "Schildt", 2005);
        books[1] = new Book("Java: The Complete Reference",
                           "Schildt", 2005);
        books[2] = new Book("The Art of Java",
                           "Schildt and Holmes", 2003);
        for (int i = 0; i < books.length; i++)
            books[i].show();
    }
}
```

```
& 'C:\Program Files\Java\jdk-11.0.12\bin\java.exe'
'@C:\Users\trnnc\AppData\Local\Temp\cp_6huarke646af882hrjupww0u
o.argfile' 'atm.paradigms.book.BookTest'
```

Java: A Beginner's Guide
Schildt
2005

Java: The Complete Reference
Schildt
2005

The Art of Java
Schildt and Holmes
2003

PACHETE ȘI INTERFEȚE

ACCESUL LA MEMBRII

- Specificatori de acces: ***private, public, protected***
- **Membrii publici** ai unei clase sunt **vizibili oriunde**, fără restricții, în alte clase și pachete
- **Membrii privați** sunt **vizibili numai pentru alți membrii ai clasei**
- **Membrii protejați** sunt accesibili **în propriul pachet dar și în subclase din alte pachete**
- **Membrii cu acces implicit** este **vizibil doar în pachet**
- Clasele au numai 2 nivele de acces:
 - **public** – vizibil oriunde
 - **implicit** – accesibil de alt cod numai din același pachet

PACHETE ȘI INTERFEȚE

ACCESUL LA MEMBRII

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

PACHETE ȘI INTERFEȚE

ACCESUL LA MEMBRII

```
package atm.paradigms.book;

public class Book {
    protected String title;
    protected String author;
    protected int year;
    public Book(String title, String author,
int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }
    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(year);
    }
}
```

Subclasa BookE are acces la membrii cu acces protejat (protected) ai superclasei

```
package atm.paradigms;
import atm.paradigms.book.Book;

public class BookE extends Book {
    private String publisher;
    public BookE(String title, String author, int year,
String publisher) {
        super(title, author, year);
        this.publisher = publisher;
    }
    @Override
    public void show() {
        super.show();
        System.out.println(this.publisher);
        System.out.println();
    }
    @Override
    public String toString() {
        return super.title + " " + super.author ;
    }
}
```

PACHETE ȘI INTERFEȚE

ACCESUL LA MEMBRII

```
package atm.paradigms.book;

import atm.paradigms.BookE;

public class BookTest {
    public static void main(String[] args) {
        BookE books[] = new BookE[3];
        books[0] = new BookE("Java: A Beginner's Guide",
                            "Schildt", 2005, "O'Reilly");
        books[1] = new BookE("Java: The Complete Reference",
                            "Schildt", 2005, "O'Reilly");
        books[2] = new BookE("The Art of Java",
                            "Schildt and Holmes", 2003, "O'Reilly");
        for (int i = 0; i < books.length; i++)
            books[i].show();
    }
}
```

Java: A Beginner's Guide
Schildt
2005
O'Reilly

Java: The Complete Reference
Schildt
2005
O'Reilly

The Art of Java
Schildt and Holmes
2003
O'Reilly



PACHETE ȘI INTERFEȚE

IMPORTAREA PACHETELOR

- Clasele din alte pachete pot fi **importate cu numele întreg – nume pachet, punct, nume clasă**
- Nu este o abordare practică, se folosește instrucțiunea **import**
- Clasele membre ale pachetului pot fi utilizate cu numele lor fără numele pachetului
- Forma generală:

import *pkg.classname*;

- ***pkg*** este numele calificat al pachetului, ***classname*** este numele clasei ce se importă
- Se poate **importa un întreg pachet** dacă se folosește asterisc (*) în loc de numele clasei



PACHETE ȘI INTERFEȚE INTERFEȚE

- **Interfețele definesc un set de metode ce vor fi implementate de o clasă**, dar nu le implementează ele
- Se separă interfața clasei de implementarea acesteia. Interfețele sunt sintactic similare claselor abstracte.
- **O clasă poate implementa mai multe interfețe**. Clasa stabilește detaliile implementării
- Forma generală a unei interfețe

```
access interface name {  
    ret-type method-name1(param-list);  
    type var1 = value;  
    // ...  
}
```

Metodele declară tipul returnat și semnatura. **Variabilele declarate sunt constante** ce trebuie initializare, fiind implicit **public**, **final** și **static**



PACHETE ȘI INTERFEȚE INTERFEȚE

- Pentru a implementa o interfață, în definiția clasei se include clauza **implements** și apoi se implementează metodele definite de interfață
- Forma generală:

