Dragon Era

Our world is a remarkable set of events. All its way to nowadays it's been through different ages, each of them with its own ecosystem, habitats and level of evolution. And the most unbelievable thing is all these events repeat the pattern infinite times. The world as we know it is coming to an end and there's nothing we can do about it, except that only one person is the chosen one to be placed in the event loop and result in a different age at the exact moment of this world iteration's end.

Flash! It's gone! Don't worry, you were the chosen one to travel through the spacetime. You are now in an epoch unknown to your previous civilization, and thus, to you. The dragon era!

If you want this world iteration to last more, you need to work for it. The dragon era is time boxed and is very predictable, unfortunately until now there were no one who could predict it. As you are technically advanced from your past epoch, you compute the result of the era, which will be very useful in the jump to the next era.

Here are the rules: The era starts with some dragons spawned from unknown source. Each of them has laid some eggs. The era has fixed time of years. Next, for each passing year, these dragons will age and lay eggs. The eggs can hatch and the dragon now has children.

There are several rules to breeding dragons, though.

Dragons lay eggs only when they are 3 or 4 years old. Every dragon lays one egg a year. The eggs hatch after two years. When a dragon gets 6 years old, one dies. A dead dragon cannot hatch its eggs.

There are also year types. In a normal year, all eggs that will hatch will produce one dragon. A bad year means no dragons per egg, and a good year will give you two dragons instead of one.

When the age passes you have to print all alive dragons along with their children ordered by appearance order (the ones that were hatched first will be printed first). Let's say we have the first and the only dragon you have started with - alive. It has two children. The first children hatched an egg, and the second has not. The final print should be:

```
Dragon_1
 Dragon_1/Dragon_2
  Dragon_1/Dragon_2/Dragon_4
 Dragon_1/Dragon_3
```

Because first Dragon_1 appeared. Then it hatched its first child – Dragon_2. Because Dragon_2's parent is Dragon_1 we keep the pattern Parent_id/SubParent_id/SubParent_id/LastChild_id, its name is Dragon_1/Dragon_2

Then it hatched its second child. It's the third dragon overall in our world (Dragon_3). Again keeping the pattern, its name is Dragon_1/Dragon_3.

Then Dragon_2 hatches its first child. Because it's the fourth dragon overall its name is Dragon_4. Because Dragon_4's parent is Dragon_2 and Dragon_2's parent is Dragon_1, the final name of Dragon_4 is Dragon_1/Dragon_2/Dragon_4

Input

- On the first line of input you receive the number **n** – the number of dragons you start with.

- On the next **n** lines you will receive an integer – the number of starting eggs for each dragon.
- Next you receive the number **y –** the number of years your hatchery will work
- On each of the next **y** lines you will receive the year type **Bad, Normal or Good**

The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

Each dragon on separate line indented by parents * 2 spaces.

Constraints

Dragons count will always be a positive integer between 1 and 2,000,000,000

Examples

Input	Output

`2`
`3`
`3`
`5`
`Normal`
`Normal`
`Normal`
`Normal`
`Normal`
	`Dragon_1`
	`  Dragon_1/Dragon_3`
	`  Dragon_1/Dragon_4`
	`  Dragon_1/Dragon_5`
	`  Dragon_1/Dragon_9`
	`Dragon_2`
	`  Dragon_2/Dragon_6`
	`  Dragon_2/Dragon_7`
	`  Dragon_2/Dragon_8`
	`  Dragon_2/Dragon_10`

`2`
`10`
`10`
`5`
`Normal`
`Bad`
`Normal`
`Normal`
`Normal`
	`Dragon_1`
	`  Dragon_1/Dragon_3`
	`Dragon_2`
	`  Dragon_2/Dragon_4`

