






Definire gli oggetti

-  Enums
-  Interfaces
-  Classi astratte
-  Inner Class
-  Anonymous Class

Java enum

- ❑ *Classe* speciale che rappresenta un gruppo di **costanti** (variabili non modificabili, come variabili final)
- ❑ Si usa la parola chiave **enum** per definirle al posto di **class**

```
enum Level {  
    LOW,  
    MEDIUM,  
    HIGH  
}
```
- ❑ Si accede a livello di classe

```
Level myVar = Level.MEDIUM;
```
- ❑ Ogni singola costante può associarsi a un insieme di parametri specifici
- ❑ I parametri vengono recepiti da specifici costruttori
- ❑ Possono essere definiti **attributi** e **metodi**

Java enum 2

```
1 package esempiVari;
2
3 public enum EnumLevelSimple {
4     LOW,
5     MEDIUM,
6     HIGH
7 }
8
```

<terminated> Enumerazioni [Java Application] C:\Program Files\

```
Value:HIGH
Ordinal: 2
Ordinal:0 Value:LOW
Ordinal:1 Value:MEDIUM
Ordinal:2 Value:HIGH
Enum value obtained by string 'MEDIUM':MEDIUM
```

```
1 package esempiVari;
2
3 public class Enumerazioni {
4     public static void main(String[] args) {
5         Enumerazioni enumerazioni = new Enumerazioni();
6         enumerazioni.run();
7     }
8
9     private void run() {
10         EnumLevelSimple e;
11         System.out.println("Value:" + EnumLevelSimple.HIGH);
12         System.out.println("Ordinal: " + EnumLevelSimple.HIGH.ordinal());
13
14         // Scan and print elements
15         for (EnumLevelSimple enumElement : EnumLevelSimple.values()) {
16             System.out.println("Ordinal:" + enumElement.ordinal() + " Value:" + enumElement.toString());
17         }
18
19         // Get enum from string
20         e = EnumLevelSimple.valueOf(EnumLevelSimple.class, "MEDIUM");
21         System.out.println("Enum object value obtained by string 'MEDIUM':" + e.toString());
22     }
23 }
24
```

Java enum advanced

```
1 package esempiVari;
2
3 public enum EnumLevelComplex {
4     LOW(11,"low-value"),
5     MEDIUM(34,"medium-value"),
6     HIGH("high-value");
7
8     // Instance variables
9     int counter;
10    String altText;
11
12    // Constructors
13    EnumLevelComplex(int counter, String altText) {
14        this.counter = counter;
15        this.altText = altText;
16    }
17    EnumLevelComplex(String altText) {
18        this.counter = 50;
19        this.altText = altText;
20    }
21
22    public int getCounter() {
23        return this.counter;
24    }
25
26    public String getAltText() {
27        return this.altText;
28    }
29 }
30
31
```

24/04/2023

```
1 package esempiVari;
2
3 public class EnumerazioniComplex {
4     public static void main(String[] args) {
5         EnumerazioniComplex enumerazioni = new EnumerazioniComplex();
6         enumerazioni.run();
7     }
8
9     private void run() {
10        EnumLevelComplex e;
11        System.out.println("Value:" + EnumLevelComplex.HIGH);
12        System.out.println("Ordinal: " + EnumLevelComplex.HIGH.ordinal());
13
14        // Scan and print elements
15        for (EnumLevelComplex enumElement : EnumLevelComplex.values()) {
16            System.out.println("Ordinal:"+enumElement.ordinal() + " Value:"+enumElement.toString());
17        }
18
19        // Get enum from string
20        e = EnumLevelComplex.valueOf(EnumLevelComplex.class, "MEDIUM");
21        System.out.println("Enum object value obtained by string 'MEDIUM':"+e.toString());
22
23        // Get info from specific enum value
24        System.out.println("Counter for MEDIUM:"+e.getCounter());
25        System.out.println("Alternative text for MEDIUM:"+e.getAltText());
26
27    }
28 }
29
```

