Reflection and Attributes

Problem 1. Harvesting Fields

Provided skeleton.

You are given a **RichSoilLand** class with lots of fields (look at the provided skeleton). Like a good farmer as what you are, you must harvest them. Harvesting means that you must print each field in a certain format (see output).

Input

You will receive a maximum of 100 lines with one of the following commands:

- **private -** print all private fields
- **protected** - print all protected fields
- **public** - print all public fields
- **all** - print ALL declared fields
- **HARVEST** - end the input

Output

For each command, you must print the fields that have the given access modifier as described in the input section. The format in which the fields should be printed is:

"<access modifier> <field type> <field name>"

Examples

Input	Output
protected HARVEST	protected String testString protected Double aDouble protected Byte testByte protected StringBuilder aBuffer protected BigInteger testBigNumber protected Single testFloat protected Object testPredicate protected Object fatherMotherObject protected String moarString protected Exception inheritableException protected Stream moarStreamz
private public private HARVEST	private Int32 testInt private Int64 testLong private Calendar aCalendar private Char testChar private BigInteger testBigInt private Thread aThread private Object aPredicate private Object hiddenObject private String anotherString private Exception internalException private Stream secretStream public Double testDouble public String aString public StringBuilder aBuilder

	public Int16 testShort public Byte aByte public Single aFloat public Thread testThread public Object anObject public Int32 anotherIntBitesTheDust public Exception justException public Stream aStream private Int32 testInt private Int64 testLong private Calendar aCalendar private Char testChar private BigInteger testBigInt private Thread aThread private Object aPredicate private Object hiddenObject private String anotherString private Exception internalException private Stream secretStream
all HARVEST	private Int32 testInt public Double testDouble protected String testString private Int64 testLong protected Double aDouble public String aString private Calendar aCalendar public StringBuilder aBuilder private Char testChar public Int16 testShort protected Byte testByte public Byte aByte protected StringBuilder aBuffer private BigInteger testBigInt protected BigInteger testBigNumber protected Single testFloat public Single aFloat private Thread aThread public Thread testThread private Object aPredicate protected Object testPredicate public Object anObject private Object hiddenObject protected Object fatherMotherObject private String anotherString protected String moarString public Int32 anotherIntBitesTheDust private Exception internalException protected Exception inheritableException public Exception justException public Stream aStream protected Stream moarStreamz private Stream secretStream

Problem 2. Black Box Integer

Provided skeleton.

You are helping a buddy of yours who is still in the OOP Basics course - his name is Peshoslav (not to be mistaken with real people or trainers). He is rather slow and made a class with all private members. Your tasks are to instantiate an object from his class (always with start value 0) and then invoke the different methods it has. Your restriction is to not change anything in the class itself (consider it a black box). You can look at his class but don't touch anything! The class itself is called **BlackBoxInt** it is a wrapper for the **int** primitive.

The methods this class has are:

- Add(int)
- Subtract(int)
- Multiply(int)
- Divide(int)
- LeftShift(int)
- RightShift(int)

Input

The input will consist of lines in the form:

<method name>_<value>

For instance: **Add_115**

Input will always be valid and in the format described, so there is no need to check it explicitly. You stop receiving input when you encounter the command "**END**".

Output

Each command (except the **END** one) should print the current value of **innerValue** of the BlackBoxInt object you instantiated. Don't cheat by overriding ToString() in the class - you must get the value from the **private** field.

Examples

Input	Output
Add_999999 Subtract_19 Divide_4 Multiply_2 RightShift_1 LeftShift_3 END	999999 999980 249995 499990 249995 1999960

Problem 3. BarracksWars - A New Factory

You are given a small console based project called Barracks (the code for it is included in the provided skeleton).

