# OOP Advanced Retake Exam – One Hit Dungeon

Ivo is stuck in an unending cycle. He is bored because he is too lazy to do anything and he is too lazy to do anything because he is bored. Ivo however decided to make a game, but he is lazy, so you will make it for him. The game is called One Hit Dungeon and it’s about an endless dungeon which you can traverse with your character of choice.

### Overview

The software will work with Heroes and Items as the main data models. Heroes own items, and they can upgrade them. By default the items have some battle power which accumulates by a different formula depending on the hero. A hero can fight his current enemy or advance to his next enemy. Heroes can also train their items, to increase their power in order to fight tougher enemies.

### Structure

The structure of the software circles around Heroes and Items.

#### Items

The Items are initialized with a battlePower (**Integer**), a priceForUpgrade (**Double**).

There are generally 3 types of Items.

##### Weapon Item

The Weapon Item is just a normal item. Nothing special here.

##### Offhand Item

The Offhand Item is just a normal item. Nothing special here.

##### Armor Item

The Armor Item is just a normal item. Nothing special here.

#### Heroes

The Heroes are initialized with an name (**String**), a weapon (**WeaponItem**), an offhand (**OffhandItem**) and an armor (**ArmorItem**).  
There are generally 2 types of Heroes.

##### Paladin Hero

The PaladinHero is just a normal hero.

* **Initialized** with a weapon with the following property values:
  + battlePower – 20 (**Integer**)
  + priceForUpgrade – 10 (**Double**)
* **Initialized** with a offhand with the following property values:
  + battlePower – 10 (**Integer**)
  + priceForUpgrade – 10 (**Double**)
* **Initialized** with a armor with the following property values:
  + battlePower – 25 (**Integer**)
  + priceForUpgrade – 20 (**Double**)

##### Mage Hero

The MageHero is just a normal hero.

* **Initialized** with a weapon with the following property values:
  + battlePower – 45 (**Integer**)
  + priceForUpgrade – 15 (**Double**)
* **Initialized** with a offhand with the following property values:
  + battlePower – 25 (**Integer**)
  + priceForUpgrade – 20 (**Double**)
* **Initialized** with a armor with the following property values:
  + battlePower – 10 (**Integer**)
  + priceForUpgrade – 25 (**Double**)

#### Hero Trainer

The HeroTrainer will be given to you in the skeleton.   
You can check more info about it in the Skeleton section.

### Functionality

The main functionality circles around the Heroes and their Items. Heroes are being registered in the system, and after that they can start fighting, advancing deeper in the dungeon and upgrading their Items.

When a Hero is registered, he is **instantiated** with his **items** – Weapon, Offhand and Armor. His **items** define his battle power. Each **hero** has a **different formula** for defining his total battle power.

Heroes can fight the **current dungeon level** using their battle power as a factor for victory, **earning gold** for each victory. They can also advance to the **next level**, to **earn more gold** per victory. If they however **lose** a **fight**, they **return** to the **previous level**.

The **initial level**‘s **battle power** is 20 and the **gold earned** for defeating it is 15. Each time you **advance** the **level** of the dungeon the **current values** are **doubled** (**\* 2**).

Heroes can also train, in order to **increase** the battle power of their items, thus **increasing** their total battle power. Training requires **gold**, which can be **earned** through **fights**. An **upgraded hero** can fight **higher levels** of the dungeon, thus **earning more money**, and **upgrading** himself even more. This is an endless game, or at least until the datatypes explode in exceptions :D.

As we said in the beginning, One Hit Dungeon is a game. As such, it should support **multiple character plays**. When you register the **first** Hero, it **automatically** becomes the **selected hero**. Registering a new hero **should NOT affect** the current selection. The game however should **support functionality** to **select** one of your **registered heroes**.

The system should also support a **functionality** for checking the **status** of the currently selected Hero.

The **currently selected hero** is the **one** around whom the **functionality circles**. Each functionality of the application affects the **currently selected hero** – fighting, advancing, training, checking the status.