ing. Giampietro Zedda

Java enum advanced 2

```
1 package esempiVari;
2
3 public class EnumerazioniComplex {
4     public static void main(String[] args) {
5         EnumerazioniComplex enumerazioni = new EnumerazioniComplex();
6         enumerazioni.run();
7     }
8
9     private void run() {
10         EnumLevelComplex e;
11         System.out.println("Value:" + EnumLevelComplex.HIGH);
12         System.out.println("Ordinal: " + EnumLevelComplex.HIGH.ordinal());
13
14         // Scan and print elements
15         for (EnumLevelComplex enumElement : EnumLevelComplex.values()) {
16             System.out.println("Ordinal:"+enumElement.ordinal() + " Value:"+enumElement.toString());
17         }
18
19         // Get enum from string
20         e = EnumLevelComplex.valueOf(EnumLevelComplex.class, "MEDIUM");
21         System.out.println("Enum object value obtained by string 'MEDIUM':"+e.toString());
22
23         // Get info from specific enum value
24         System.out.println("Counter for MEDIUM:"+e.getCounter());
25         System.out.println("Alternative text for MEDIUM:"+e.getAltText());
26
27     }
28 }
```

Console X Problems Debug Shell

<terminated> EnumerazioniComplex [Java Application] C:\Program Files\Java\jdk-17\

Value:HIGH

Ordinal: 2

Ordinal:0 Value:LOW

Ordinal:1 Value:MEDIUM

Ordinal:2 Value:HIGH

Enum object value obtained by string 'MEDIUM':MEDIUM

Counter for MEDIUM:34

Alternative text for MEDIUM:medium-value

Classi astratte

```
abstract class  
Base {  
    abstract int m();  
}
```

```
class Derivata  
    extends Base {  
        int m() {  
            return 1;  
        }  
}
```

```
Base b= new Derivata();  
System.out.println(b.m());
```

Interfacce

- ❑ Una classe astratta può contenere metodi non astratti
 - A beneficio delle proprie sottoclassi
- ❑ In alcuni casi, si vogliono definire metodi astratti **senza vincolare la gerarchia di ereditarietà** delle classi che li implementeranno
- ❑ Si utilizzano le **interfacce**:
 - Insiemi di metodi astratti e costanti (attributi **static final**)
 - Pubblici per definizione
- ❑ Una classe può **implementare** un'interfaccia
 - Fornendo il codice relativo a tutti i metodi dichiarati nell'interfaccia

Interfacce e tipi

- ❑ Analogamente alle classi, ogni interfaccia definisce un **tipo**
 - Un oggetto che implementa una data interfaccia ha come tipo **anche** il tipo dell'interfaccia
 - Un oggetto può implementare **molte** interfacce
 - Di conseguenza può avere molti tipi
- ❑ Si può verificare se un oggetto ha un dato tipo con l'operatore **"instanceof"**
 - `if (myObject instanceof Comparable) ...`

Esempio

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

```
public class Rettangolo extends Forma  
    implements Comparable {  
    public int compareTo(Object o) {  
        //codice relativo...  
    }  
    //altri attributi e metodi...  
}
```

Interfacce

- ❑ Una interfaccia è una completa «**abstract class**» usata per raggruppare metodi dal corpo vuoto
 - Deve essere implementata da una classe concreta
 - Il corpo del metodo dell'interfaccia è fornito dalla classe che la implementa

```
// interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void run(); // interface method (does not have a body)
}
```

Interface

```
// Interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}
```

Interfacce vuote

- ❑ Alcune interfacce **non hanno** metodi
 - Servono solo come “**marcatori**” o indicatori di tipo
 - Indicano che gli oggetti delle classi che le implementano godono di qualche **proprietà**

Interfacce vuote

```
public interface Cloneable {  
    //indica che è lecito usare,  
    // sulle istanze di questa classe  
    // il metodo Object.clone()  
}
```

```
public class Rettangolo extends Forma  
    implements Comparable , Cloneable {  
    public int compareTo(Object o) {  
        //codice relativo...  
    }  
    //altri attributi e metodi...  
}
```

Interfacce da Java 8

- ❑ Possibilità di avere metodi con del codice, non più astratti
 - Il corpo del metodo dell'interfaccia non è fornito dalla classe che la implementa ma può essere sovrascritto (override)
 - Il metodo può essere dichiarato **static** non può essere sovrascritto e viene richiamato come *NomeInterface.nomeMetodo*
 - Di fatto si può realizzare una sorta di ereditarietà multipla
- ❑ Introduzione del **Default** method
 - Si tratta di metodi *non astratti* che possono essere sovrascritti
 - Necessario per supportare le **functional** interfaces e **lambda**
 - Definisce una implementazione comune a tutte le classi che implementano l'interfaccia
 - Permette di aggiungere un metodo a una interfaccia esistente senza modificare le classi che già la implementano