```
access class classname extends superclass implements interface {
```

```
// class-body
```

```
}
```

- **access are valoarea public sau nu este definit.** Dacă se implementează mai multe interfețe numele acestora este separat cu virgulă
- Metodele care implementează o interfață trebuie să fie publice
- **Moștenirea se aplică și pentru interfețe folosind extends** (asemănător claselor)

PACHETE ȘI INTERFEȚE INTERFEȚE

```
package atm.paradigms;

public interface Series {
    // return next value
    public int getNext();
    // restart
    public void reset();
    // set start value
    public void setStart(int x);
}
```

```
package atm.paradigms.tests;
import atm.paradigms.Series;

public class Even implements Series {
    private int start;
    public Even() {
        this.start = 0;
    }
    @Override
    public int getNext() {
        this.start += 2;
        return this.start;
    }
    @Override
    public void reset() {
        this.start = 0;
    }
    @Override
    public void setStart(int x) {
        this.start = x;
    }
}
```

PACHETE ȘI INTERFEȚE INTERFEȚE

```
package atm.paradigms.tests;

public class EvenTest {
    public static void main(String[] args) {
        // interface as reference
        Series e = new Even();
        for (int i = 0; i < 3; i++)
            System.out.println("Next: " + e.getNext());
        e.reset();
        System.out.println("Next: " + e.getNext());
        e.setStart(100);
        System.out.println("Next: " + e.getNext());
    }
}
```

Rezultat:
Next: 2
Next: 4
Next: 6
Next: 2
Next: 102



TRATAREA EXCEPTIILOR

- Exceptiile sunt erori care se produc la execuția codului
- În Java exceptiile sunt clase derivate din clasa **Throwable**
- **Throwable** are 2 subclase **Exception** și **Error**
 - **Error** - tratează erori care se produc în mașina virtuală, nu în cod
 - **Exception** – erori care se produc în program
- Cuvinte cheie folosite în tratarea erorilor: **try**, **catch**, **throw** și **throws**
- Instrucțiunile programului pe care le monitorizăm în vederea prinderii exceptiilor sunt incluse în blocul **try**
- Tratarea exceptiilor are loc în blocul **catch**
- Pentru generarea manuală a exceptiilor se folosește **throw**
- Exceptiile generate în metode se specifică cu **throws**
- Instrucțiunile care trebuie executate obligatoriu la ieșirea din blocul **try** sunt puse în blocul **finally**

TRATAREA EXCEPTIILOR

FOLOSIREA TRY-CATCH

- Se folosesc împreună
- Forma generală:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExcepType1 exOb) {  
    // handler for ExcepType1  
}  
  
//..
```

- Pot fi mai multe instrucțiuni **catch**. Se execută cea corespunzătoare exceptiei generate în blocul **try**
- Dacă nu se generează nici o excepție blocul **try** se termină normal

TRATAREA EXCEPTIILOR

FOLOSIREA TRY-CATCH

```
package atm.paradigms.tests;

public class ExceptionTest {
    public static void main(String[] args) {
        int[] values = new int[4];
        try{
            System.out.println("try block starts");
            // generate exception
            values[10] = 10;
        } catch (ArrayIndexOutOfBoundsException e){
            // handle exception
            System.out.println("Exception: " + e.getMessage());
        }
        System.out.println("Execution continues!");
    }
}
```

Rezultat:

try block starts

Exception: Index 10 out of bounds for length 4

Execution continues!

TRATAREA EXCEPTIILOR

FOLOSIREA THROW

- În exemplul anterioare exceptia a fost generată automat de JVM
- Este posibil să se genereze manual exceptii folosind instrucțiunea **throw**
- Forma generală:
throw exceptOb;
- **exceptObj** este un obiect al unei clase exceptie derivată din **Throwable**

```
package atm.paradigms.tests;

public class ThrowTest {
    public static void main(String[] args) {
        try{
            System.out.println("try block starts");
            // manually throw exception
            throw new ArithmeticException("oops!");
        }
        catch (ArithmeticException e){
            // catch exception
            System.out.println("Exception info: "
                + e.getMessage());
        }
        System.out.println("After try/catch block.");
    }
}
```



TRATAREA EXCEPȚIILOR

DETALII THROWABLE

- Clauza **catch** are un parametru cu tipul excepției care primește obiectul excepție
- În clasa **Throwable** cele mai utile metode sunt:
 - *printStackTrace()* – afișează mesajul eroare precum și metodele invocate ce au condus la excepție
 - *getMessage()* – afișează mesajul de eroare standard
 - *toString()* - afișează excepția și mesajul de eroare standard

```
try{  
    System.out.println("try block starts");  
    // generate exception  
    values[10] = 10;  
} catch (ArrayIndexOutOfBoundsException e){  
    // handle exception  
    System.out.println("Exception: ");  
    e.printStackTrace();  
}
```

```
try block starts  
Exception:  
java.lang.ArrayIndexOutOfBoundsException:  
Index 10 out of bounds for length 4  
at  
atm.paradigms.tests.ExceptionTest.main(ExceptionTest.java:9)
```



