OOP Basics Exam – Dungeons and Code Wizards

The time has come for your OOP Basics exam. It will be a perilous journey, but if you manage to keep a cool head, you'll be able to get through it with minimal damage.

Overview

In this exam, you need to build a Dungeons and Dragons-esque project, which has support for **characters**, **items** and **inventories** for storing each character's items. The project will consist of the **entity classes** and a **controller class**, which manages the **interaction** between the characters and items.

Task 1: Structure (150 points)

Item

This is a **base class** for any **items** and it **should not be able to be instantiated**.

Data

`Weight` – an **integer number**

Behavior

Each **item** has the following **behavior**:

void AffectCharacter(Character character)

For an item to affect a character, the character **needs to be alive**.

If not, **throw** an **`InvalidOperationException` with the message** "**Must be alive to perform this action!**".

Throw this exception everywhere a character needs to be alive to perform the action.

Constructor

An **item** should take the following values upon initialization:

```
int weight
```

HealthPotion

The **health potion** always has a **weight** of **5**.

Behavior

Each **`HealthPotion`** has the following **behavior**:

void AffectCharacter(Character character)

For an item to affect a character, the character **needs to be alive**.

The character's **health** gets **increased** by **20 points**.

Constructor

An **item** should be able to be instantiated **without any parameters**.

PoisonPotion

The **poison potion** always has a **weight** of **5**.

Behavior

Each **`PoisonPotion`** has the following **behavior**:

void AffectCharacter(Character character)

For an item to affect a character, the character **needs to be alive**.

The character's **health** gets **decreased** by **20 points**. If the character's health **drops to zero**, the character **dies** (**`IsAlive`** ➔ **`false`**).

Constructor

A **`PoisonPotion`** should be able to be instantiated **without any parameters**.

ArmorRepairKit

The armor repair kit always has a **weight** of **10**.

Behavior

Each **`ArmorRepairKit`** has the following **behavior**:

void AffectCharacter(Character character)

For an item to affect a character, the character **needs to be alive**.

The character's **armor** restored up to the **base armor** value.

Example: Armor: 10, Base Armor: 100 ➔ Armor: 100

Constructor

An **`ArmorRepairKit`** should be able to be instantiated **without any parameters**.

Bag

This is a **base class** for any **bags** and it **should not be able to be instantiated**.

Data

- **`Capacity`** – an **integer number**. Default value: 100
- **`Load`** – Calculated property. The **sum of the weights** of the **items** in the bag.
- **`Items`** – **Read-only collection** of type **`Item`**

Behavior

Each **bag** has the following **behavior**:

void AddItem(Item item)

If the current load + the weight of the item attempted to be added is **greater than** the bag's **capacity**, throw an **`InvalidOperationException`** with the message "**`Bag is full!`**"

If the check passes, the **item** is added to the **bag**.

Item GetItem(string name)

If no items exist in the bag, throw an **`InvalidOperationException`** with the message "**`Bag is empty!`**"

If an item with that **name doesn't exist** in the bag, throw an **`ArgumentException`** with the message "**`No item with name {name} in bag!`**"

If both checks pass, the **item** is removed from the **bag** and **returned** to the **caller**.

Constructor

A **Bag** should take the following values upon initialization:

```
int capacity
```

Backpack

This is a **type of bag** with 100 capacity.

Satchel

This is a **type of bag** with 20 capacity.

Character

This is a **base class** for any **characters** and it **should not be able to be instantiated**.

Data

- `Name` – a **string (cannot be null or whitespace)**.
    - **Throw an ArgumentException with the message "Name cannot be null or whitespace!"**
- `BaseHealth` – a **floating-point number**
- `Health` – a **floating-point number** (current health).
    - Health maxes out at **BaseHealth** and mins out at 0.
- `BaseArmor` – a **floating-point number**
- `Armor` – a **floating-point number** (current armor)
    - Armor maxes out at **BaseArmor** and mins out at 0.
- `AbilityPoints` – a **floating-point number**
- `Bag` – a parameter of type **Bag**
- `Faction` – a constant value with **2 possible values**: **CSharp** and **Java**
- `IsAlive` – boolean value (default value: `True`)
- `RestHealMultiplier` – a **floating-point number** (default: **0.2**), **could be overriden**

Behavior

Each **character** has the following **behavior**:

void TakeDamage(double hitPoints)

For a character to take damage, they need to **be alive**.

The character takes damage equal to the **hit points**. Taking damage works like so:

The character's **armor** is **reduced** by the **hit point amount**, then if there are **still hit points left**, they take that amount of **health damage**.