Interfacce da Java 8 ...default

```
1 package esempiVari;
2
3 interface IDisplay{
4
5     //Default method
6     default void display(){
7         System.out.println("Hello from default method");
8     }
9
10    //Abstract method
11    void displayMore(String msg);
12
13 }
```

```
1 package esempiVari;
2
3 public class InterfaceWithDefault implements IDisplay{
4
5     //implementing abstract method
6     public void displayMore(String msg){
7         System.out.println(msg);
8     }
9
10    public static void main(String[] args) {
11        InterfaceWithDefault dm = new InterfaceWithDefault();
12
13        //calling default method
14        dm.display();
15
16        //calling abstract method
17        dm.displayMore("Hello from implemented method");
18    }
```

Console × Problems Debug Shell

<terminated> InterfaceWithDefault [Java Application] C:\Program Files\Java\jdk-17

Hello from default method

Hello from implemented method

Interfacce da Java 8 ...static

```
1 package esempiVari;
2
3 interface IDisplayStatic {
4     // Static field
5     static String staticString = "This is a string available accessed by static";
6     static int MAX = 10;
7
8     // Default method
9     default void display() {
10         int i;
11         System.out.println("Method with code");
12         for (i = 0; i < MAX; i++) {
13             System.out.println("i="+i);
14         }
15         System.out.println("Hello from default method");
16     }
17
18     // Abstract method
19     void displayMore(String msg);
20
21     // Static method
22     static void show(String msg) {
23         System.out.println(msg);
24     }
25 }
26
```

```
1 package esempiVari;
2
3 public class InterfaceWithStatic implements IDisplayStatic {
4     // implementing abstract method
5     public void displayMore(String msg) {
6         System.out.println(msg);
7     }
8
9     public static void main(String[] args) {
10         InterfaceWithStatic smt = new InterfaceWithStatic();
11
12         // calling default method
13         smt.display();
14
15         // calling abstract method
16         smt.displayMore("Hello from implemented abstract method");
17
18         // calling static method
19         IDisplayStatic.show("Hello Java from static method");
20
21         // Accessing a static field
22         System.out.println(IDisplayStatic.staticString);
23     }
24 }
```

```
Console X Problems Debug Shell
<terminated> InterfaceWithStatic [Java Application] C:\Program
Method with code
i=0
i=1
i=2
i=3
i=4
i=5
i=6
i=7
i=8
i=9
Hello from default method
Hello from implemented abstract method
Hello Java from static method
This is a string available accessed by static
```


Interfacce vs ereditarietà multipla

- ❑ In java non è prevista l'ereditarietà multipla da classi diverse
- ❑ Il progetto originale di interface prevedeva solo metodi astratti implementati dalla classe con **implements**
- ❑ L'implementazione di interfacce, con metodi **concreti**, non dichiarati abstract, permette di ereditare codice **senza una gerarchia fra classi**

Classi astratte

```
abstract class  
Base {  
    abstract int m();  
}
```

```
class Derivata  
    extends Base {  
        int m() {  
            return 1;  
        }  
}
```

```
Base b= new Derivata();  
System.out.println(b.m());
```

Classi astratte 1

- Le **classi astratte in Java** sono utilizzate per poter dichiarare caratteristiche comuni fra classi di una determinata gerarchia. Pur definendo il nome di un tipo,
- **Non puo essere istanziata**, analogamente a quanto accade per le interfacce e può avere campi non statici, metodi non pubblici, un costruttore,
- Insomma una classe a tutti gli effetti ma non istanziabile.
- La sua dichiarazione è caratterizzata dall'utilizzo della keyword **abstract**:

```
public abstract class A {  
    //campi  
    //metodi  
}
```

Classi astratte 2

- Una classe astratta può contenere o meno metodi astratti, ma una classe che contiene metodi astratti deve necessariamente essere dichiarata come astratta, i.e. facendo uso della keyword `abstract`.

```
public abstract class A {  
    //campi  
    //metodi  
}
```

