Java Grinds Task № 1

Gary Munnelly, Trinity College Dublin February 22, 2016

Introduction

Perhaps the most popular application of software development is that of the creation of games. The proliferation of video games makes it difficult to find anyone who isn't familiar with basic operations such as stat point allocation or inventory management. Taking the latter as an example, we all take for granted that an inventory will be used to store a list of items found in the game irrespective of what that item does or how it is defined in code. Yet every in-game object has some form of unique behaviour, whether it is the minor difference between two different types of rifle (hunting rifle vs. lever-action rifle) or the immense difference between a consumable medi-pack and a wearable piece of armour (e.g. stimpack vs. ranger battle armour).

Given our current knowledge of programming, how is it possible to have such a gaming feature without needing to define an array for every possible type of item that our hero may encounter? The answer, of course, lies in interfaces. Our task for this grind will be to develop a simple inventory system that will allow our player to store a list of various items which will come to differ in subtle (and not so subtle) ways.

This grind also offers the chance to review more basic concepts such as array manipulation as we will insert, remove, select and display our inventory contents for the benefit of our user.

Future grind tasks should build on this one, so it is imperative that this code work correctly before advancing to the next problem.

Requirements

The goal of this grind is to create a basic inventory system which will allow our player to store various items that they find in the game world. Solving this problem involves the definition of one interface and one class – an item interface and an inventory class.

The item interface (which we will name **IItem**) will define the basic behaviours which should be exhibited by any item stored in the inventory. There are only two behaviours required:

`String getName()` Returns the name of the object which we may display to the user, e.g. "Sword of Might", "Mace of Spraying"

`String getDescription()` Returns a longer description of the object which may be displayed to inform the user what they are holding, e.g. "Someone has scrawled "*wuss*" all along the outer rim of this shield."

We could also choose to include behaviours such as `getWeight` if we are planning to limit the maximum amount of weight our hero can carry.

For the purposes of demonstration we shall define three classes which implement the **IItem** interface:

- **`Sword`** An equippable attack item
- **`Shield`** An equippable defence item
- **`Potion`** A consumable item

Of course, we could call these objects whatever we want. What is important is that they are all derived from the **IItem** interface. It is this key step that will allow us to store all these different item objects in a single array. For now they'll all behave very much the same, but later we'll start to introduce differences in how our player can interact with them beyond the basic `getName`, `getDescription` functionality.

The vast majority of code will belong to the **Inventory** class. This class is responsible for managing all the items that our player is carrying. We will define it such that the player is limited by the number of items they can carry, rather than the total weight, e.g. the user will only be allowed to carry 10 items. We will require two constructors:

`Inventory()` Creates a new **Inventory** object with the default maximum capacity (we'll set this at 10)

`Inventory(int maxCapacity)` Creates a new instance of the **Inventory** class which allows the user to hold `maxCapacity` number of items

Two numbers which may be of interest to us are the number of items we are currently carrying and the maximum number of items we are permitted to carry. We will define two functions to retrieve these values:

`int getMaxCapacity()` Gets the maximum number of items that we can hold. This may simply be returned as the length of the array that we are using to store items

`int getNumItems()` Returns the number of items we currently have stored in our inventory. This is a count which you might want to update each time you add or remove and item to/from the array

Finally, we wish to define the basic operations of an inventory. Those simple tasks that we expect of any system which is managing our loot:

`IItem getItem(int index)` Gets the item stored at `index` in the array. This function will return `null` if no item is found or if `index` is invalid. Note that this function should *remove* the item from the inventory before returning it as we take it that an item retrieved from the inventory is now being held

`boolean insertItem(IItem item)` Stores `item` in our inventory array. This function will return `true` if the operation succeeds and `false` otherwise, e.g. if the array is full or the item passed is `null`

`boolean dropItem(int index)` Removes the item stored at `index` from the inventory. This function will return `true` if the item is successfully dropped and `false` otherwise e.g. if the value of `index` is invalid

`void listContents()` Prints the entire contents of the inventory to the console in the form:

```
0: <name> --> <description>
1: <name> --> <description>
2: <name> --> <description>
   ⋮
8: EMPTY
9: EMPTY
```

where the string "EMPTY" denotes unfilled slots

Extra challenge

For the sake of making the task of manipulating the inventory array more challenging, see if you can implement **Inventory** in such a way that items are stored alphabetically at all times, i.e. the insertion or deletion of an object from the inventory will trigger some form of sorting behaviour which ensures that items always appear in a particular sequence.

