



UNIVERSITÀ
DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA
E INFORMATICA**

Interfacce Grafiche e Programmazione ad Eventi

Carmine Dodaro

Anno Accademico 2019/2020

Sintassi

```
public class Father {  
    ...  
}  
public class Son extends Father {  
    ...  
}
```

- In Java, una classe può estendere **una sola** altra classe.
- I metodi/campi **public** della classe padre (superclasse) sono automaticamente metodi **public** della classe figlio (sottoclasse).
- I metodi/campi **protected**, sono accessibili **solo all'interno** della classe figlio.
- I metodi/campi **private** della classe padre **non sono accessibili** dai metodi della classe figlio.

Classe Object

- In Java, tutte le classi **estendono implicitamente** la classe predefinita `Object`.
- La classe `Object` è istanziabile e contiene implementazioni di default per alcuni **metodi di utilità generale**.
- Ogni oggetto Java dispone di tali metodi automaticamente. Ad esempio:
 - `boolean equals(Object o);`
 - `String toString();`
 - `int hashCode();`
 - ...

TestObject.java

```
public class TestObject {
    private int value;

    public TestObject(int v) {
        value = v;
    }

    public static void main(String [] args) {
        TestObject t1 = new TestObject(1);
        TestObject t2 = new TestObject(2);
        //Output: TestObject@6d06d69c
        System.out.println(t1.toString());
        //Output: TestObject@7852e922
        System.out.println(t2);
    }
}
```

Ridefinizione (overriding) e binding dinamico

- Una sottoclasse può **ridefinire** i metodi della superclasse.
- La firma del metodo è la stessa.
- A differenza del C++ non si deve specificare la keyword **virtual**.
- La JVM stabilisce **dinamicamente** sulla base della classe della particolare istanza quale metodo invocare (**binding**).

Ridefinizione di toString()

TestObject2.java

```
public class TestObject2 {  
    private int value;  
  
    public TestObject2(int v) {  
        value = v;  
    }  
  
    public String toString() {  
        return "Value: " + value;  
    }  
  
    public static void main(String[] args) {  
        TestObject2 t1 = new TestObject2(1);  
        TestObject2 t2 = new TestObject2(2);  
  
        System.out.println(t1.toString()); // Value: 1  
        System.out.println(t2);          // Value: 2  
    }  
}
```

Binding dinamico e polimorfismo: esempio

TestObject3.java

```
public class TestObject3 {  
    public static void main(String[] args) {  
        Object[] arrayMisto = new Object[2];  
        TestObject2 t = new TestObject2(1);  
        String s = "Stringa";  
        arrayMisto[0] = t;  
        arrayMisto[1] = s;  
  
        for(int i = 0; i < arrayMisto.length; i++) {  
            System.out.println(arrayMisto[i]);  
        }  
    }  
}
```

Output

```
Value: 1  
Stringa
```

Binding dinamico e polimorfismo: considerazioni

- Negli esempi, **Object** è superclasse di **TestObject** e di **TestObject2**.
- Riferendosi agli oggetti di **TestObject** e di **TestObject2** come se fossero **Object**, si possono utilizzare su di essi **tutti e soli** i metodi definiti in **Object**.
- Se i metodi sono stati ridefiniti (come nel caso di **TestObject2**) il binding dinamico invocherà il metodo corrispondente.
- Il **polimorfismo** consente a porzioni di codice di essere:
 - **indipendenti** dagli specifici oggetti utilizzati, ma
 - **specifici** nel comportamento in fase di esecuzione.