If the character's **health** drops to **zero**, the character **dies** (`IsAlive` become **false**)

Example: Health: **100**, Armor: **30**, Hit Points: **40** ➔ Health: **90**, Armor: **0**

void Rest()

For a character to rest, they need to **be alive**.

The character's **health** increases by their `BaseHealth`, multiplied by their `RestHealMultiplier`

Example: `Health`: **50**, `BaseHealth`: **100**, `RestHealMultiplier`: **0.2** ➔ `Health`: 50 + (100 * 0.2) ➔ 70

void UseItem(Item item)

For a character to use an item, they need to be **alive**.

The item affects the character with the item effect.

void UseItemOn(Item item, Character character)

For a character to use an item on another character, **both of them** need to be **alive**.

The item affects the targeted character with the item effect.

void GiveCharacterItem(Item item, Character character)

For a character to give another character an item, **both of them** need to be **alive**.

The targeted character **receives the item**.

void ReceiveItem(Item item)

For a character to receive an item, they need to be **alive**.

The character puts the **item** into their **bag**.

Constructor

A **character** should take the following values upon initialization:

```
string name, double health, double armor, double abilityPoints, Bag bag, Faction faction
```

IAttackable

A **contract** for any class that **implements it**, that includes an "**`Attack(Character character)`**" method

IHealable

A **contract** for any class that **implements it**, that includes a "**`Heal(Character character)`**" method

Warrior

The **Warrior** is a special class, who can **attack** other characters.

Data

The Warrior class always has **100 Base Health**, **50 Base Armor**, **40 Ability Points**, and a **Satchel** as a bag.

Constructor

The **Warrior** only needs a **name** and a **faction** for initialization:

```
string name, Faction faction
```

Behavior

The warrior only has **one special behavior** (every other behavior is **inherited**):

void Attack(Character character)

For a character to attack another character, **both of them need to be alive**.

If the character they are trying to attack is the same character, throw an **`InvalidOperationException`** with the message "**`Cannot attack self!`**"

If the character the character is attacking is from the **same faction**, throw an **`ArgumentException`** with the message "**`Friendly fire! Both characters are from {faction} faction!`**"

If all of those checks pass, the **receiving character takes damage** with **hit points** equal to the **attacking character's ability points**.

Cleric

The **Cleric** is a special class, who can **heal** other characters.

Data

The Cleric class always has **50 Base Health**, **25 Base Armor**, **40 Ability Points**, and a **Backpack** as a bag.

The cleric's **`RestHealMultiplier`** is **0.5**.

Constructor

The **Cleric** only needs a **name** and a **faction** for initialization:

```
string name, Faction faction
```

Behavior

The cleric only has **one special behavior** (every other behavior is **inherited**):

void Heal(Character character)

For a character to heal another character, **both of them need to be alive**.

If the character the character is healing is from a **different faction**, throw an **`InvalidOperationException`** with the message "**`Cannot heal enemy character!`**"

If both of those checks pass, the **receiving character's health increases by the healer's ability points**.

Task 2: Business Logic (200 points)

The Controller Class

The business logic of the program should be concentrated around several **commands**. Implement a class called **`DungeonMaster`**, which will hold the **main functionality**.

The Dungeon Master keeps track of the **character party** and the **item pool** (the items in the game, which can be picked up). It also keeps track of the **last survivor's consecutive rounds (explained below)**.

Note: The `DungeonMaster` class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!

The main functionality is represented by these **public methods**:

`DungeonMaster.cs`
```
public string JoinParty(string[] args)
{
    throw new NotImplementedException();
}

public string AddItemToPool(string[] args)
{
    throw new NotImplementedException();
}

public string PickUpItem(string[] args)
{
    throw new NotImplementedException();
```

```
}

public string UseItem(string[] args)
{
    throw new NotImplementedException();
}

public string UseItemOn(string[] args)
{
    throw new NotImplementedException();
}

public string GiveCharacterItem(string[] args)
{
    throw new NotImplementedException();
}

public string GetStats()
{
    throw new NotImplementedException();
}

public string Attack(string[] args)
{
    throw new NotImplementedException();
}

public string Heal(string[] args)
{
    throw new NotImplementedException();
}

public string EndTurn(string[] args)
{
    throw new NotImplementedException();
}

public bool IsGameOver()
{
     throw new NotImplementedException();
}
```

NOTE: DungeonMaster class methods are called from the outside so these methods **must NOT** receive the command parameter (the **first argument** from the input line) as part of the arguments!