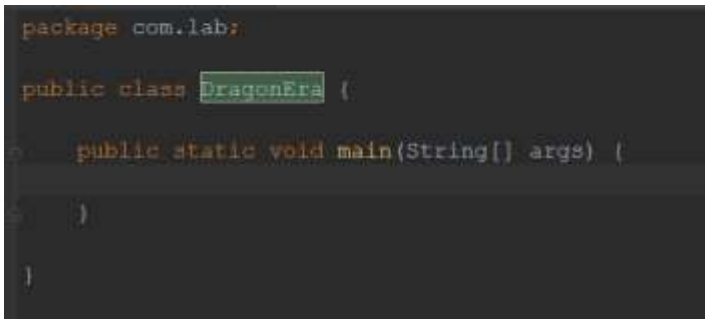
`3`
`1`
`1`
`1`
`7`
`Good`
`Good`
`Good`
`Good`
`Good`
`Bad`
`Normal`
	`  Dragon_1/Dragon_4`
	`    Dragon_1/Dragon_4/Dragon_16`
	`  Dragon_1/Dragon_5`
	`    Dragon_1/Dragon_5/Dragon_17`
	`  Dragon_1/Dragon_10`
	`  Dragon_1/Dragon_11`
	`  Dragon_2/Dragon_6`
	`    Dragon_2/Dragon_6/Dragon_18`
	`  Dragon_2/Dragon_7`
	`    Dragon_2/Dragon_7/Dragon_19`
	`  Dragon_2/Dragon_12`
	`  Dragon_2/Dragon_13`
	`  Dragon_3/Dragon_8`
	`    Dragon_3/Dragon_8/Dragon_20`
	`  Dragon_3/Dragon_9`
	`    Dragon_3/Dragon_9/Dragon_21`
	`  Dragon_3/Dragon_14`

	Dragon_3/Dragon_15

SPOILER!

Task 1. Create an empty console application

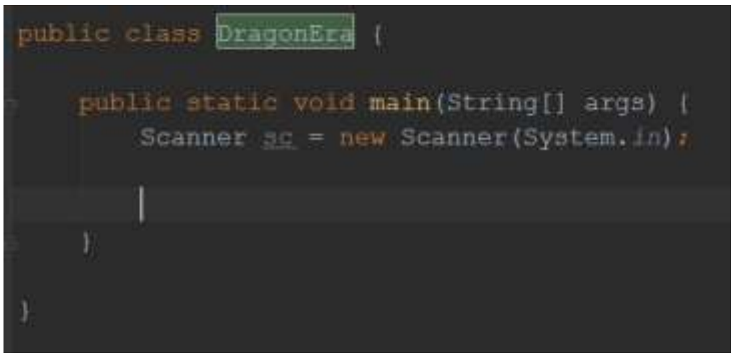
We will use IntelliJ IDEA as an Integrated Development Environment with JDK 1.8. Let's create a console project called DragonEra. The final result should be something like



```
package com.lab;

public class DragonEra {

    public static void main(String[] args) {

    }

}
```

Task 2. Start getting input

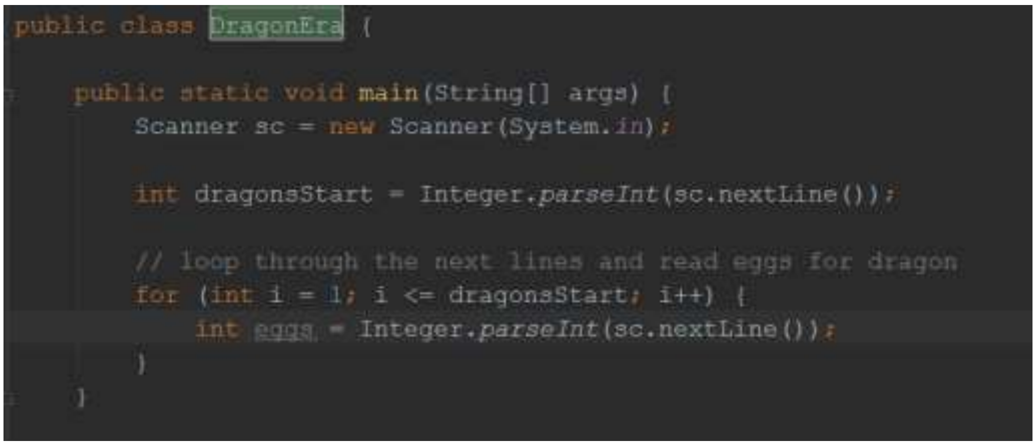
We have to use a wrapper that reads the standard input, because this is a console application. Let's use Scanner for it and inject the Scanner the standard input (stdin -> System.in). We will call it "sc"



```
public class DragonEra {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

    }

}
```

On the first line we will receive the number of dragons the era starts with. Then we will receive as much lines as dragons we have, so we need to read eggs per dragons.