Classi astratte 3

Errata

```
public class A {  
    public abstract void metodo();  
}
```

L'esempio che segue invece è corretto dal punto di vista sintattico, infatti la classe ha dichiarato al suo interno un metodo abstract e la classe è di tipo astratto.

```
public abstract class A {  
    //campi  
    private boolean visible=true;  
    //metodi  
    public boolean isVisible() { return this.visible; }  
    public abstract void metodo();  
}
```

Classi astratte 4

- Una classe astratta può dichiarare dei campi (che ne descrivono lo stato) e dei metodi (non astratti) che ne specificano il funzionamento
- Un metodo astratto è un metodo che può essere dichiarato ma per il quale non viene fornita una implementazione; la sintassi da utilizzare in generale è la seguente:

```
abstract <return type> nomeMetodo(lista dei parametri);
```

Classi astratte 5

- ***Una classe astratta non può in alcun modo essere istanziata***, quindi può essere utilizzata esclusivamente come **classe base**
- Quando estendiamo (o deriviamo) da una classe astratta, la **classe derivata deve fornire una implementazione per tutti i metodi astratti** dichiarati nella classe genitrice
- se così non dovesse essere, anche la sotto-classe deve essere dichiarata come *abstract*.

```
public abstract class AbstractProcess {  
    public void process() {  
        init();  
        baseAction();  
        stepAfter();  
    }  
    public void init() {  
        // metodo dichiarato direttamente  
        // all'interno della classe astratta  
    }  
    public abstract void baseAction();  
    // metodo astratto la cui implementazione  
    // è demandata alla sottoclasse  
    public void stepAfter() {  
        // metodo dichiarato direttamente  
        // all'interno della classe astratta  
    }  
}  
  
class MyProcess extends AbstractProcess {  
    @Override  
    public void baseAction() {  
        // implementazione del metodo baseAction()  
    }  
}
```


Classi/interface nidificate (Inner)

Si tratta di classi o interfacce definite dentro altre classi (outer)

- ❑ **Inner class**
- ❑ **Local inner class**
- ❑ **Inner interface**
- ❑ **Static inner class**
- ❑ **Anonymous class**, classi ed interfacce che sono definite ed istanziate contemporaneamente alla loro definizione, all'interno di classi e metodi.

Classi nidificate (Inner)

```
class Enclosing {  
    // ...  
    class Inner {  
        //...  
    }  
    //...  
    static class InnerStatic {  
        //...  
    }  
    interface InnerInterface {  
        //...  
    }  
    // ...  
    public void metod(...) {  
        //...  
        class LocalClass {  
            // ...  
        }  
        // ...  
    }  
}
```

Classi nidificate (Inner)

- ❑ Per fare riferimento ad una classe inner si deve usare il nome della classe ospitante (outer) seguito da un punto e dal nome della classe:

```
Enclosing.Inner  
Enclosing.InnerInterface  
Enclosing.InnerStatic
```

}

Classi local Istanziamento

- ❑ Per istanziare una classe inner **non statica** deve essere prima ottenuta una istanza della classe enclosing, con una sintassi piuttosto sorprendente:

```
Enclosing enclosing = new Enclosing(...);  
Enclosing.Inner v = enclosing.new Inner(...);
```

- ❑ Mentre per le **static** inner classes è possibile utilizzare la comune sintassi dell'operatore new:

```
Enclosing.InnerStatic v = new Enclosing.InnerStatic(...);
```

Inner Class

- ❑ Può essere definita **ovunque** nel corpo di una classe ma **all'esterno** del corpo dei metodi (altrimenti diventa local)
- ❑ Una Inner Class **ha completa visibilità** di tutti i membri della classe ospitante (outer), anche di quelli privati, di fatto è una **estensione** della classe ospitante
- ❑ Si può istanziare direttamente da un metodo della classe ospitante (outer) o dall'esterno, avendo prima a disposizione **una istanza** della classe ospitante

```
OuterInnerClass myOuter = new OuterInnerClass();  
OuterInnerClass.InnerClass myInner = myOuter.new InnerClass();
```

- ❑ La classe ospitante (outer) **non ha visibilità** degli attributi della inner class
- ❑ Si può utilizzare come **classe di servizio** per raggruppare nella stessa struttura più variabili, magari da passare come parametro in un metodo

