ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

2: Classi e Tipi (1/2)

CLASSI E TIPI

In Java, l'unità principale di organizzazione del codice è la *Classe*.

Ogni oggetto appartiene necessariamente ad una *Classe*, che rappresenta il suo *Tipo* e determina la struttura del suo stato ed il codice che opera su tale stato.

CLASSI E PACKAGE

Una classe in Java è dichiarata con la parola chiave `class` seguita dal nome e dalla definizione.

```
class App {

}
```

Per convenzione, le classi Java sono denominate in *Pascal Case* con l'iniziale maiuscola.

Una classe appartiene ad un *Package*, che permette di organizzare le classi in gruppi gerarchici.

```
package it.unipd.app2020;

class App {

}
```

La parola chiave `package` se presente **deve** essere la prima riga di codice del file;

Se la classe pubblica `App` viene definita all'interno del package `it.unipd.app2020`, il suo file sorgente **deve** trovarsi all'interno della directory `it/unipd/app2020` rispetto al CLASSPATH:

```
it/unipd/app2020/App.java
```

Per convenzione, i *package* sono denominati con nomi di dominio, in ordine inverso, per es.:

- *org.apache.commons*
- *com.oracle.jdbc*

I package `java`, `javax` sono riservati.

VISIBILITÀ

Una classe non può usare un'altra classe qualsiasi: deve averne visibilità e, in certi casi, deve dichiarare l'intenzione di usarla.

Una configurazione della JVM può impedire l'accesso a determinati insiemi di classi per motivi di sicurezza; essendo questo controllo applicato al runtime, il codice può compilare regolarmente, ma fallire con un errore durante l'esecuzione.

VISIBILITÀ DI DEFAULT

In mancanza di indicazioni, una classe è visibile da parte di tutte le classi dello stesso package, ma non dalle classi al di fuori di esso.

Una classe può usare una qualsiasi altra classe all'interno dello stesso package senza indicazioni particolari.

VISIBILITÀ PUBBLICA

Una classe dichiarata `public` è visibile da qualsiasi altra classe caricata dalla JVM.

```
package it.unipd.app2020;

public class App {

}
```

Un file sorgente può contenere **al più una** classe pubblica, e **deve** chiamarsi come la classe contenuta.

Usando la direttiva `import` si aggiunge al file sorgente il nome della classe importata, che può quindi essere richiamato senza essere prefissato dal package.

```
package it.unipd.app2020;

import it.unipd.pcd2019.Util;

public class App {
  Util a;
}
```

STRUTTURA

Una classe può contenere:

- variabili
- metodi
- altre classi
- blocchi di codice anonimi

VARIABILI

Una classe può contenere diverse variabili che definiscono la struttura dello stato di ciascun oggetto della classe.

```
package it.unipd.app2020;

public class App {
  int a;
  String b;
}
```

Una variabile viene dichiarata con il nome del suo tipo, il suo nome e un punto e virgola a chiudere la dichiarazione.

Può essere presente un'espressione che inizializza la variabile con un valore.

Più variabili dello stesso tipo possono essere dichiarate di seguito separando i nomi con una virgola.

```java
package it.unipd.app2020;

public class App {
  int a, a2 = 5, a3 = 7;
  String b;
}
```

Le variabili si dividono principalmente in due categorie:

- *statiche*: ne esiste una sola copia, legata alla classe.
- *di istanza*: ogni oggetto ha la propria e fa parte del suo stato.

```java
package it.unipd.app2020;

public class App {
  static char c;
  int a;
  String b;
}
```

Per la loro somiglianza con delle variabili globali, le variabili statiche devono essere usate con particolare cautela, sopratutto se sono mutabili.

La vita delle variabili statiche è legata alla vita della classe; quella delle variabili di istanza alla vita di ciascun oggetto.

Le variabili statiche vengono allocate ed inizializzate nel momento in cui la classe viene caricata dal `ClassLoader` e preparata per l'uso.

Questo può avvenire, a volte, in momenti sorprendenti.

Le variabili hanno più classi di visibilità:

- `public`: possono essere lette e scritte da ogni classe
- `protected`: possono essere lette e scritte da classi che estendono la classe
- default: possono essere lette e scritte da classi del package
- `private`: possono essere lette e scritte solo da codice della classe

```java
package it.unipd.app2020;

public class App {
  public static char c;
  int a;
  protected String internal;
  private boolean secret;
}
```

Modificatore	Classe	Package	Sottoclasse	Univ
`public`	✅	✅	✅	✅
`protected`	✅	❌	✅	❌
nessuno	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Altri modificatori che influiscono sulle variabili:

- `final`: il valore della variabile non può essere modificato dopo l'assegnamento; inoltre, un valore **deve** essere assegnato alla costruzione.
- `transient`: la variabile va ignorata in sede di serializzazione
- `volatile`: la variabile ha un comportamento particolare in relazione all'accesso concorrente

Le variabili hanno sempre un nome in Camel Case con l'iniziale minuscola.