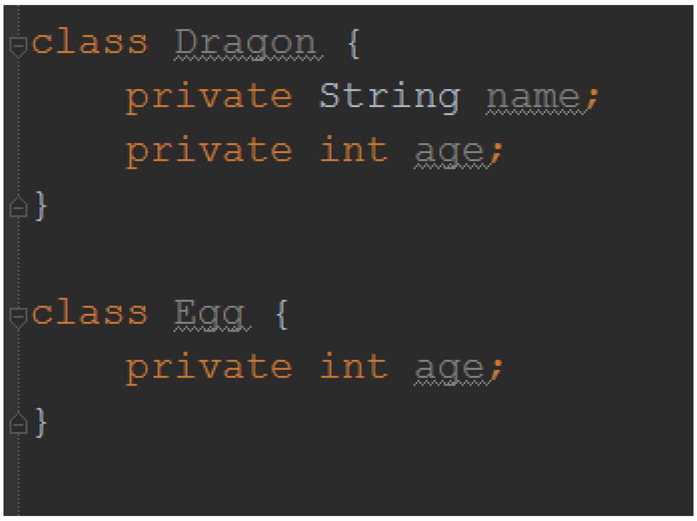
The numeric input should be casted to numeric type in order to use in loops or do mathematical operations with it.



```
public class DragonEra {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int dragonsStart = Integer.parseInt(sc.nextLine());

        // loop through the next lines and read eggs for dragon
        for (int i = 1; i <= dragonsStart; i++) {
            int eggs = Integer.parseInt(sc.nextLine());
        }
    }
```

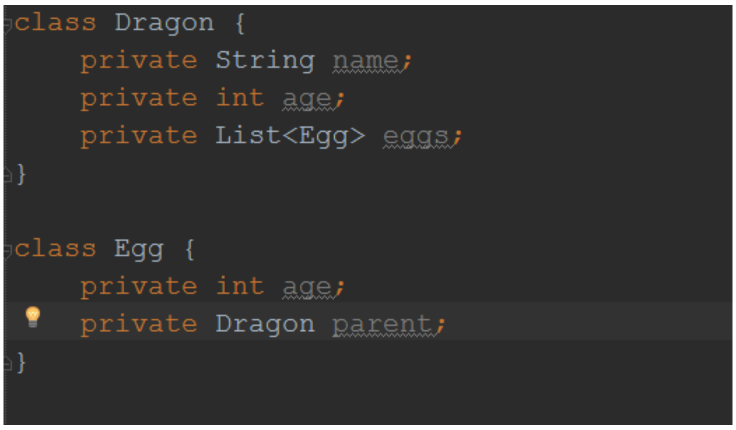
Task 3. The input is not enough. We need dragons and eggs.

Well, we already have the dragons' count we start with and the eggs for each dragon on each iteration, but we need to persist them somewhere. Because each egg and dragon will have name and age, let's define a template for this. The most logical thing is a class for a Dragon and a class for an Egg.