Inner Class

```
package esempiVari;
public class OuterInnerClass {
    InnerClass ic;
    private int x = 5; // Visible in InnerClass
    int y = 20;

    // Inner class
    class InnerClass {
        int y = 10;
        int z = 20;
        void innerClassMethod() {
            System.out.println("From innerClassMethod " + (x + y));
        }
    }

    public void newInnerClassInstance() {
        ic = new InnerClass();
        ic.y = 100;
        // z = 5; // NOT visible
        System.out.println("Created instance of InnerClass");
        System.out.println("Set y To 100");
    }

    public int getX() {
        return x;
    }
}
```

Inner Class

```
package esempiVari;|
public class OuterInnerClassTest {

    public static void main(String[] args) {

        OuterInnerClass myOuter = new OuterInnerClass();
        OuterInnerClass.InnerClass myInner = myOuter.new InnerClass();
        System.out.println("From Outer Class " + (myInner.y + myOuter.getX()));
        myInner.innerClassMethod();
        myOuter.newInnerClassInstance();
        System.out.println("From Outer Class " + (myInner.y + myOuter.getX()));
    }
}
```

```
From Outer Class 15
From innerClassMethod 15
Created instance of InnerClass
Set y To 100
From Outer Class 15
|
```

Static Inner Class

- ❑ Può essere definita in qualsiasi punto della classe ospitante (outer) ma non all'interno di un metodo
- ❑ Essendo **static non** può accedere direttamente ai membri della classe ospitante (outer) ma può farlo attraverso un riferimento alla stessa
- ❑ Utile in caso di **composizione stretta** (es. outer=Testata e static inner=Dettaglio) per avere le classi nello stesso sorgente
- ❑ Di fatto una classe Static Inner Class è una classe indipendente da quella che lo ospita (outer)
- ❑ L'istanziamento viene fatto in modo **standard**:

```
OuterStaticInnerClass.InnerStatic  
    myInnerStatic = new OuterStaticInnerClass.InnerStatic();
```


Static Inner Class

```
package esempiVari;

public class OuterStaticInnerClass {

    int x = 5;  // Not Visible in InnerStatic

    // Static Inner class
    static class InnerStatic {
        int z = 10;

        protected void innerStaticMethod(OuterStaticInnerClass osic) {
            int y=30;
            // x=40;          // Error
            System.out.println("From Static Class by reference X=" + osic.x);
        }
    }
}
```

Static Inner Class

```
package esempiVari;  
public class OuterStaticInnerClassTest {  
  
    public static void main(String[] args) {  
  
        OuterStaticInnerClass myOuter = new OuterStaticInnerClass();  
        System.out.println("From Outer Class X=" + myOuter.x);  
        OuterStaticInnerClass.InnerStatic  
            myInnerStatic = new OuterStaticInnerClass.InnerStatic();  
        myInnerStatic.innerStaticMethod(myOuter);  
    }  
}
```

From Outer Class X=5

From Static Class by reference X=5

Local Inner Class

- ❑ Si può definire all'interno dell'ambito di qualsiasi blocco
 - All'interno di un blocco definito da un metodo
 - All'interno del corpo di un ciclo for
 - ...
- ❑ Una Local Inner Class **ha completa visibilità** di tutti i membri della classe ospitante (outer), *anche di quelli privati*, di fatto è una **estensione** della classe ospitante
- ❑ Si utilizza per definire attributi e comportamenti utili solo all'interno di un blocco o di un metodo specifico

Local Inner Class

```
package esempiVari;

public class OuterInnerClassLocal {
    private int x = 5; // Visible in InnerClass
    int y = 20;        // Visible in InnerClass

    public void testInnerClassLocal() {
        class InnerClass {
            int z = 10;
            void display() {
                System.out.println("From LocalinnerClass " + (x + y));
            }
        }
        InnerClass inner = new InnerClass();
        inner.display();
    }
}
```

```
package esempiVari;

public class OuterInnerClassLocalTest {

    public static void main(String[] args) {
        OuterInnerClassLocal myOuter = new OuterInnerClassLocal();
        myOuter.testInnerClassLocal();
    }
}

<terminated> OuterInnerClassLocalTest [Java]
From LocalinnerClass 25
```

