

Java SE: OOP

- Principi di programmazione Object Oriented
- Override e overload
- Ereditarietà
- Interfacce
- Classi astratte
- Progetto di riferimento
 - <https://github.com/egalli64/jse> (*modulo 6*)

Principi OOP

- **Incapsulamento**, raggruppamento di dati e funzionalità in una classe
 - Astrazione: selezione tra possibili membri in base al problema particolare
 - Coesione: si mira a mantenere una forte correlazione interna
 - Data hiding: i dati sono (normalmente) visibili solo internamente (privati)
 - Interfaccia: le funzionalità sono (normalmente) visibili esternamente (pubbliche)
- **Ereditarietà** in gerarchie di classi
 - Dal generale (super) al particolare (sub) – DRY: don't repeat yourself
- **Polimorfismo**
 - Una interfaccia, molti metodi (override)

Classe

- Blocco che incapsula dati e funzionalità
- Normalmente usato come modello per oggetti
 - Un oggetto, a sua volta è l'istanza di una classe
- Può essere usato come semplice raccolta di metodi (*namespace*)
 - Esempio: classe Math
- Le classi “normali” (non nested) possono essere visibili
 - A chiunque ha accesso al package: public
 - Solo alle classi del package corrente: (default)

Pet
- name: string
+ eat(): void

Access modifier per data member

- Aiuta l'incapsulamento
 - **private**: accesso limitato ai metodi della classe
- Dubbio
 - protected: pensato per gerarchie di classi
 - Come private, con in più i metodi delle classi derivate
 - Ma anche package!
 - Una sub può violare l'incapsulamento di una sua super?
- Normalmente sconsigliati
 - package (default)
 - Tutti i metodi delle classi dello stesso package
 - pubblico
 - Chi vede la classe può accederlo

```
public class Pet {  
    private String name;  
    protected double weight;  
    static int count;  
    // public long doNotDoThis;  
  
    // ...  
}
```

<i>Pet</i>
- name: string # weight: double ~ <u>count: int</u>
- increaseWeigth(): void ~ eat(): void + <u>getCount(): int</u> + getName(): string

Access modifier per metodi

- Uso normale
 - **public**: accesso dal resto del mondo
- Casi particolari (test, ...)
 - package
- Helper, costruttori in casi particolari
 - Protetto / Privato
- **Getter e setter** pubblici
 - Accesso **regolamentato** ai data member
 - Vanno usati solo quando necessario

```
public class Pet {  
    // ...  
  
    static { count = 12; }  
  
    public Pet() {  
        this.name = "Waffle";  
        this.weight = 2.18;  
    }  
  
    private void increaseWeight() {  
        weight += weight / 50;  
    }  
  
    void eat() { increaseWeight(); }  
  
    public static int getCount() {  
        return count;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

static initializer

costruttore

La classe Object

- Definita nel package `java.lang`, base della gerarchia di classi in Java
 - Ogni classe deriva (esplicitamente o implicitamente, direttamente o indirettamente) da `Object`
- Confronto di uguaglianza tra istanze via **`equals()`**(`Object`)
 - Deve essere: riflessivo, simmetrico, transitivo, consistente
 - Legato a **`hashCode()`**, pensato per uso delle istanze in hash table
 - Due oggetti uguali, nel senso definito da `equals()`, devono avere lo stesso hash code.
 - Vedi classe `Objects`, metodi `equals()` e `hash()`, e tool per la generazione di codice dell'IDE
- Rappresentazione di una istanza per log / debug via **`toString()`**
 - Per gli array si usa il metodo statico `Arrays.toString(array)`
- Creazione di un clone di una istanza
 - Compito complesso e delicato
 - La classe deve implementare l'interfaccia `Cloneable` e ridefinire il metodo **`clone()`**

L'annotazione Override

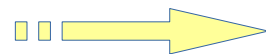
- Annotazione: dà informazioni aggiuntive a un elemento
- @Override
 - Annotazione applicabile solo ai metodi
 - Causa un errore di compilazione se non esiste un “super”-metodo ridefinibile
- Ricordando che la signature è
 - Nome di un metodo, combinato con il numero, tipo e ordine dei suoi parametri
- **Override**: il metodo ridefinito ha la **stessa signature** e tipo di ritorno di un metodo super
 - La classe super può impedire la ridefinizione di un suo metodo, indicandolo come final
 - La visibilità del metodo ridefinito non può essere più ridotta di quella del metodo super
 - Il tipo di ritorno può essere covariante
- **Overload**: metodi con stesso nome ma **signature diversa**

interface

- **Cosa** deve fare una classe, **non come** deve farlo (fino a Java 8)
 - Modificatori della definizione di interfaccia (impliciti): `abstract public`
- Metodi
 - Solo dichiarati, di istanza e (implicitamente) **public**
 - *In Java “moderno” è permessa anche la definizione di*
 - *metodi statici*
 - *un body di default per i metodi di istanza*
 - *metodi privati (sia di istanza, sia statici)*
- Proprietà
 - Sono permesse solo costanti di classe pubbliche
 - Implicitamente **public static final**

Relazioni tra classi/interfacce

- Ereditarietà (**is-a**) keyword **extends** e **implements**
 - extends
 - Classe o interfaccia che ne estende un'altra
 - Eredita proprietà e metodi da super
 - p. es.: Mammal superclass di Cat e Dog
 - implements
 - Classe che implementa un'interfaccia
- Aggregazione (**has-a**)
 - Classe che ha come proprietà un'istanza di un'altra classe
 - p. es.: Tail in Cat e Dog



Ereditarietà in Java

- Single inheritance: una sola superclass, implicita derivazione da **Object** by default
- Una subclass può essere usata al posto della sua superclass (is-a)
 - Per ogni classe X si può scrivere `Object object = new X();`
- Una subclass può aggiungere proprietà e metodi a quelli ereditati dalla superclass
 - (attenzione a non nascondere proprietà della superclass con lo stesso nome!)
- I membri pubblici della superclass fanno parte dell'interfaccia della classe derivata
 - I costruttori fanno eccezione
- Dalla classe si possono accedere tutti i membri della superclass, esclusi quelli private
- La relazione di derivazione è transitiva
 - Se C deriva da B e B deriva da A, allora C deriva da A

interface vs class