The general functionalities of the project are adding new units to its repository and printing a report with statistics about the units currently in the repository. First let's go over the original task before the project was created:

Input

The input consists of commands each on a separate line. Commands that execute the functionality are:

- **add** <**Archer**/**Swordsman**/**Pikeman**/**{…}**> - adds a unit to the repository.
- **report** - prints a lexicological ordered statistic about the units in the repository.
- **fight** - ends the input.

Output

Each command except **fight** should print output on the console.

- **add** should print: "<**Archer/Swordsman/Pikeman/{…}> added!**"
- **report** should print all the info in the repository in the format: "<**UnitType**> **->** <**UnitQuantity**>", sorted by UnitType

Constraints

- Input will consist of no more than **1000** lines
- **report** command will never be given before any valid add command was provided

Your task

1) You have to **study the code of the project and figure out how it works**. However, there are parts of it that are not implemented (left with TODOs). You must implement the functionality of the **CreateUnit** method in the **UnitFactory** class so that it creates a unit based on the unit type received as parameter. Implement it in such a way that whenever you add a new unit it will be creatable without the need to change anything in the **UnitFactory** class (*psst - use reflection*). You can use the approach called **Simple Factory**.

2) Add two new unit classes (there will be tests that require them) - **Horseman** with 50 health and 10 attack and **Gunner** with 20 health and 20 attack.

If you complete everything correctly for this problem, you should add code only inside the **Factories** and **Units** folders.

Examples

Input	Output
add Swordsman add Archer add Pikeman report add Pikeman add Pikeman report fight	Swordsman added! Archer added! Pikeman added! Archer -> 1 Pikeman -> 1 Swordsman -> 1 Pikeman added! Pikeman added! Archer -> 1 Pikeman -> 3 Swordsman -> 1
add Pikeman add Pikeman add Gunner add Horseman add Archer add Gunner add Gunner add Horseman report fight	Pikeman added! Pikeman added! Gunner added! Horseman added! Archer added! Gunner added! Gunner added! Horseman added! Archer -> 1 Gunner -> 3 Horseman -> 2 Pikeman -> 2

Problem 4. BarracksWars - The Commands Strike Back

Provided skeleton.

As you might have noticed commands in the project from Problem 3 are implemented via a switch case with method calls in the **Engine** class. Although this approach works it is flawed when you add a new command because you have to add a new case for it. In some projects, you might not have access to the engine and this would not work. Imagine this project will be outsourced and the outsourcing firm will not have access to the engine. Make it so whenever they want to add a new command they won't have to change anything in the **Engine.**

To do so employ the design pattern called **Command Pattern**. We've done this in the **BashSoft Lab** and you can look there for tips too. Use the provided **Executable** interface as a frame for the command classes. Put the new command classes in the provided **commands** package inside **core**. You can also make a Command Interpreter to decouple that functionality from the Engine. Here is how the base (abstract) command should look like:

```
public abstract class Command : IExecutable
{
    private string[] data;
    private IRepository repository;
    private IUnitFactory unitFactory;

    protected Command(string[] data, IRepository repository, IUnitFactory unitFactory)
    {
        this.Data = data;
        this.Repository = repository;
        this.UnitFactory = unitFactory;
    }

    protected string[] Data
    {
        get { return this.data; }
        private set { this.data = value; }
    }

    protected IRepository Repository
    {
        get{ return this.repository; }
        private set{ this.repository = value; }
    }

    protected IUnitFactory UnitFactory
    {
        get { return this.unitFactory; }
        private set { this.unitFactory = value; }
    }

    public abstract string Execute();
}
```

Notice how all commands that extend this one will have both a Repository and a **UnitFactory** although not all of them need these. Leave it like this for this problem, because for the reflection to work we need all constructors to accept the same parameters. We will see how to go around this issue in problem 5.

Once you've implemented the pattern add a new command. It will have the following syntax:

- **retire** <**UnitType**> - All it has to do is remove a unit of the provided type from the repository.
    - If there are no such units currently in the repository print: "**No such units in repository.**"

- If there is such a unit currently in the repository, print: "**<UnitType> retired!**"

To implement this command, you will also have to implement a corresponding method in the **UnitRepository**.