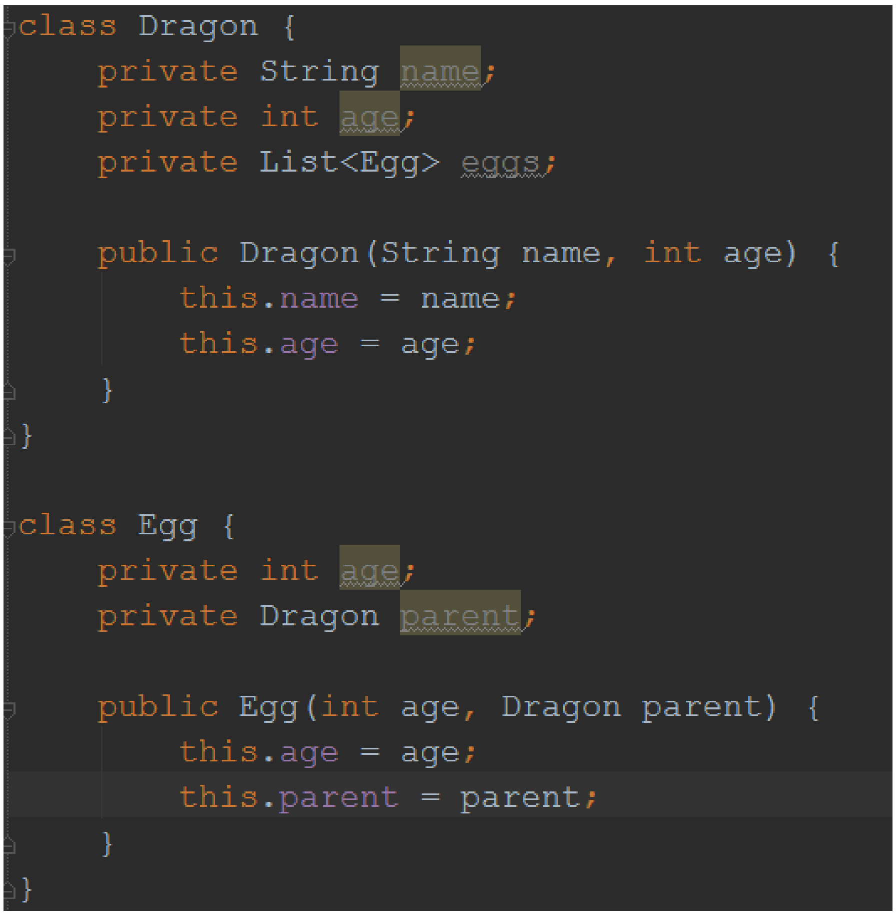
```
class Dragon {
    private String name;
    private int age;
}

class Egg {
    private int age;
}
```

We now have the templates and we need to define relations. Each dragon has many eggs and each egg is laid by only one dragon.



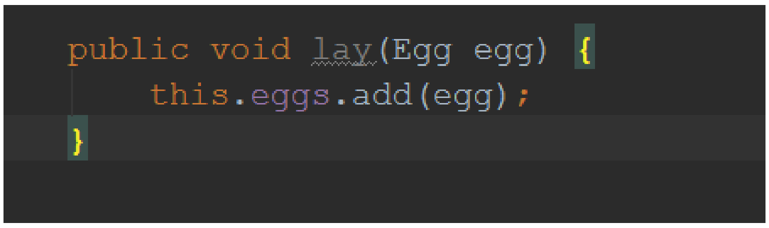
```
class Dragon {
    private String name;
    private int age;
    private List<Egg> eggs;
}

class Egg {
    private int age;
    private Dragon parent;
}
```

When a dragon is born it has assigned a name and starts with age 0. When an egg is laid it starts with age 0 and its laid by its parent.



```
class Dragon {
    private String name;
    private int age;
    private List<Egg> eggs;

    public Dragon(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class Egg {
    private int age;
    private Dragon parent;

    public Egg(int age, Dragon parent) {
        this.age = age;
        this.parent = parent;
    }
}
```

As the eggs are laid after the dragon is born (the object is created) we need to expose a method which makes the relation between the newly laid egg and the collection of eggs. Let's just define a method lay(Egg egg) which adds the egg to the collection.



```
public void lay(Egg egg) {
    this.eggs.add(egg);
}
```

Task 4. Use the newly defined classes to store the received input

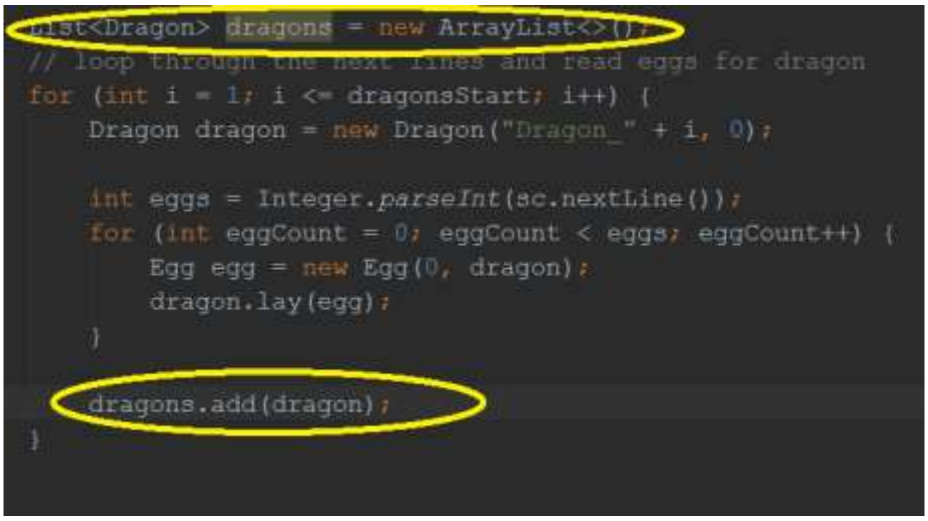
Let's go back to our main method where we have parsed the input. On each dragon iteration we need to create a dragon and we need to make as many eggs as the dragon has in order to make him lay them.