Super

- Nel costruttore si può usare `super(...)` per invocare il costruttore della superclasse
- Per invocare un metodo della superclasse si può utilizzare `super.nomeMetodo()`

Invocare metodi della super classe

Superclasse.java e Sottoclasse.java

```
class Superclasse {
    private int value;
    public Superclasse(int value) { this.value = value; }
    public int getValue() { return value; }
}

class Sottoclasse extends Superclasse {
    public Sottoclasse(int value) {
        super(value);
    }

    public int getValue() {
        return super.getValue()+1;
    }

    public static void main(String[] args) {
        Sottoclasse s = new Sottoclasse(1);
        System.out.println(s.getValue()); //Stampa: 2
    }
}
```

Elementi costanti

In Java non esiste il concetto di `const` come in C++, tuttavia esiste la keyword **final** che può essere utilizzata per campi, metodi e classi:

- I campi dichiarati `final` non possono essere modificati
- I metodi dichiarati `final` non possono essere ridefiniti nelle sottoclassi
- Le classi dichiarate `final` non possono essere estese

Campi final

```
public class ProvaFinal {  
    private final String stringa = "STRINGA COSTANTE";  
  
    ProvaFinal() {  
        System.out.println(stringa);  
        stringa = "ciao"; // Errore  
    }  
  
    public static void main(String[] args) {  
        ProvaFinal prova = new ProvaFinal();  
    }  
}
```

Ereditarietà e conversione di tipo

```
// 1) Upcast: sempre OK!
Object number1 = new TestObject(1);
// 2) Downcast: genera errore
TestObject number2 = number1;
// 3) Downcast: può dare errore se il cast non è permesso
TestObject number3 = (TestObject) number1;
// 4) Checked downcast: OK!
if (number1 instanceof TestObject) {
    TestObject number4 = (TestObject) number1;
}
```

- 1) La conversione da **sottoclasse** a **superclasse**, detta **upcast**, è automatica.
- 2) La conversione automatica da **superclasse** a **sottoclasse**, detta **downcast**, genera errore.
- 3) La conversione esplicita potrebbe dare errore in fase di esecuzione se i due tipi non sono **compatibili** tra loro.
- 4) Il modo corretto di effettuare un downcast è quello di verificare la compatibilità di tipo con il costrutto **instanceof**.

Definizione

```
public abstract class BaseClass {  
    ...  
    public abstract ... abstractMethod( ... );  
  
    public ... concreteMethod( ... ) {  
        ...  
    }  
}
```

- Una classe è astratta (abstract) se **almeno uno** dei suoi metodi è dichiarato tale.
- I metodi astratti sono solamente **dichiarati**, ma non vengono definiti.
- Una classe astratta non è **istanziabile** (non si può creare un oggetto della classe).
- L'implementazione dei metodi astratti **deve** essere **realizzata dalle sottoclassi**, a meno che non siano anch'esse astratte.

Definire un'interfaccia

```
public interface Declarator {  
    <dichiarazione di costanti>  
    <dichiarazione di metodi>  
}  
  
public class Implementor implements Declarator {  
    ...  
}
```

- Un'interfaccia **dichiara** un insieme di **costanti** e **metodi**.
- Una classe che **implementa** un'interfaccia deve rispettarne il **contratto** e definirne tutti i metodi.
- Una classe può implementare **più di una** interfaccia.

- Le classi definiscono la **sintassi** e la **semantica** degli oggetti, ossia il comportamento.
- Una classe **astratta** delega una parte della definizione del comportamento alle sottoclassi concrete.
- Un'interfaccia definisce un **contratto** di programmazione a cui gli implementatori si conformano.
- Le classi **puramente astratte** sono realizzabili in Java, ma sono sotto tutti gli aspetti meno vantaggiose delle interfacce.

Ereditarietà vs. implementazione

```
public interface Interfaccia1 { ... }  
public interface Interfaccia2 { ... }  
  
public class Superclasse { ... }  
  
public class Prova extends SuperClasse implements Interfaccia1 ,  
    Interfaccia2 { ... }
```

- Le seguenti dichiarazioni sono **tutte lecite**

```
Superclasse obj1 = new Prova(...);  
Interfaccia1 obj2 = new Prova(...);  
Interfaccia2 obj3 = new Prova(...);
```

- Sull'oggetto **obj1** potrò invocare tutti e soli i metodi di **Superclasse**.
- Sull'oggetto **obj2** potrò invocare tutti e soli i metodi di **Interfaccia1**.
- Sull'oggetto **obj3** potrò invocare tutti e soli i metodi di **Interfaccia2**.