If you do everything correctly for this problem, you should write/refactor code only in the **Core** and **Data** packages.

Examples

Input	Output
retire Archer add Pikeman add Pikeman add Gunner add Horseman add Archer add Gunner add Gunner add Horseman report retire Gunner retire Archer report retire Swordsman retire Archer fight	No such units in repository. Pikeman added! Pikeman added! Gunner added! Horseman added! Archer added! Gunner added! Gunner added! Horseman added! Archer -> 1 Gunner -> 3 Horseman -> 2 Pikeman -> 2 Gunner retired! Archer retired! Archer -> 0 Gunner -> 2 Horseman -> 2 Pikeman -> 2 No such units in repository. No such units in repository.

Problem 5. * BarracksWars - Return of the Dependencies

In the final part of this epic problem trilogy we will resolve the issue where all Commands received all utility classes as parameters in their constructors. We can accomplish this by using an approach called **dependency injection container**. This approach is used in many frameworks.

We will do a little twist on that approach. Remove all fields from the abstract command except the **data.** Instead put whatever fields each command needs in the concrete class. Create an attribute called **Inject** and make it so it can be used only on fields. Put the attribute over the fields we need to set trough reflection. Once you've prepared all of this, write the necessary reflection code in the **Command Interpreter** (which you should have refactored out from the engine in problem 4).

You can use the same example as in Problem 4 to check if you completed the task correctly.

Problem 6. Traffic Lights

Implement a simple state machine in the form of a traffic light. Every traffic light has three possible signals - red, green and yellow. Each traffic light can be updated, which changes the color of its signal (e.g. if it is currently red, it changes to green, if it is green it changes to yellow). The order of signals is red -> green -> yellow -> red and so on.

On the first line you will be given multiple traffic light signals in the format "Red Green Yellow". You need to make as many traffic lights as there are signals in the input.

On the second line, you will receive the **n** number of times you need to change each traffic light's signal.

Your output should consist of **n** number of lines, including each updated traffic light's signal. To better understand the problem, see the example below.

Examples

Input	Output
`Green Red Yellow` 4	Yellow Green Red Red Yellow Green Green Red Yellow Yellow Green Red

Problem 7. Inferno Infinity

If you've been involved with the creation of Inferno III last year, you may be informed of the disastrous critic reception it has received. Nevertheless, your company is determined to satisfy its fan base, so a sequel is coming and yeah, you will develop the crafting module of the game using the latest OOP trends.

You have three different weapons (Axe, Sword and Knife) which have base stats and a name. The base stats are min damage, max damage and number of sockets (sockets are basically holes, in which you can insert gems). Below are the base stats for the three weapon types:

- Axe (5-10 damage, 4 sockets)
- Sword (4-6 damage, 3 sockets)
- Knife (3-4 damage, 2 sockets)

What's more, every weapon comes with a different level of rarity (how rare it is to come across such an item). Depending on its rarity, a weapon's maximum and minimum damage can be modified.

- Common (increases damage x1)
- Uncommon (increases damage x2)
- Rare (increases damage x3)
- Epic (increases damage x5)

So a Common Axe would have its damage modified in the following way: minimum damage = 5 *1, maximum damage = 10 *1. Whereas an Epic Axe would look like this: minimum damage = 5 * 5, maximum damage = 10 * 5.

Additionally, every weapon provides a bonus to three magical stats - strength, agility and vitality. At first the bonus of every magical stat is zero and can be increased with gems which are inserted into the weapon.

Every gem provides a bonus to all three of the magical stats. There are three different kind of gems:

- Ruby (+7 strength, +2 agility, +5 vitality)
- Emerald (+1strength, +4 agility, +9 vitality)
- Amethyst (+2 strength, +8 agility, +4 vitality)

Every point of strength adds +2 to min damage and +3 to max damage. Every point of agility adds +1 to min damage and +4 to max damage. Vitality does not add damage.