ALSO NOTE: The `DungeonMaster` class should not handle any exceptions. That should be the responsibility of the class, which reads the commands and passes them to the `DungeonMaster`.

Commands

There are several commands that control the business logic of the application and you are supposed to build. They are stated below.

JoinParty Command

Parameters

- **`faction`** – a **string**
- **`characterType`** – **string**
- **`name`** – **string**

Functionality

Creates a character and **adds them** to the **party**.

If the **faction** is invalid, throw an **`ArgumentException`** with the message "**`Invalid faction "{faction}"!`**"

If the **character type** is invalid, throw an **`ArgumentException`** with the message "**`Invalid character type "{characterType}"!`**"

Returns the **string** "**`{name} joined the party!`**"

AddItemToPool Command

Parameters

- **`itemName`** –**string**

Functionality

Creates an item and **adds it** to the **item pool**.

If the **item type** is **invalid**, throw an **`ArgumentException`** with the message "**`"Invalid item "{name}"!`**"

Returns the **string** "**`{itemName} added to pool.`**"

PickUpItem Command

Parameters

- **`characterName`** – **string**

Functionality

Makes the character with the specified name **pick up the last item in the item pool**.

If the character doesn't exist in the **party**, throw an **`ArgumentException`** with the message "**`Character {name} not found!`**"

If there's **no items left** in the pool, throw an **`InvalidOperationException`** with the message "**`No items left in pool!`**"

Returns the **string** "**`{characterName} picked up {itemName}!`**"

UseItem Command

Parameters

- **`characterName`** – a **string**
- **`itemName`** – **string**

Functionality

Makes the character with that name use an **item with that name from their bag**.

If the character doesn't exist in the **party**, throw an **`ArgumentException`** with the message "**`Character {name} not found!`**"

The rest of the exceptions should be processed by the called functionality (empty bag, etc.)

Returns the **string** "**`{character.Name} used {itemName}.`**"

UseItemOn Command

Parameters

- **`giverName`** – a **string**
- **`receiverName`** – **string**
- **`itemName`** – **string**

Functionality

Makes the giver get an **item with that name from their bag** and **uses it on the receiving character**.

Process any edge cases (giver not found, receiver not found, item not found, etc.) in the same way as in the above commands.

Returns the **string** "`{giverName} used {itemName} on {receiverName}.`"

GiveCharacterItem Command

Parameters

- `giverName` – a **string**
- `receiverName` – **string**
- `itemName` – **string**

Functionality

Makes the giver get an **item with that name from their bag** and **gives it to the receiving character**.

Process any edge cases (giver not found, receiver not found, item not found, etc.) in the same way as in the above commands.

Returns the **string** "`{giverName} gave {receiverName} {itemName}.`"

GetStats Command

Parameters

No parameters.

Functionality

Returns info about **all characters**, sorted by **whether they are alive** (**descending**), **then by** their **health** (**descending**)

The format of a single character is:

```
{name} - HP: {health}/{baseHealth}, AP: {armor}/{baseArmor}, Status: {Alive/Dead}
```

Returns the formatted character info for each character, **separated by new lines**.

Attack Command

Parameters

- `attackerName` – a **string**
- `receiverName` – **string**

Functionality

Makes the **attacker attack** the **receiver**.

If any of the characters don't exist in the party, throw exceptions with messages just like the above commands.

If the **attacker cannot attack**, throw an **`ArgumentException`** with the message "**`{attacker.Name} cannot attack!`**"

The command output is in the following format:

```
{attackerName} attacks {receiverName} for {attacker.AbilityPoints} hit points!
{receiverName} has {receiverHealth}/{receiverBaseHealth} HP and
{receiverArmor}/{receiverBaseArmor} AP left!
```

If the attacker ends up **killing** the receiver, add a **new line**, plus "**`{receiver.Name} is dead!`**" to the output.

Returns the **formatted string**

Heal Command

Parameters

- `healerName` – a **string**
- `healingReceiverName` – **string**

Functionality

Makes the **healer heal** the **healing receiver**.

If any of the characters don't exist in the party, throw exceptions with messages just like the above commands.

If the **healer cannot heal**, throw an **`ArgumentException`** with the message "**`{healerName} cannot heal!`**"

The command **output** is in the following format:

`{healer.Name} heals {receiver.Name} for {healer.AbilityPoints}! {receiver.Name} has {receiver.Health} health now!`

Returns the **formatted string**

EndTurn Command

Parameters

No parameters.

Functionality

Ends the turn. Several things happen when a turn ends.