Classi Anonime

- ❑ Sono classi "**locali**" senza un nome assegnato, definite e istanziate un'unica volta attraverso una singola espressione
- ❑ La sintassi per definire una classe anonima è identica a quella utilizzata per la chiamata di un **costruttore**, con successivo blocco di codice
- ❑ Classe anonima a partire da una **classe da estendere**

```
new ClassName ( [ argument-list ] ) { class-body }
```

- ❑ Classe anonima a partire da una **interfaccia da implementare**

```
new InterfaceName () { class-body }
```

Classi Anonime

- ❑ Rendono il codice molto più **conciso**
- ❑ Dichiarazione e istanziazione **contestuale**
- ❑ Utile per utilizzo **estemporaneo** una sola volta nel codice

```
public interface TitledName {  
    public String femaleTitle(String name);  
    public String maleTitle(String name);  
}
```

- ❑ Con istanza di oggetto anonimo si ha semplicemente:

```
BaseTitle englishTitle= new TitledName() {  
    @Override  
    public String femaleTitle(String name) {  
        return "Ms "+name;  
    }  
    @Override  
    public String maleTitle(String name) {  
        return "Mister "+name;  
    }  
};
```

Classi Anonime

- Normalmente si utilizzano classi anonime a partire da interfaces predefinite per esempio per scopi di sorting

```
Comparator<String> cmp = new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return o1.compareTo(o2);  
    }  
};
```

Classi Anonime punti chiave

- ❑ Dichiarazione attraverso l'operatore **new**
- ❑ **Interfaccia** da implementare o **classe** da estendere
- ❑ Parentesi con gli argomenti per il **costruttore**, proprio come una classica espressione di istanziamento di una classe
- ❑ All'interno del corpo di definizione della classe anonima sono **consentiti metodi**, ma non istruzioni singole
- ❑ Quando si istanzia anonimamente una interfaccia **tutti i metodi** devono essere implementati
- ❑ In una classe anonima si possono dichiarare **campi e metodi**
- ❑ Si utilizzano le classi anonime **per passare del codice come parametro**

Classi Parametriche (Generics)

- ❑ Letteralmente il termine *generics* significa *tipi parametrizzati*
- ❑ Permettono di creare classi, interface e metodi dove il **tipo di dato** su cui operano viene specificato come **parametro**
- ❑ Una classe, interfaccia o metodo che operi su un tipo parametrizzato viene chiamata **generica**, come classe generica o metodo generico
- ❑ Java ha sempre dato la possibilità di creare classe, interface e metodi generalizzati operando con riferimenti di tipo ***Object***, senza però la sicurezza del tipo
- ❑ I generics aggiungono la sicurezza del tipo, **senza cast espliciti**
- ❑ Con i generics tutti i cast sono automatici e impliciti

Esempio di Generics

```
// Qui, T è un parametro di tipo
// che verrà sostituito da un tipo reale
// quando viene creato un oggetto di tipo Gen
public class Gen<T> {
    T ob;    // Dichiarare un oggetto di tipo T

    // Passa al costruttore un riferimento
    // a un oggetto di tipo T
    public Gen(T ob) {
        super();
        this.ob = ob;
    }
    // Restituisce ob
    public T getOb() {
        return ob;
    }
    // Mostra il tipo di T
    public void showType() {
        System.out.println("Il tipo di T è "
            + ob.getClass().getName());
    }
}
```

Esempio di Generics

```
// Dimostra la classe generica
public class GenDemo {
    public static void main(String[] args) {
        Gen<Integer> iOb; // Crea un riferimento di Gen per Integer

        // Crea e assegna oggetto Gen<Integer>
        // Utilizza autoboxing
        iOb = new Gen<Integer>(88);
        // Mostra il tipo di dati di iOb
        iOb.showType();
        // Ottiene il valore senza casting
        int v = iOb.getOb();
        System.out.println("value: " + v);
        System.out.println();

        // Crea un riferimento di Gen per String
        Gen<String> strOb = new Gen<String>("Generics Test");
        // Mostra il tipo di dati di strOb
        strOb.showType();
        // Ottiene il valore ancora senza casting
        String str = strOb.getOb();
        System.out.println("value: " + str);
        System.out.println();
    }
}
```