Furthermore, every gem comes in different levels of clarity (basically level of quality). Depending on its level of clarity, a gem's stats can be modified in the following manner:

- Chipped (increases each stat by 1)
- Regular (increases each stat by **2**)
- Perfect (increases each stat by 5)
- Flawless (increases each stat by 10)

So a Chipped Amethyst will have its stats modified like this: strength = 2 + 1, agility = 8 + 1, vitality = 4 + 1. Whilst a Perfect Emerald would look like this: strength = 1 + 5, agility = 4 + 5, vitality = 9 + 5.

Your job is to implement the functionality to read some weapons from the console and optionally to insert or remove gems at different socket indexes until you receive the END command.

Also, upon the **Print** command, in order to print correct final stats for a given weapon, first calculate the weapon's **base stats** taking into account **its type and rarity**. Afterwards, calculate the stats of each of its gems based on their **clarity** and finally add everything together. For the specific format of printing refer to the Output section.

Note

If you add gem on top of another, just overwrite it. If you add a gem to an invalid index, nothing happens. If you try to remove a gem from an empty socket or from invalid index, nothing happens. Upon receiving the END command print the weapons in order of their appearance in the format provided below.

Input

Each line consists of three types of commands in which the tokens are separated by ";".

Command types:

- Create;{weapon type};{weapon name}
- Add;{weapon name};{socket index};{gem type}
- Remove;{weapon name};{socket index}
- Print;{weapon name}

Output

Print weapons in the given format:

"{weapon's name}: {min damage}-{max damage} Damage, +{points} Strength, +{points} Agility, +{points} Vitality"

Examples

Input	Output

`Create;Common Axe;Axe of Misfortune`
`Add;Axe of Misfortune;0;Chipped Ruby`
`Print;Axe of Misfortune`
`END`
	`Axe of Misfortune: 24-46 Damage, +8 Strength, +3 Agility, +6 Vitality`

`Create;Common Axe;Axe of Misfortune`
`Add;Axe of Misfortune;0;Flawless Ruby`
`Remove;Axe of Misfortune;0`
`Print;Axe of Misfortune`
`END`
	`Axe of Misfortune: 5-10 Damage, +0 Strength, +0 Agility, +0 Vitality`

Problem 8. Create Custom Class Attribute

Create a custom attribute that can be applied to classes and can be accessed at runtime. The attribute type elements it should contain are author, revision, description and reviewers. Apply the attribute to the Weapon class you have created for the Inferno Infinity problem. Provide these **exact** values:

- author = "Pesho"
- revision = 3
- description = "Used for C# OOP Advanced Course - Enumerations and Attributes."
- reviewers = "Pesho", "Svetlio"

Implement additional commands for extracting different attribute values:

- Author - prints the author of the class
- Revision - prints the revision of the class
- Description - prints the class description
- Reviewers - prints the reviewers of the class

Examples

Input	Output

Author
Revision
Description
Reviewers
END
	`Author: Pesho`
	`Revision: 3`
	Class description: Used for C# OOP Advanced Course - Enumerations and Attributes.
	Reviewers: Pesho, Svetlio

Problem 9. *Refactoring - Bonus

Refactor your Inferno Infinity problem code according to all HQC standards.

- Think about the proper naming of all your variables, methods, classes and interfaces.
- Review all of your methods and make sure they are doing only one highly concrete thing.
- Review your class hierarchy and make sure you have no duplicating code.
- Consider making your classes less dependent of each other. If you have the **new** keyword anywhere inside the body of a non-factory or main class, think about how to remove it. Read about **dependency injection.**
- Consider adding independent classes for reading input and writing output.
- Create repository class that stores all weapon data.
- Create an engine, weapon creator and so on. Try using design patterns like command and factory.
- Make you classes **highly cohesive** and **loosely coupled**.