First, each **alive** character **rests**. Then the line "**`{character.Name} rests ({healthBeforeRest} => {currentHealth})`**" is added to the output.

If there are **one or zero alive** characters left, the **last survivor rounds** are **incremented by one**.

Returns all the "**x rests…**" commands, separated by new lines.

IsGameOver Command

Parameters

No parameters.

Functionality

If the **last survivor** survives **alone more than one round**, the game is **over**.

The command returns whether the game is over or not (**`true`**/**`false`**)

Task 3: Input / Output (100 points)

Input

- You will receive commands **until the game is over** or **until the command** you read **is null or empty**.

Below, you can see the **format** in which **each command** will be given in the input:

- **`JoinParty {Java/CSharp} {class} {name}`**
- **`AddItemToPool {itemName}`**
- **`PickUpItem {characterName}`**

- **`UseItem {characterName} {itemName}`**
- **`UseItemOn {giverName} {receiverName} {itemName}`**
- **`GiveCharacterItem {giverName} {receiverName} {itemName}`**
- **`GetStats`**
- **`Attack {attackerName} {attackTargetName}`**
- **`Heal {healerName} {healingTargetName}`**
- **`EndTurn`**
- **`IsGameOver`**

Output

Print the output from each command when issued. When the game is over, print "**`Final stats:`**" and the output from the **`GetStats`** command.

If an exception is thrown during any of the commands' execution, print:

- "**`Parameter Error:`** " plus the message of the exception if it's an **`ArgumentException`**
- "**`Invalid Operation:`** " plus the message of the exception if it's an **`InvalidOperationException`**

Constraints

- The commands will always be in the provided format.

Examples

Input
```
JoinParty CSharp Warrior Gosho
JoinParty Java Warrior Pesho
AddItemToPool HealthPotion
AddItemToPool ArmorRepairKit
AddItemToPool PoisonPotion
PickUpItem Gosho
EndTurn
JoinParty Java Cleric Ivan
Attack Gosho Pesho
Attack Gosho Pesho
EndTurn
Attack Gosho Pesho
Heal Ivan Pesho
EndTurn
Attack Gosho Ivan
Attack Gosho Ivan
Attack Gosho Pesho
Attack Gosho Pesho
Attack Gosho Pesho
EndTurn
EndTurn
```
Output
```
Gosho joined the party!
Pesho joined the party!
HealthPotion added to pool.
ArmorRepairKit added to pool.
PoisonPotion added to pool.
```

```
Gosho picked up PoisonPotion!
Gosho rests (100 => 100)
Pesho rests (100 => 100)
Ivan joined the party!
Gosho attacks Pesho for 40 hit points! Pesho has 100/100 HP and 10/50 AP left!
Gosho attacks Pesho for 40 hit points! Pesho has 70/100 HP and 0/50 AP left!
Gosho rests (100 => 100)
Pesho rests (70 => 90)
Ivan rests (50 => 50)
Gosho attacks Pesho for 40 hit points! Pesho has 50/100 HP and 0/50 AP left!
Ivan heals Pesho for 40! Pesho has 90 health now!
Gosho rests (100 => 100)
Pesho rests (90 => 100)
Ivan rests (50 => 50)
Gosho attacks Ivan for 40 hit points! Ivan has 35/50 HP and 0/25 AP left!
Gosho attacks Ivan for 40 hit points! Ivan has 0/50 HP and 0/25 AP left!
Ivan is dead!
Gosho attacks Pesho for 40 hit points! Pesho has 60/100 HP and 0/50 AP left!
Gosho attacks Pesho for 40 hit points! Pesho has 20/100 HP and 0/50 AP left!
Gosho attacks Pesho for 40 hit points! Pesho has 0/100 HP and 0/50 AP left!
Pesho is dead!
Gosho rests (100 => 100)
Gosho rests (100 => 100)
Final stats:
Gosho - HP: 100/100, AP: 50/50, Status: Alive
Pesho - HP: 0/100, AP: 0/50, Status: Dead
Ivan - HP: 0/50, AP: 0/25, Status: Dead
```