TRATAREA EXCEPTIILOR

FOLOSIREA FINALLY

- Bloc de cod ce se execută la ieșirea din blocurile try/catch
- Se folosește pentru eliberarea unor resurse precum un fișier deschis sau o conexiune de rețea
- Forma generală:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExcepType1 exOb) {  
    // handler for ExcepType1  
}  
//...  
finally {  
    // finally code  
}
```



TRATAREA EXCEPTIILOR

FOLOSIREA THROWS

- Dacă o metodă generează o excepție pe care nu o tratează trebuie să o declare cu clauza **throws**
- Forma generală:

```
ret-type methName(param-list) throws except-list {
```

```
// body
```

```
}
```

- **except-list** este o listă de excepții separate cu virgulă pe care metoda le generează
- Subclasele **Error** și **RuntimeException** nu se specifică în lista **throws**
- Celelalte excepții trebuie declarate, altfel se generează eroare la compilare

TRATAREA EXCEPTIILOR

FOLOSIREA THROWS

```
ackage atm.paradigms.tests;

import java.io.IOException;

public class ThrowsTest {
    public static char prompt(String str) throws IOException{
        System.out.print(str + ": ");
        return (char) System.in.read();
    }

    public static void main(String[] args) {
        char ch;
        try{
            ch = prompt("Enter a letter: ");
            System.out.println("You pressed " + ch);
        } catch (IOException e){
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

TRATAREA EXCEPTIILOR

SUBCLASE EXCEPTII

- Crearea propriilor clase de exceptii se face prin moștenirea clasei **Exception**

```
package atm.paradigms.tests;

public class MyException
    extends Exception {
    public MyException(String s){
        super(s);
    }
}
```

```
package atm.paradigms.tests;

public class MyExceptionTest {
    public static void main(String[] args) {
        try{
            // throw exception
            throw new MyException("Programming Paradigms");
        } catch(MyException e){
            // catch
            System.out.println("Exception caught: " +
                e.getMessage());
        }
    }
}
```



CONVENȚII DE NUME

Tip	Regulă	Exemplu
Clasă	Începe cu literă mare și trebuie să fie un substantiv (Color, Button, System, Thread, etc.).	public class Employee { //code snippet }
Interfață	Începe cu literă mare și trebuie să fie un adjectiv (Runnable, Remote, ActionListener etc.)	interface Printable { //code snippet }
Metodă	Începe cu literă mică și trebuie să fie un verb (run(), print(), println() etc.)	class Employee{ void draw(){...} }
Variabilă	Începe cu literă mică și nu trebuie să înceapă cu caractere speciale (&, \$, _ etc.)	int id; String firstName; Employee employee;

Notă:

- Folosiți cuvinte în loc de acronime
- Dacă numele conține cuvinte multiple, primul cuvânt începe cu literă mare sau mică după caz, iar următoarele obligatoriu cu literă mare (FileInputStream, toUpperCase etc.)



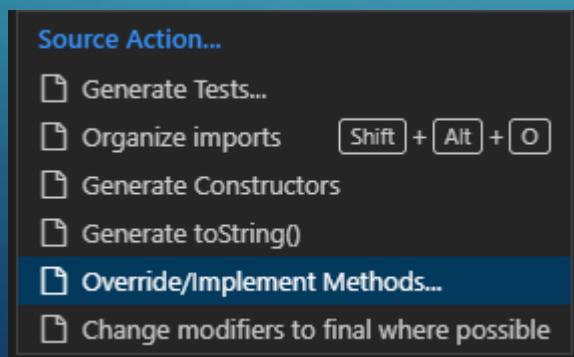
CONVENȚII DE NUME

Tip	Regulă	Exemplu
Pachet	Trebuie să conțină doar litere mici. Dacă este format din mai multe cuvinte acestea se separă cu punct (java.util, java.lang etc.)	package atm.paradigms ; class Employee { //code snippet }
Constante	Pentru constante se folosesc litere mari (RED, YELLOW). Dacă numele conține mai multe cuvinte acestea se separă cu caracterul underscore "_" (MAX_PRIORITY)	class Employee{ //constant static final int MIN_AGE = 18; //code snippet }



GHID DE STRUCTURARE COD

- Proiectele în cadrul cursului se creează **exclusiv cu Maven**
- Se respectă **convențiile de nume**
- În general **atributele vor fi private, iar metodele și constructorii vor fi publice**
- În general **clasele vor fi publice și implementate într-un fișier (.java) cu același nume**
- Folosiți funcționalitatea de formatare automată a textului din VS Code (Shift+Alt+F)
- Folosiți din meniul contextual opțiunea **Source Action ...** pentru generarea automată a codului



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