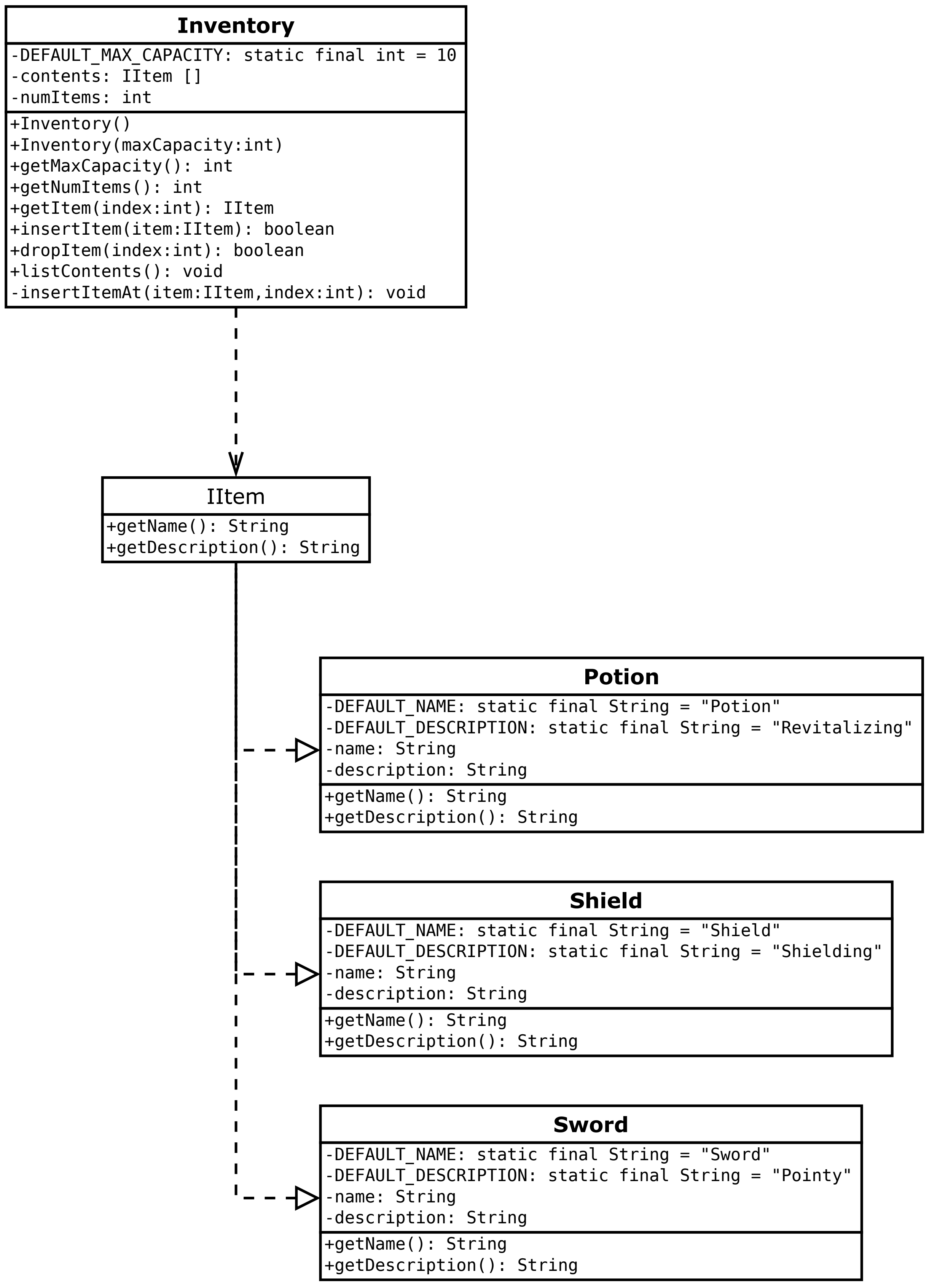
In order to facilitate this behaviour, you may wish to define one extra function:

`void insertItemAt(IItem item, int index)` A private helper function. This will be called by `insertItem`, not the user. It will handle the process of reordering the items in the array given that the new item is to be inserted at `index`

Notes

For testing the intermediate stages of your program, you should write a main class which defines a series of simple operations (adding and deleting objects from the array). However, for the final product, a main class for testing the software will be provided. It is therefore important to adhere to the structure outlined in this problem sheet. Where you feel an extra function may be required you are free to implement one. However, all functions listed here must be provided in order for the final tests to run (except where certain functions have been labelled as optional).

UML Diagram



Inventory
-DEFAULT_MAX_CAPACITY: static final int = 10
-contents: IItem []
-numItems: int
+Inventory()
+Inventory(maxCapacity:int)
+getMaxCapacity(): int
+getNumItems(): int
+getItem(index:int): IItem
+insertItem(item:IItem): boolean
+dropItem(index:int): boolean
+listContents(): void
-insertItemAt(item:IItem,index:int): void
IItem
+getName(): String
+getDescription(): String
Potion
-DEFAULT_NAME: static final String = "Potion"
-DEFAULT_DESCRIPTION: static final String = "Revitalizing"
-name: String
-description: String
+getName(): String
+getDescription(): String
Shield
-DEFAULT_NAME: static final String = "Shield"
-DEFAULT_DESCRIPTION: static final String = "Shielding"
-name: String
-description: String
+getName(): String
+getDescription(): String
Sword
-DEFAULT_NAME: static final String = "Sword"
-DEFAULT_DESCRIPTION: static final String = "Pointy"
-name: String
-description: String
+getName(): String
+getDescription(): String