Input

```
JoinParty CSharp Warrior Gosho
JoinParty CSharp Warrior Pesho
AddItemToPool HealthPotion
AddItemToPool PoisonPotion
PickUpItem Pesho
PickUpItem Gosho
PickUpItem Pesho
UseItem Pesho HealthPotion
UseItem Pesho PoisonPotion
UseItemOn Gosho Pesho HealthPotion
AddItemToPool PoisonPotion
PickUpItem Gosho
GiveCharacterItem Gosho Pesho PoisonPotion
JoinParty Java Warrior Ivan
Attack Ivan Gosho
Attack Ivan Gosho
Attack Ivan Gosho
Attack Gosho Ivan
Attack Ivan Gosho
EndTurn
Attack Ivan Pesho
Attack Ivan Pesho
Attack Ivan Pesho
Attack Ivan Pesho
EndTurn
EndTurn
```

Output

```
Gosho joined the party!
Pesho joined the party!
```

```
HealthPotion added to pool.
PoisonPotion added to pool.
Pesho picked up PoisonPotion!
Gosho picked up HealthPotion!
Invalid Operation: No items left in pool!
Invalid Operation: No item with name HealthPotion in bag!
Pesho used PoisonPotion.
Gosho used HealthPotion on Pesho.
PoisonPotion added to pool.
Gosho picked up PoisonPotion!
Gosho gave Pesho PoisonPotion.
Ivan joined the party!
Ivan attacks Gosho for 40 hit points! Gosho has 100/100 HP and 10/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 70/100 HP and 0/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 30/100 HP and 0/50 AP left!
Gosho attacks Ivan for 40 hit points! Ivan has 100/100 HP and 10/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 0/100 HP and 0/50 AP left!
Gosho is dead!
Pesho rests (100 => 100)
Ivan rests (100 => 100)
Ivan attacks Pesho for 40 hit points! Pesho has 100/100 HP and 10/50 AP left!
Ivan attacks Pesho for 40 hit points! Pesho has 70/100 HP and 0/50 AP left!
Ivan attacks Pesho for 40 hit points! Pesho has 30/100 HP and 0/50 AP left!
Ivan attacks Pesho for 40 hit points! Pesho has 0/100 HP and 0/50 AP left!
Pesho is dead!
Ivan rests (100 => 100)
Ivan rests (100 => 100)
Final stats:
Ivan - HP: 100/100, AP: 10/50, Status: Alive
Gosho - HP: 0/100, AP: 0/50, Status: Dead
Pesho - HP: 0/100, AP: 0/50, Status: Dead
```

Input

```
JoinParty Java Warrior Gosho
JoinParty CSharp Warrior Ivan
Attack Gosho Gosho
PickUpItem Gosho
AddItemToPool InvalidItem
AddItemToPool HealthPotion
UseItem Gosho InvalidItem
UseItem Gosho HealthPotion
PickUpItem InvalidCharacter
Attack Ivan Ivan
Attack Pesho Ivan
Attack Ivan Pesho
Attack A B
Attack Ivan Gosho
Attack Ivan Gosho
Attack Ivan Gosho
Attack Ivan Gosho
EndTurn
EndTurn
```

Output

```
Gosho joined the party!
Ivan joined the party!
Invalid Operation: Cannot attack self!
Invalid Operation: No items left in pool!
Parameter Error: Invalid item "InvalidItem"!
```

```
HealthPotion added to pool.
Invalid Operation: Bag is empty!
Invalid Operation: Bag is empty!
Parameter Error: Character InvalidCharacter not found!
Invalid Operation: Cannot attack self!
Parameter Error: Character Pesho not found!
Parameter Error: Character Pesho not found!
Parameter Error: Character A not found!
Ivan attacks Gosho for 40 hit points! Gosho has 100/100 HP and 10/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 70/100 HP and 0/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 30/100 HP and 0/50 AP left!
Ivan attacks Gosho for 40 hit points! Gosho has 0/100 HP and 0/50 AP left!
Gosho is dead!
Ivan rests (100 => 100)
Ivan rests (100 => 100)
Final stats:
Ivan - HP: 100/100, AP: 50/50, Status: Alive
Gosho - HP: 0/100, AP: 0/50, Status: Dead
```

Task 4: Bonus (50 points)

Factories

You know that the keyword **new** is a bottleneck and we are trying to use it as little as possible. We even try to separate it in classes. These classes are called **Factories** and the naming convention for them is **{TypeOfObject}Factory**. You need to have **two different factories**, **one for Characters and one for Items**. This is a design pattern and you can read more about it. Factory Pattern. The factories must contain a method ("**`CreateCharacter`**/**`CreateItem`**"), which instantiates objects of that type.

If you try to create a character with an invalid type, throw an **`ArgumentException`** with a message "**`Invalid character type "{type}"!`**".

If you try to create a character with an invalid type, throw an **`ArgumentException`** with a message "**`Invalid item type "{type}"!`**".