Check below, each section, and the functionality it describes.

#### Heroes

Heroes can **earn gold** by **fighting**, and they can **pay gold** in order to **train** their **items**. When trained, their **items** are upgraded, and their battle power and price for upgrade are **increased**.

The **heroes** fight an **imaginary foe** which is represented by a **hardcoded battle power** and **earn gold** by defeating that foe. A foe is **defeated** if the hero has more total battle power than it.

The total battle power is decided by a different formula for each hero type:

Paladin – ((weaponBattlePower + offhandBattlePower + armorBattlePower) \* 4) / 9

Mage – ((weaponBattlePower + armorBattlePower - offhandBattlePower) \* 3) / 4

#### Items

The Items have no business logic around themselves. They are just **data models**.

#### Commands

There are several commands which are given from the user input, in order to control the program.   
Here you can see how they are formed.

The **parameters** will be given in the **EXACT ORDER**, as the one **specified below**.   
You can see the exact input format in the **Input section**.

**Each** **command** will **generate an output** **result**, which you must **print**.  
You can see the exact output format in the **Output section**.

##### Hero Command

**Parameters** – **type** (string), **name** (string).

The type will always be one of the following values: “Paladin”, “Mage”.

Creates a Hero of the **given type** with the **given name**. If this is the **first registered hero** in the game, it automatically becomes the **currently selected hero**.

##### Select Command

**Parameters** – **name** (string).

**Selects** the Hero with the **given name** as the **currently selected hero**.

Note: **Selecting** a **different hero**, does **NOT** **reset** the **current level** of the dungeon.

##### Status Command

**Parameters** – no parameters here.

Brings report of the **currently selected hero**, providing **detailed** **information** about the corresponding object.

##### Fight Command

**Parameters** – no parameters here.

The **currently selected hero** fights the **current level** of the dungeon. By **default** the **first level** of the **dungeon** is **easily defeated** by the **hero’s primary state** (without any training), so it can be used for earning enough gold to train.

If the Hero **wins** the **battle** – his **battle power** is **higher** than the **current dungeon level**’s **battle power**, he **earns** the **current dungeon level**’s **gold** on **victory**.

If the Hero **loses** the **battle** – his **battle power** is **lower** than the **current dungeon level**’s **battle power**, he must be **returned** to the **previous level**.

##### Advance Command

**Parameters** – no parameters here.

**Advances** the **currently selected hero** to the **next level** of the **dungeon**. **Doubles** the **current dungeon level**’s **battle power** and **gold earned** on **victory**.

##### Train Command

**Parameters** – no parameters here.

**Trains** the **current selected hero**, trough the HeroTrainer. The HeroTrainer defines the full logic of the training. You just need to **check** if the Hero has **enough gold** to **train himself**. This is **done** by **checking** his **total gold** and his **total price for upgrade**.

##### Quit Command

**Exits** the program. Prints **detailed information** about the **whole** system.

### Skeleton

In this section you will be given information about the Skeleton, or the code that has been given to you.

You are allowed to change the **internal** and **private logic** of the **classes** that have been given to you.   
In other words, you can change the **body code** and the **definitions** of the **private members** in whatever   
way you like.

However. . .

You are **NOT ALLOWED** to **CHANGE** the **Interfaces** that have been provided by the **skeleton** in **ANY way**.   
You are **NOT ALLOWED** to **ADD** more **PUBLIC LOGIC**, than the **one**, **provided** by the **Interfaces**, **ASIDE FROM** the toString() method and compareTo() method.

#### Interfaces & Others

You will be given the **interfaces** for the Hero and Item entities. You should use them when you are implementing your entities.

You will **also be given** an **interface** for the HeroTrainer class, but you will also be given the **class**.

#### HeroTrainer

The HeroTrainer class **instantiates** a map with values which act as percentages with which the items are upgraded.