Fanno eccezione le variabili `static final`: esse sono, a tutti gli effetti, costanti; il loro nome si scrive solitamente in MAIUSCOLO, con parole eventualmente separate da underscore "_".

METODI

Una classe organizza il codice in Metodi.

Un metodo è definito da:

- **alcuni modificatori** (opzionali)
- **un tipo di ritorno** (richiesto, con una eccezione)
- **un nome** (minuscolo, con una eccezione)
- **un elenco di parametri** (richiesto)
- **un elenco di eccezioni** (opzionale)
- **un blocco di codice da eseguire** (opzionale)

```java
package it.unipd.app2020;

public class App {
  public App() { };

  int apply(char d) { return 0; }

  static boolean prepare(String target, int count)
    throws RuntimeException {
    return false;
  }
}
```

Una classe può avere uno o più metodi denominati come la classe stessa che sono detti *costruttori*.

Un *costruttore* viene chiamato quando si richiede la creazione di un oggetto della classe.

La tupla formata da:

- nome del metodo
- parametri di tipo
- elenco dei tipi degli argomenti

è detta *firma* (*signature*) del metodo. Una classe non può avere più metodi con la stessa firma.

Il tipo di ritorno `void` indica al compilatore che il metodo non ritorna nessun risulato.

Se il metodo ha un tipo di ritorno, il compilatore considera un errore la presenza di un percorso del codice in cui non venga ritornato al chiamante nessun valore, o un valore di tipo non compatibile con quello dichiarato.

Un metodo dichiarato `static` è legato alla classe: non può essere richiamato su di un oggetto, e non ha accesso alle variabili di istanza.

I metodi seguono le stesse classi di visibilità delle variabili: `public`, `protected`, `default`, `private`.

I metodi vengono richiamati con la notazione `valore.nomemetodo(parametri)` dove `valore` è un oggetto della classe che li ha definiti, oppure la classe stessa per i metodi statici.

Un costruttore viene richiamato con la parola chiave `new`, e ritorna un oggetto della classe.

```
boolean p = App.prepare("baz", 1);
App a = new App();
app.apply('z');
```

Una classe ha sempre un costruttore.

Se non ha un costruttore dichiarato esplicitamente, il compilatore genera un costruttore *di default*, privo di argomenti.

Attenzione che *se* un costruttore viene dichiarato, non viene generato quello di default.

ECCEZIONI

La gestione degli errori in Java è implementata con un sistema di Tipi di Eccezione.

Le Eccezioni sono oggetti, ma vengono creati ed usati in casi particolari, e sono supportate da apposita sintassi.

Al tempo in cui Java è stato ideato ed inizialmente implementato, si trattava di una scelta al passo con la disciplina dell'OOP dell'epoca.

Oggi l'effettiva utilità dei tipi di eccezione è molto dibattuta. Molte metodologie di Functional Programming le evitano o cercano di confinarle nel minor uso possibile.

Tutte le eccezioni derivano dalla classe `Throwable`.
Una prima suddivisione avviene fra le due sottoclassi di `Throwable`:

- `Exception`: gli errori nonostante i quali il programma dovrebbe essere in grado di proseguire
- `Error`: gli errori dai quali il programma non è in grado di proseguire.

Una particolare sottoclasse di `Exception` è `RuntimeException`: essa rappresenta ogni errore che può avvenire durante la normale valutazione di espressioni.

Viene lanciata direttamente dalla JVM, e quindi non necessita di essere dichiarata.

Eccezioni derivate da `RuntimeException` e `Error` sono dette *unchecked exceptions* e non necessitano dichiarazione nella definizione di un metodo.

Tutte le altre, discendenti da `Exception` o `Throwable` direttamente, sono dette *checked exception* e **devono** essere dichiarate nella definizione di un metodo.

La disciplina di OOP che ha ispirato questa parte di Java incoraggia la definizione di classi di eccezione legate al dominio del problema che il programma rappresenta, per esplicitare maggiormente il significato di tali condizioni di errore.

Questo approccio è oggi molto dibattuto, e nella pratica creare eccezioni di dominio è una decisione che va accuratamente ponderata.

CLASSI INTERNE

Una classe può dichiarare come membro una o più classi; queste vengono dette *Nested Classes*.

Come variabili e metodi possono essere statiche o meno, e una delle quattro visibilità.

La visibilità si comporta in modo analogo agli altri casi.

Classi interne statiche e non hanno invece comportamenti profondamente differenti, tanto da avere due nomi distinti: *static nested classes* per le prime, e *inner classes* le seconde

STATIC NESTED CLASSES

Una classe `static` non ha un accesso privilegiato ai membri (statici o meno) della classe ospite.