```
interface Barker {  
    String bark();  
}
```

```
interface WaggingBarker extends Barker {  
    int DEFAULT_WAG_COUNT = 3;  
  
    String wag();  
}
```

```
public class Fox implements Barker {  
    @Override  
    public String bark() {  
        return "Yap";  
    }  
}
```

extends vs implements

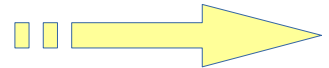
```
public class Dog implements WaggingBarker {  
    @Override  
    public String bark() {  
        return "Woof";  
    }  
  
    @Override  
    public String wag() {  
        StringBuilder sb = new StringBuilder();  
  
        // ...  
  
        return sb.toString();  
    }  
}
```

abstract class

- Una classe abstract non può essere istanziata
- Un metodo abstract non ha body
- Una classe che ha un metodo abstract deve essere abstract
 - Ma non viceversa
- Una subclass di una classe abstract
 - Se implementa tutti i suoi metodi abstract è concreta
 - Altrimenti è a sua volta abstract

this e super

- **this** è una reference all'oggetto corrente
- **super** indica che si intende accedere ad un membro di una *superclass* dal contesto corrente
 - Si risale nella gerarchia fino a trovare il metodo cercato
- Come metodo indicano una relazione tra costruttori
 - Possono essere solo il primo statement di un costruttore
 - **this()**
 - Richiama un altro costruttore nella stessa classe
 - **super()**
 - Richiama un costruttore nella immediata superclass



this e super – esempio

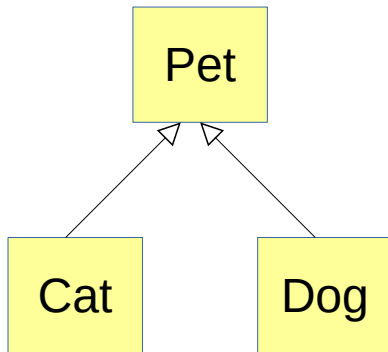
```
public class Pet {  
    private String name;  
  
    public Pet(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
Dog tom = new Dog("Tom");  
  
String name = tom.getName();  
double speed = tom.getSpeed();
```

```
public class Dog extends Pet {  
    private double speed;  
  
    public Dog(String name) {  
        this(name, 0.0);  
    }  
  
    public Dog(String name, double speed) {  
        super(name);  
        this.speed = speed;  
    }  
  
    public double getSpeed() {  
        return speed;  
    }  
}
```

Reference casting

- Upcast: da subclass a superclass
 - Sicuro: subclass is-a superclass
- Downcast: da superclass a subclass
 - Rischioso, va protetto con l'uso di **instanceof**



```
// Cat cat = (Cat) new Dog(); // Cannot cast from Dog to Cat

Pet pet = new Dog("Bob");
Dog dog = (Dog) pet;      // OK here, but unsafe
Cat cat = (Cat) pet;      // trouble at runtime
if(pet instanceof Cat) { // OK
    Cat tom = (Cat) pet;
}
```

Final

- Costante primitiva

```
final int SIZE = 12;
```

- Reference che non può essere riassegnata

```
final StringBuilder sb = new StringBuilder("hello");
```

- Metodo di istanza che non può essere sovrascritto nelle classi derivate

```
public final void f() { // ...
```

- Metodo di classe che non può essere nascosto nelle classi derivate

```
public static final void g() { // ...
```

- Classe che non può essere estesa

```
public final class FinalSample { // ...
```