The HeroTrainer receives a Hero and **extracts** its **item fields** using **Reflection**. It then **extracts** the **fields** of each item and **increase** their **value** by a **specific percentage**, based on the Item type and the Hero type.

Each time a Paladin is trained:

* Its Weapon’s:
* battlePower is increased by **60%**.
* priceForUpgrade is increased by **50%**.
* Its Offhand’s:
* battlePower is increased by **50%**.
* priceForUpgrade is increased by **100%**.
* Its Armor’s:
* battlePower is increased by **20%**.
* priceForUpgrade is increased by **75%**.

Each time a Mage is trained:

* Its Weapon’s:
* battlePower is increased by **20%**.
* priceForUpgrade is increased by **67%**.
* Its Offhand’s:
* battlePower is increased by **20%**.
* priceForUpgrade is increased by **100%**.
* Its Armor’s:
* battlePower is increased by **100%**.
* priceForUpgrade is increased by **100%**.

### Input

The input consists of several commands which will be given in the format, specified below: :

* Hero {type} {name}
* Select {name}
* Status
* Fight
* Advance
* Train
* Over

### Output

Each of the commands generates **output**. Here are the **output formats** of each command:

* Hero Command – registers a Hero of the given type with given name. Prints the following result:

**Successfully added {heroType} – {name}.**

* Select Command – selects the Hero, with the given name, as the currently selected hero.  
  Prints the following result:

**Successfully selected {heroType} - {heroName}.**

* Fight Command – makes the currently selected hero fight the current dungeon level.
  + Prints the following result if the battle is won:

**Fight won. You’ve gained {goldEarned} gold.**

* + Prints the following result if the battle is won:

**Fight lost. You’ve returned to the previous level.**

* Advance Command – advances the current dungeon level.

**Successfully advanced to dungeon level {newDungeonLevel}.**

* Train Command – trains the currently selected hero.
  + Prints the following result if the Hero has enough gold and successfully trains:

**Successfully trained hero. Current total battle power: {upgradedTotalBattlePower}.**

* + Prints the following result if the Hero does NOT have enough gold and does NOT train:

**Insufficient gold for training. Needed {upgradeCost}. Got {currentGold}.**

* Status command – provides **detailed** **information** about the **currently selected hero**, in the following format:

|  |
| --- |
| Paladin |
| {heroName} – Lvl. {timesTrained + 1} Paladin  \* Mace – {weaponBattlePower} (BP)  \* Shield – {offhandBattlePower} (BP)  \* Cuirass – {armorBattlePower} (BP)  ####################  Gold: {gold}  Upgrade cost: {totalItemsPriceForUpgrade} |
| Mage |
| {heroName} – Lvl. {timesTrained + 1} Mage  \* Staff – {weaponBattlePower} (BP)  \* Orb – {offhandBattlePower} (BP)  \* Cape – {armorBattlePower} (BP)  ####################  Gold: {gold}  Upgrade cost: {totalItemsPriceForUpgrade} |

* + Note: The timesTrained is the times the hero has been trained. By initially it is 0, so +1 it will appear as 1.
  + Note: There are exactly 20 “#” in the string representation of the heroes.
* Quit command – Terminates the program; **prints** detailed statistics about the whole system. The format, in which the statistics should be printed is:

{hero1type} {hero1name} – {hero1totalBattlePower} (BP)  
{hero2type} {hero2name} – {hero2totalBattlePower} (BP)  
{hero3type} {hero3name} – {hero3totalBattlePower} (BP)  
. . .  
####################  
Dungeon level reached: {dungeonLevelReached}

* + You should **traverse all heroes** and print **each** Hero in the format specified above.
  + You should print the dungeonLevelReached. The first level is **1**. Each time you **advance** the **level** is **incremented**.

Note: All output **floating-point numbers** must be formatted to the **2nd digit** after the **decimal point**.