Nella pratica, una classe interna `static` è una normale classe di package che ha un nome sintatticamente prefissato da quello della classe ospite.

```java
package it.unipd.app2020;

public class App {
  static class Foo {
    int a;
  }

  static String s;
}
```

INNER CLASSES

Una classe interna non statica è una parte dello stato di un oggetto del tipo ospite. Quindi, ha lo stesso ciclo di vita, ed ha un riferimento privilegiato all'oggetto ospitante.

Non può dichiarare membri `static` ma solo membri di istanza.

```java
package it.unipd.app2020;

public class App {
  class Bar {
    int a;
  }

  String s;
}

App a = new App();
App.Bar b = a.new Bar();
```

Le classi *static nested* sono spesso legate a qualche design pattern (per es. Builder o Factory). Sebbene possano anche essere scritte al di fuori della classe ospitante, può essere più chiaro a volte includerle per rendere più apparente il legame esistente.

Le classi *inner* sono il segnale di un modello dati particolarmente complesso. Usatele con particolare cautela.

INIZIALIZZATORI

In una classe possono essere contenuti anche blocchi di codice anonimi.

Vengono eseguiti in sede di *inizializzazione* di una classe o di un oggetto.

INIZIALIZZATORI STATICI

I blocchi di inizializzazione dichiarati `static` vengono eseguiti, in ordine lessicale, al caricamento della classe.

```java
package it.unipd.app2020;

public class App {

  static String s = "foo";
  static int l;

  static {
    l= s.length();
  }

}
```

L'uso dei blocchi statici non è comune, ma nemmeno inconsueto.

Va posta attenzione a scrivere codice veloce, che non possa fallire, e legato strettamente alla preparazione della classe per l'uso.

INIZIALIZZATORI DI ISTANZA

I blocchi di inizializzazione privi di indicazioni sono eseguiti, in ordine lessicale, durante la creazione di ciascuna istanza di oggetto della classe.

In particolare, sono eseguiti *dopo* il supercostruttore ma *prima* di *qualsiasi* costruttore.

```java
package it.unipd.app2020;

public class App {

  String s = "foo";
  int l, j;

  App(int param) {
    j = param;
  }

  { l= s.length(); }
}
```

Valgono, ancora di più, le note per gli inizializzatori statici.

Inoltre, le interazioni con l'ereditarietà ed i costruttori raccomandano ancora maggiore cautela.

Scrivere codice che dipende dall'ordine di inizializzazione delle classi o delle istanze è una ricetta sicura per ottenere errori inattesi nei momenti meno opportuni.

EREDITARIETÀ

Java, come linguaggio OO, mette a disposizione un meccanismo di ereditarietà singola: una classe può avere una sola superclasse, di cui eredita codice e (parte) dello stato.

Una sottoclasse ha accesso ai membri pubblici, `package` e `protected` della superclasse, ma non ai membri `private`.

```
package it.unipd.app2020;

class App {

  private int a;
  protected int b;

}

class Foo extends App {
  private String c;
}
```

Limitandosi all'ereditarietà singola, Java ha evitato (in passato) il "Diamond Problem": è immediatamente individuabile a quale classe della gerarchia fornisce l'implementazione di un metodo.

Parte dei vantaggi dell'ereditarietà multipla viene recuperata con altri meccanismi.

Una sottoclasse è anche un *sottotipo* della classe che estende. Può cioè essere usata in ogni posto in cui viene richiesta la classe superiore.

Per costruzione, tutti i metodi in Java sono "virtual" nel senso che ha il termine in C++. Vale a dire, il codice che realmente viene eseguito alla chiamata di un metodo è noto con certezza esclusivamente al runtime.

Una classe dichiara di essere sottoclasse di un'altra con la parola chiave `extends` dopo il nome della classe.

Una classe dichiarata `final` non può essere usata come superclasse; non è possibile derivarne una sottoclasse.

```java
package it.unipd.app2020;

final class App {

  private int a;
  protected int b;

}

// Errore di compilazione:
class Foo extends App {
  private String c;
}
```

Una classe dichiarata `abstract` *deve* essere usata come superclasse; non è possibile istanziarla direttamente.

```java
package it.unipd.app2020;

abstract class App {
  private int a;
  protected int b;
}

class Foo extends App {
  private String c;
}

App app = new App(); // Errore di compilazione
App foo = new Foo(); // OK
```

Sebbene classicamente l'ereditarietà nasca come metodo principe per il riuso del codice e per l'organizzazione dei tipi nell'OOP, già il GOF evidenziava i limiti di questo approccio suggerendo maggiore enfasi sull'uso della *composizione* in tutti i casi in cui ciò sia possibile.

All'ereditarietà vengono lasciate solo quelle casistiche che le competono più strettamente.

Tutti gli oggetti in Java discendono implicitamente da `java.lang.Object`, ereditandone alcuni metodi fondamentali:

- `int hashCode()`
- `boolean equals(Object o)`
- `String toString()`

LINK INTERESSANTI

COMMAND LINE HEROES

https://www.redhat.com/en/command-line-heroes