Classi anonime

- Le classi anonime permettono di rendere il codice più conciso
- Permettono allo stesso tempo di dichiarare e istanziare una classe
- Sono utili nel caso una classe è usata una sola volta

Classi anonime

- Le classi anonime permettono di rendere il codice più conciso
- Permettono allo stesso tempo di dichiarare e istanziare una classe
- Sono utili nel caso una classe è usata una sola volta

File MyInterface.java e Prova.java

```
public interface MyInterface {
    public void ciao();
}

public class Prova {
    public static void main(String [] args) {
        MyInterface m = new MyInterface () {
            public void ciao () {
                System.out.println ("prova");
            }
        };
        m.ciao ();
    }
}
```

Classe Pair.java

```
public class Pair < T1, T2 > {  
    public T1 first;  
    public T2 second;  
  
    public Pair(T1 first , T2 second) {  
        this.first = first;  
        this.second = second;  
    }  
}
```

- Le classi possono essere definite sulla base di **uno o più parametri** relativi al **tipo** dei dati che trattano.
- Il **nome** e il **numero** dei parametri sono dichiarati nell'intestazione della classe (**T1** e **T2** nell'esempio **Pair**).
- Il **nome** dei parametri può essere utilizzato in luogo del nome di classi all'interno della definizione (**first** e **second** nel costruttore di **Pair**).

Classe Pair2.java

```
public class Pair2 {  
    public Object first;  
    public Object second;  
  
    public Pair2(Object first , Object second) {  
        this.first = first;  
        this.second = second;  
    }  
}
```

- Le classi generiche in Java sono **costrutti sintattici** per gestire una forma di **polimorfismo dinamico**.
- I tipi parametrici vengono gestiti tramite riferimenti a Object.
- Per questo motivo non si possono utilizzare i tipi base!

Esempio

```
//Si può utilizzare lo stesso tipo
Pair<String , String> p1 = new Pair<String , String>("str1", "str2");

//Si possono utilizzare tipi diversi
Pair<String , Scanner> p1 = new Pair<String , Scanner>("str1", null);

//Non si possono utilizzare tipi base!
Pair<int , int> p1 = new Pair<int , int>(1,2); //← errore
```

Metodi generici

```
public class Prova {
    public void printArrayString(String[] array) {
        for(String s : array)
            System.out.println(s);
    }

    public void printArrayComplex(Complex[] array) {
        for(Complex c : array)
            System.out.println(c);
    }

    public <T> void printArray(T[] array) {
        for(T t : array)
            System.out.println(t);
    }

    public static void main(String[] args) {
        Prova p = new Prova();
        p.printArrayString(args);
        p.printArray(args);
    }
}
```

Classi Wrapper

- Per ogni tipo primitivo in java, esiste una classe corrispondente, chiamata classe Wrapper
- È utile in tutti quei contesti in cui non si possono utilizzare i tipi base (ad esempio con le classi generiche)
- Come la classe String sono **immutabili**
- Quali sono:
 - 1 **byte** → Byte
 - 2 **short** → Short
 - 3 **int** → Integer
 - 4 **long** → Long
 - 5 **float** → Float
 - 6 **double** → Double
 - 7 **char** → Character
 - 8 **boolean** → Boolean

Come si usano le Classi Wrapper?

Come si usano?

```
public class Prova {  
    public static void main(String [] args) {  
        int num1 = 5;  
        Integer num2 = new Integer(5);  
        Integer num3 = new Integer(num2);  
        Integer num4 = num1;  
        Integer num5 = num4;  
        num5++;  
        System.out.println(num4); // Output: 5  
        System.out.println(num5); // Output: 6  
    }  
}
```