<terminated> GenDemo [Java Application] C:\Program Files\Java\jdk-17\bin\j

Il tipo di T è java.lang.Integer
value: 88

Il tipo di T è java.lang.String
value: Generic Test

Generics

- ❑ T è il nome di un parametro di tipo

```
class Gen<T>
```

- ❑ Successivamente viene utilizzato per dichiarare l'oggetto ob

```
T ob;
```

- ❑ L'oggetto reale, Integer o String viene passato al costruttore

```
public Gen(T ob) {  
    this.ob = ob;  
}
```

- ❑ Il Parametro T specifica anche il tipo di ritorno del metodo

```
public T getOb() {  
    return ob;  
}
```

Generics

- ❑ Il metodo ***getClass*** su **ob** viene definito da **Object** e quindi è disponibile su tutti i tipi di classe

```
ob.getClass().getName());
```

- ❑ **Integer** è un argomento di tipo passato al parametro di tipo di **Gen**, **T**. Tutti i riferimenti a **T** vengono trasformati in **Integer**.

```
Gen<Integer> iOb;
```

- ❑ Chiamando il costruttore **Gen** si specifica anche l'argomento di tipo **Integer**

```
iOb = new Gen<Integer>(88);
```

- ❑ **iOb** è di tipo **Integer**, il controllo avviene a livello di compilazione

```
9 iOb = new Gen<Double>(88.0); // Errore
```

Generics

❑ Funzionano solo con gli **oggetti** e non con i dati primitivi

```
9 iOb = new Gen<int>(88); // Errore
```

❑ Aumentano la **sicurezza** del tipo (iOb e strOb sono sempre di tipo Gen<T>)

```
28 iOb=strOb; // Errore
```

❑ Assicurano **automaticamente** la sicurezza del tipo di tutte le operazioni che riguardano Gen, cosa non possibile con Object e casting in modo automatico

Generics con più parametri di tipo

```
// Classe generica con due parametri
// di tip T e V
public class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Passa al costruttore un riferimento
    // a un oggetto di tipo T e di tipo V
    public TwoGen(T ob1, V ob2) {
        this.ob1 = ob1;
        this.ob2 = ob2;
    }

    // Mostra i tipi di T e V
    public void showTypes() {
        System.out.println("Il tipo di T è "
            + ob1.getClass().getName());
        System.out.println("Il tipo di V è "
            + ob2.getClass().getName());
    }
}
```

```
    T getObj1() {
        return ob1;
    }
    V getObj2() {
        return ob2;
    }
}
```

Generics con più parametri di tipo

```
package esempiVari.generics;
```

```
// Dimostra la classe generica con due tipi
```

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

```
        TwoGen<Integer, String> tgObj =  
            new TwoGen<Integer, String>(88, "Generics");
```

```
        // Mostra i tipi
```

```
        tgObj.showTypes();
```

```
        // Ottiene e mostra i valori
```

```
        int v = tgObj.getObj1();
```

```
        System.out.println("value: " + v);
```

```
        String str = tgObj.getObj2();
```

```
        System.out.println("value: " + str);
```

```
    }
```

```
}
```

<terminated> TwoGenDemo [Java Application] C:\Program Files\Java\jdk-

Il tipo di T è java.lang.Integer

Il tipo di V è java.lang.String

value: 88

value: Generics

Generics con tipi limitati

- ❑ Non sempre i parametri di tipo si possono sostituire con qualsiasi tipo di classe
- ❑ A volte è utile limitare i tipi che possono essere passati
- ❑ Per esempio è impossibile creare una classe generica che possa calcolare la media di un array di qualsiasi tipo di numero, int, double e float, con una sola variabile come parametro T, usando il metodo **doubleVar()**
- ❑ Il compilatore non ha modo di sapere che si vuole usare T riferendosi solo a dati numerici e bisogna garantire inoltre che vengano passati **solo dati numerici**
- ❑ Per garantire questa situazione Java offre i tipi limitati, ovvero creare un limite superiore che dichiara la classe da cui derivare tutti gli argomenti di tipo
<T **extends** *superclass*>
- ❑ Questo specifica che T può essere sostituito solo da *superclass* o dalle sue *sottoclassi*

Generics con tipi limitati

```
// L'argomento di tipo per T deve essere
// Number o una classe derivata da Number
public class Stats<T extends Number> {
    T[] nums; // Array di Number o sottoclasse

    // Passa al costruttore un riferimento
    // a Number o sua sottoclasse
    public Stats(T[] ob) {
        this.nums = ob;
    }

    // Restituisce il tipo double in tutti i casi
    public double average() {
        double sum = 0.0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i].doubleValue();
        }
        return sum/nums.length;
    }
}
```

