Introduzione alla programmazione Java
corso di Sistemi Operativi (IEL/IDT) a.a. 2002/2003
Ing. Jürgen Assfalg

Una **classe** rappresenta un tipo di dato complesso, che prevede attributi e metodi. Istanziando una classe, cioè creando un **oggetto** di una determinata classe, si ottiene una struttura che contiene sia i dati, sia le funzionalità necessarie per operare su questi. All'insieme dei dati contenuti in un oggetto, ovvero nei suoi attributi, si fa in genere riferimento con il termine di stato.

In Java, la sintassi per la definizione di una classe è la seguente:

```
[modificatori] class nome_classe  {
      corpo_della_classe
}
```

Possibili modificatori sono quelli di visibilità (`public, protected, private, package`), nonché `abstract` e `final`. La sintassi per la definizione degli attributi, all' interno del corpo di una classe, è:

```
[modificatori] tipo nome [=valore_predefinito];
```

dove agli usuali modificatori di visibilità si aggiungono i modificatori `static` e `final`. Il tipo può essere un tipo primitivo del linguaggio Java (`int`, `byte`, `float`, `double`, `boolean`, ecc.) oppure una classe o un' interfaccia (per quest' ultima, vedi nel seguito). Infine, la definizione dei metodi deve aderire alla seguente sintassi:

```
[modificatori] tipo_di_ritorno nome_metodo ([argomenti]) {
    corpo_del_metodo
}
```

con gli stessi modificatori applicabili agli attributi, e cioè `static` e `final`, oltre ai modificatori di visibilità.

Attraverso i modificatori di visibilità si riescono a realizzare gli obiettivi di **incapsulamento** dei dati e delle implementazioni, facendo sì che all'esterno siano disponibili solo le funzionalità necessarie per interagire con un determinato oggetto, mentre l' effetiva rappresentazione dei dati, le implementazioni ed altre funzionalità (accessorie) sono note solo a chi ha sviluppato la classe.

Si supponga allora di voler definire una classe Java per rappresentare un giocatore, ad esempio di una squadra di calcio. Per semplicità, si consideri che il giocatore sia caratterizzato da un nome e dal numero di goal segnati in campionato. Si prevedono inoltre dei metodi atti a leggere le suddette informazioni, nonché ad aggiornarle (si prevede la possibilità di aggiornare il numero di goal segnati, con il vincolo che non possano diminuire). Il codice della classe potrebbe allora essere il seguente:

```
public class Player
{
      private String name = null;
      private int goals = 0;
```

```
	public Player( String n )
	{
		name = n;
	}

	public String getName()
	{
		return name;
	}

	public int getGoals()
	{
		return goals;
	}

	public void incGoals()
	{
		goals++;
	}

	public String toString()
	{
		return new String( name +
			" has scored " + goals + " goals." );
	}
}
```

Altra importante caratteristica della programmazione ad oggetti è il **polimorfismo**, per cui possono esistere più metodi con lo stesso nome che differiscono solo per il numero ed il tipo dei parametri. Questa proprietà potrebbe essere sfruttata per definire un nuovo metodo `incGoals` che consenta di incrementare il numero di reti segnati per più di un' unità alla volta:

```
	public void incGoals( int g )
		throws IllegalArgumentException
	{
		if ( g <= 0 )
			throw new IllegalArgumentException(
				"argument goals must be > 0" );
		else
			goals += g;
	}
```

Le **eccezioni** in Java rappresentano un meccanismo utilizzato per segnalare condizioni di errore. Quando viene generata un' eccezione, l' esecuzione si interrompe ed il controllo ritorna a livelli via via superiori, finché non viene raccolta e gestita. Questo consente la gestione asincrona delle condizioni di errore, ovvero non si richiede che ogni singola condizione di errore sia gestita là dove si verifica, a tutto vantaggio della leggibilità del codice.