### Constrains

* All **input lines** will be **absolutely valid**.
* There will be **NO** non-existent **names** in the input.
* There will be **NO** duplicate **names** in the input.
* You **CAN** use the **names** as a **unique identifier**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Hero Paladin Pesho  Status  Fight  Fight  Fight  Fight  Train  Fight  Fight  Fight  Fight  Fight  Fight  Train  Advance  Fight  Quit | Successfully added Paladin - Pesho.  Pesho - Lvl. 1 Paladin  \* Mace - 20 (BP)  \* Shield - 10 (BP)  \* Cuirass - 25 (BP)  ####################  Gold: 0,00  Upgrade cost: 40,00  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Successfully trained hero. Current total battle power: 34.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Successfully trained hero. Current total battle power: 48.  Successfully advanced to dungeon level 2.  Fight won. You've gained 30,00 gold.  Paladin Pesho - 48 (BP)  ####################  Dungeon level reached: 2 |
| Hero Mage Pesho  Hero Paladin Gosho  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Fight  Train  Train  Advance  Fight  Fight  Status  Select Gosho  Train  Fight  Quit | Successfully added Mage - Pesho.  Successfully added Paladin - Gosho.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Fight won. You've gained 15,00 gold.  Successfully trained hero. Current total battle power: 33.  Successfully trained hero. Current total battle power: 51.  Successfully advanced to dungeon level 2.  Fight won. You've gained 30,00 gold.  Fight won. You've gained 30,00 gold.  Pesho - Lvl. 3 Mage  \* Staff - 64 (BP)  \* Orb - 36 (BP)  \* Cape - 40 (BP)  ####################  Gold: 64,95  Upgrade cost: 221,83  Successfully selected Paladin - Gosho.  Insufficient gold for training. Needed 40,00. Got 0,00.  Fight lost. You've returned to the previous level.  Mage Pesho - 51 (BP)  Paladin Gosho - 24 (BP)  ####################  Dungeon level reached: 1 |

### Tasks

#### Task 1: High Quality Structure

##### Refactor the given Skeleton code and use it.

You have been given some **interfaces**, which you must **implement** – each and every one of them!

Ivo wrote some code before you, but he writes really bad and broken code… He somehow managed to write the HeroTrainer class. His work, however, is not that trustworthy, so you might have to give it an eye or two, for potential **functionality bugs** and things that **do NOT follow** the **good practices** of **Object-Oriented Programming**.

Refactor anything, which will **improve** the **code quality**, in your opinion. Be careful **NOT** to **break the code** or one of the **rules** specified in the **Skeleton** **section**.

##### High Quality Code.

Achieve good separation of concerns using abstractions and interfaces to decouple classes, while reusing code through inheritance and polymorphism. Your classes should have strong cohesion - have single responsibility and loose coupling - know about as few other classes as possible.

##### Correct Implementation

Implement all classes and commands, exactly as they are said in the Problem description. If you have a class “Pesho Tractor”, you implement “PeshoTractor.class”.

Note: For this task, submit only the “onehitdungeon” folder.

#### Task 2: Correct business logic.

The given code provides some functionality, but it does not cover the entire task. Implement the rest of the business logic, using the given code, and implement everything following the requirements specification. Check your solutions in the Judge system.

Note: For this task, submit the whole “src” folder.

Note: The High-Quality Structure tests are **not connected** to the Business Logic tests, which gives you the ability to break the structure completely in order for the Business Logic to pass. You are **NOT** **allowed** to do that**. Each submission** on the **Business Logic** will be **checked** with the **High-Quality Structure** tests.

#### Task 3: Unit Testing.

Test the HeroTrainer class’s **PUBLIC** methods for potential bugs. There is **only 1 public method**, but you can test a **lot** of **private logic** with it. Extensive testing might require the use of **Mocking**.

When testing, use **ONLY THE CLASSES** and **INTERFACES, PROVIDED** by the **SKELETON** classes + the **Item** classes.

For this task submit the **folder** you have put your **tests** into. The **root test package** folder.

**NOTE**: You are **NOT ALLOWED** to submit **non-test classes** for this task.