1. Utilizzare caratteri di dimensioni comprese tra 12 e 14 punti
2. Scegliere un colore di sfondo chiaro come il bianco crema
3. Adottare un interlinea di 1,5 o doppia
4. Limitare la lunghezza delle frasi a un massimo di 70 caratteri
5. Mantenere i paragrafi brevi
6. Preferire l'uso del grassetto anziché corsivo o sottolineatura per enfatizzare concetti
7. Evitare l'allineamento giustificato
8. Optare per l'allineamento a sinistra per mantenere un inizio uniforme delle righe
9. Affiancare i contenuti testuali con immagini pertinenti
10. Evitare l'uso di immagini in movimento o lampeggianti
11. Evitare immagini decorative o utilizzate solo per il layout

Luca Mastrantonio, 2049115.

Feature Implementate:

* Model:
  + Pokemon (+ Owned & Wild), Breed, Move, Type, TypeEffectiveness,

Abbiamo usato MVC …

**Pokemon** è una classe **astratta** con i principali campi d’istanza che solitamente un pokemon ha: *name* (no *nickname* attenzione!), *breed*, *moves*, *hp*, *attack*, *defense*, *special* *attack*, *special* *defense*, *speed*, *level*, *exp*.

Il campo *breed* è una **composizione** (Has-A); il campo *moves* invece è un **HashMap<String, Integer>,** dove String è un codice identificativo della mossa, e Integer è il numero rimanete di utilizzi (PP) della mossa.

Il costruttore ha in input solamente la Breed del pokemon, e assegna a *name* il nome stesso della razza (es: se la Breed è charizard, il nome del pokemon è “Charizard”),

il resto dei campi è inizializzato con valori di default.

Ci sono dei **controlli** nel metodo setLevel(), infatti non è possibile che un pokemon superi il **lvl 100**, mentre altri controlli servono per evitare che un pokemon abbia **più di 4 mosse (l’allenatore decide quale mossa sostituire)**.

**OwnedPokemon** è una classe che **estende** Pokemon, inoltre **implementa** la **serializzazione**.

Il campo nickname è disponibile in questa classe, perché un allenatore può dare un soprannome al suo pokemon.

Il nickname può essere scelto al momento della cattura. e il metodo getName() ritorna *nickname* se presente, altrimenti *name.*

Come funziona la **serializzazione** e **updateSER()**?

Il campo **final private String ID = setID();** rappresenta l’ID univoco di ogni pokemon che ha un proprietario (non ha senso assegnare ID a pokemon selvatici).

È molto frequente l’uso del metodo **updateSER()** perché deve **registrare i cambiamenti** del pokemon (es l’evoluzione, l’aumento delle statistiche, la riduzione della vita) all’interno di un file **.ser**.

Nello specifico, … // TODO

Il metodo **canEvolve()** ritorna un boolean, esso controlla se il livello del pokemon è maggiore o uguale al livello che determina la maturità tale per permettere al pokemon di evolversi ( getBreed().getEvoThreshold() ), questa soglia è caratteristica della razza (Breed) del pokemon, quindi appartiene a quella classe.

Questo **permette** anche di **non evolvere il pokemon** al momento della raggiunta del livello, ma di poter interrompere l’evoluzione e **posticiparla al prossimo livello** (come nei giochi pokemon).

Il metodo **evolve()** agisce solo se canEvolve() ritorna true. Cambia la Breed con quella del suo successore (vedi Breed per maggior dettagli). **Se il nickname non è impostato**, allora viene rinominato anche il nome del pokemon al nome della **razza successiva**.

**Move** è una classe serializzabile.

Ha tra i campi il name, il type (Has-A), description, damage, precision, ppMAX, isSpecial.

Esistono mosse Speciali e mosse Fisiche, ma **non c’è variazione nel comportamento** in questo gioco, quindi, per **non introdurre complessità** (Overengineering), **mantenere manutenibilità**, e garantire **chiarezza** del codice, esiste il campo isSpecial booleano che mantiene questa proprietà della mossa.

Motivazioni dettagliate:

Rendere Move astratta, e creare due sottoclassi SpecialMove e PhisicalMove è una **cattiva idea** perché:

* Vìola il **P**rincipio di **R**esponsabilità **S**ingola (creo una sottoclasse senza differenze comportamentali, introduco una nuova classe che non aggiunge alcuna nuova responsabilità o comportamento)
* **Complessità** Inutile (sottoclassi che non introducono nuove funzionalità o comportamenti aumenta la complessità del codice senza alcun vantaggio tangibile)
* Difficoltà di **Manutenzione** (ogni volta che dovrò apportare modifiche o correggere bug, dovrò gestire più classi, aumentando la possibilità di introdurre errori o di dover fare cambiamenti ripetitivi in più punti. Questo è contrario al principio DRY (Don't Repeat Yourself), che suggerisce di evitare la duplicazione di codice e responsabilità)
* Riduzione della **Flessibilità** (riduce la flessibilità del codice perché aggiunge vincoli inutili e impone una gerarchia che non è necessaria)
* **Confusione** per il collega sviluppatore (potrebbe chiedersi perché esistono sottoclassi senza variazioni e potrebbe tentare di cercare differenze o comportamenti particolari che non esistono. Questo può portare a confusione e a una maggiore difficoltà nel comprendere l'intenzione del codice)

Il metodo setPrecision() ha un **controllo sul valore** della precisione (precisione che va da 0.0 a 1.0): se viene inserito un valore **superiore a 1.0**, esso viene ridotto alla precisione massima.

La classe **Breed** è **serializzabile**.