Per verificare il corretto funzionamento della classe, si può scrivere un semplice programma di test:

```
public class TestPlayer
{
	public static void main( String[] args )
	{
		if ( 2 != args.length )
		{
			System.err.println(
				"java TestPlayer <name> <goals>"
```

```
                );
                System.exit( 1 );
            }
            Player p = new Player( args[ 0 ] );
            try
            {
                p.incGoals( Integer.parseInt( args[ 1 ] ) );
            }
            catch ( IllegalArgumentException iae )
            {
                System.err.println( "Exception occurred: " +
                    iae.toString()
                );
            }
            finally
            {
                System.out.println( p.toString() );
            }
        }
}
```

Si osservi, in questa seconda classe, la particolare sintassi del metodo `main`, che è il metodo principale che viene invocato dall' interprete Java quando si richiede l' esecuzione di un programma. Gli argomenti passati a linea di comando sono contenuti nel vettore di stringhe `args`. Dal codice del programma si evince anche la sintassi per istanziare un oggetto attraverso l' operatore `new` e l' assegnazione del valore restituito ad una variabile. Si noti anche il blocco `try .. catch .. finally ..`: l' invocazione di un metodo che può generare un eccezione deve avvenire all' interno del blocco `try`, e nel corrispondente blocco `catch` ha luogo l' eventuale gestione dell' eccezione; infine, nel blocco `finally` (facoltativo) è contenuto del codice che verrà comunque eseguito, anche nel caso in cui si sia verificata un' eccezione (in genere in questo blocco si rilasciano le risorse eventualmente acquisite).

Salvando la prima classe in un file dal nome `Player.java` e la seconda in un file dal nome `TestPlayer.java`, dalla cartella in cui si trovano i due files è possibile compilare le classi con il comando:

```
javac -classpath . nome_classe.java
```

e quindi eseguire il programma con il comando

```
java -classpath . TestPlayer
```

L'**ereditarietà** è un costrutto dei linguaggi orientati agli oggetti che permette di definire nuove classi sulla base di classi esistenti, ereditando da queste attributi e metodi. In particolare, la classe derivata avrà accesso diretto ad attributi e metodi con visibilità `public` e `protected`, ma non a quelli con visibilità `private`, accessibili alla sola classe entro la quale sono definiti (tralasciamo per il momento la visibilità di tipo `package`). Questa caratteristica, presente anche in Java, può risultare utile per definire una nuova classe per gestire le particolarità dei portieri: i portieri, come gli altri giocatori, possono segnare goal, ma possono anche parare i tiri in porta, oppure possono non pararli! A questo scopo si definisce una nuova classe `GoalKeeper` che discende da `Player`, ereditandone attributi e metodi, e ve ne aggiunge altri, per considerare le particolarità dei portieri:

```
public class GoalKeeper extends Player
```

```
{
	private int saves = 0;
	private int  failedSaves = 0;

	public GoalKeeper( String n )
	{
		super( n );
	}

	public void incSaves()
	{
		saves++;
	}

	public void incFailedSaves()
	{
		failedSaves++;
	}

	public String toString()
	{
		return new String (
			super.toString() +
			System.getProperty( "line.separator" ) +
			"He succeded in " + saves +
			" saves, and failed in " + failedSaves + "."
		);
	}
}
```

Si noti che, oltre all' aggiunta di nuovi attributi e metodi, è stata ridefinito il metodo `toString()`, precedentemente definito in `Player` (in realtà il metodo è definito per ogni classe che discende da `Object`, ed in Java tutte le classi discendono, seppur implicitamente, da `Object`; per cui, anche nel caso di `Player`, si trattava in realtà di una **ridefinizione**). La possibilità di ridefinire i metodi nelle classi derivate è un' altra significativa caratteristica dei linguaggi ad oggetti, e viene solitamente indicata con il termine inglese *overriding*.