Poiché il tipo T è ora limitato da Number, il compilatore Java sa che tutti gli oggetti di tipo T possono richiamare `doubleValue()`, in quanto dichiarato da Number

Generics con tipi limitati

```
// Dimostra la classe generica con due tipi
public class StatsDemo {
    public static void main(String[] args) {
        Integer[] inums = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is : " + v);

        Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is : " + w);

        // String[] strs = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs);
        // double x = sob.average();
        // System.out.println("sob average is : " + w);
    }
}
```

La parte commentata dà errore di compilazione..String NON è sottoclasse di Number

<terminated> StatsDemo [Java Application] C:\

iob average is : 3.0
dob average is : 3.3

Generics con argomenti wildcard

- ❑ Il metodo `sameAverage` così specificato **NON** può funzionare sempre perché se per esempio l'oggetto chiamante è di tipo `Stats<Integer>` anche il parametro `ob` deve essere di tipo `Stats<Integer>`

```
// Stabilisce se due medie sono uguali
boolean sameAverage(Stats<T> ob) {
    if (average() == ob.average())
        return true;
    return false;
}
```

- ❑ Per ovviare a questo problema `sameAverage` si codifica con **wildcard** ?

```
// Stabilisce se due medie sono uguali
boolean sameAverage(Stats<?> ob) {
    if (average() == ob.average())
        return true;
    return false;
}
```

Generics con argomenti wildcard

```
3 // Dimostra la classe generica con due tipi
4 public class WildcarDemo {
5     public static void main(String[] args) {
6         Integer[] inums = { 1, 2, 3, 4, 5 };
7         Stats<Integer> iob = new Stats<Integer>(inums);
8         double v = iob.average();
9         System.out.println("iob average is : " + v);
10
11         Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
12         Stats<Double> dob = new Stats<Double>(dnums);
13         double w = dob.average();
14         System.out.println("dob average is : " + w);
15
16         Float[] fnums = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
17         Stats<Float> fob = new Stats<Float>(fnums);
18         double x = fob.average();
19         System.out.println("fob average is : " + x);
20     }
```

```
21 // Si osservi quali array hanno la stessa media
22 System.out.println("Averages of iob and dob");
23 if (iob.sameAverage(dob)) {
24     System.out.println("Are the same");
25 } else {
26     System.out.println("Are different");
27 }
28 System.out.println("Averages of iob and fob");
29 if (iob.sameAverage(fob)) {
30     System.out.println("Are the same");
31 } else {
32     System.out.println("Are different");
33 }
34 }
35 }
36 }
<terminated> WildcarDemo [Java Application]
iob average is : 3.0
dob average is : 3.3
fob average is : 3.0
Averages of iob and dob
Are different
Averages of iob and fob
Are the same
```

Generics con argomenti wildcard

❑ Gli argomenti wildcard possono essere limitati esattamente come un parametro di tipo

void showCoordinate <Coord<?>

❑ Utile soprattutto quando si crea un tipo generico su una gerarchia di classi attraverso *l'argomento wildcard limitato*. La clausola ***extends*** stabilisce un limite superiore che il ? può controntare

void showCoordinate <Coord<? extends threeDim>

❑ Il tentativo di richiamare *showCoordinate()* con un riferimento *Coord<TwoD>* provocherà un errore di compilazione

Generics ... non solo classi

- ❑ I generics non sono applicabili solo alle classi ma anche a:
 - *Metodi*
 - *Costruttori*
 - *Interfacce*
- ❑ Si possono definire **gerarchie** di classi generiche
- ❑ Si possono definire **sottoclassi generiche** di classi **NON** generiche
- ❑ E' possibile **trasformare un'istanza** di una classe generica in un'altra solo se sono **compatibili** e i due argomenti di tipo sono gli stessi
- ❑ Il compilatore **trasforma** le classi generiche in classi eseguibili con il meccanismo della **cancellazione**.
- ❑ Durante il runtime **non esiste** nessun parametron di tipo (Object e casting)
- ❑ I parametri di tipo **NON** possono essere istanziati
- ❑ Una classe generica **non può estendere Throwable** quindi non si possono creare classi di eccezione generiche