Anche questa volta, per verificare il corretto funzionamento della classe, è possibile introdurre una classe di test, come la seguente:

```
public class TestGoalKeeper
{
	public static void main( String[] args )
	{
		if ( 1 != args.length )
		{
			System.err.println(
				"\tjava TestGoalKeeper <name>" );
			System.exit( 1 );
		}
		GoalKeeper gk = new GoalKeeper( args[ 0 ] );
		gk.incGoals();
		gk.incSaves();
		gk.incSaves();
		gk.incFailedSaves();
		System.out.println( gk.toString() );
	}
}
```

Per raccogliere tutti i giocatori di una squadra possiamo definire una nuova classe

`Team`. Tuttavia, per gestire l' insieme dei giocatori, ci appoggeremo alle API di Java che, nel package `java.util`, forniscono un insieme di classi adatte alla creazione e manipolazione di collezioni di oggetti. La classe `Team` dovrà soddisfare almeno due requisiti: da un lato dovrà consentire il popolamento della squadra, e dall' altro dovrà restituire la somma dei goal segnati dai suoi giocatori. Una possibile implementazione per questa classe è la seguente:

```
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;

public class Team
{
	private Collection players = null;
	private String name = null;

	public Team( String n )
	{
		name = n;
		players = new LinkedList();
	}

	public void addPlayer( Player p )
	{
		players.add( p );
	}

	public int getGoals()
	{
		int goals = 0;
		for( Iterator i = players.iterator(); i.hasNext(); )
		{
			Player p = (Player) i.next();
			goals += p.getGoals();
		}
		return goals;
	}

	public String toString()
	{
		StringBuffer sb = new StringBuffer();
		String new_line = System.getProperty( "line.separator" );
		sb.append( "Composition of team \"" + name +
			"\":" + new_line
		);
		for( Iterator i = players.iterator(); i.hasNext(); )
		{
			Player p = (Player) i.next();
			sb.append( p.toString() + new_line );
		}
		return sb.toString();
	}
}
```

La classe prevede, fra l' altro, un attributo di tipo `Collection`: consultando la documentazione Java si scopre facilmente che si tratta di un'**interfaccia** (`interface`); un' interfaccia specifica un insieme di metodi senza peraltro fornirne un' implementazione. Una classe Java può implementare un' interfaccia fornendo un' implementazione per i metodi specificati da quest' ultima. Si osservi che in Java una singola classe può implementare più interfacce. Nel programma riportato sopra, la classe `LinkedList` implementa l' interfaccia `Collection`, e pertanto una sua istanza

può essere assegnata ad una variabile di tipo `Collection`. Per una descrizione di `Collection` e `LinkedList`, così come della classe `StringBuffer` (anch' essa contenuta in `java.util`) si faccia riferimento alla documentazione delle API Java.

Qui di seguito è riportato un programma (`TestTeam`) che, facendo uso delle classi precedentemente definite, crea un' istanza di una squadra e la popola con un certo numero di istanze di giocatori (giocatori "qualunque" e portieri):

```
public class TestTeam
{
      public static void main( String[] args )
      {
            Team t = new Team( "Squadrone" );
            try
            {
                  Player p = new GoalKeeper( "Johnny Portiere" );
                  p.incGoals( 1 );
                  t.addPlayer( p );
                  p = new Player( "Gino Difensore" );
                  p.incGoals( 3 );
                  t.addPlayer( p );
                  p = new Player( "Marc Attaccante" );
                  p.incGoals( 8 );
                  t.addPlayer( p );
            }
            catch ( IllegalArgumentException iae )
            {
            }
            System.out.println( t.toString() );
            System.out.println(
                  "The team has scored " + t.getGoals() +
                  " goals."
            );
      }
}
```

Si osservi come, nel corpo del programma, alla variabile `p`, di tipo `Player`, vengano indistintamente assegnati oggetti che sono istanze sia della classe `Player`, sia della classe `GoalKeeper`; questo è assolutamente lecito, poiché in virtù dell' ereditarietà una classe derivata può essere sempre usata in luogo di una classe da cui discende (direttamente o indirittamente).