Each dragon should follow the pattern of names Dragon_X/Dragon_Y. As we now create the first dragons in the hierarchy they will not have any parents (look at them as Adam and Eve). They will just be Dragon_1, Dragon_2, etc… When we loop from 1 to DragonsCount we can use the variable in the loop for these indices

```java
int dragonsStart = Integer.parseInt(sc.nextLine());

// loop through the next lines and read eggs for dragon
for (int i = 1; i <= dragonsStart; i++) {
    Dragon dragon = new Dragon("Dragon_" + i, 0);

    int eggs = Integer.parseInt(sc.nextLine());
    for (int eggCount = 0; eggCount < eggs; eggCount++) {
        Egg egg = new Egg(0, dragon);
        dragon.lay(egg);
    }
}
```

Task 5. Store the newly created objects

So far so good, we create dragons and attach to them eggs and vice versa, but after the loop ends – all of them are gone, but we need to print a hierarchical tree when we are done, so we need dragons stored somewhere. Let's create a collection of dragons before the first loop and attach each dragon to the collection once we are ready creating dragon with eggs

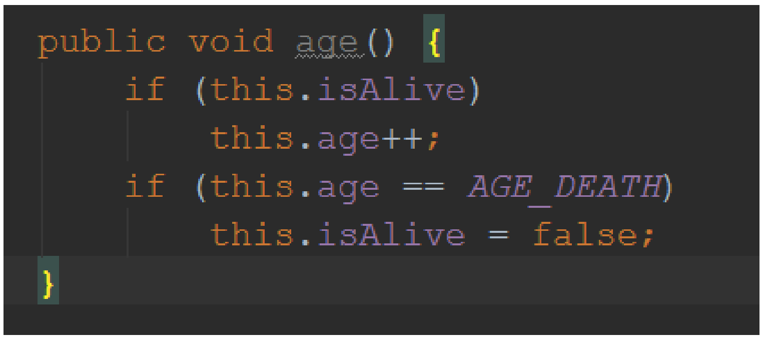


```java
List<Dragon> dragons = new ArrayList<>();
// loop through the next lines and read eggs for dragon
for (int i = 1; i <= dragonsStart; i++) {
    Dragon dragon = new Dragon("Dragon_" + i, 0);

    int eggs = Integer.parseInt(sc.nextLine());
    for (int eggCount = 0; eggCount < eggs; eggCount++) {
        Egg egg = new Egg(0, dragon);
        dragon.lay(egg);
    }

    dragons.add(dragon);
}
```

Task 6. Implement the main object logic

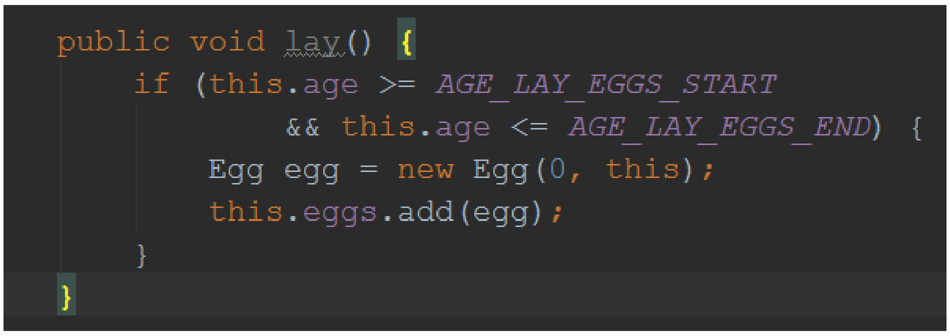
Before we proceed parsing the years as they are the next from the input, let's implement some of the main logic each object does.

1. The dragons need to age. When they age up to 6 years, they, unfortunately, die (poor, little dragons) so they stop aging. We need to know if dragon is alive or dead, so we will create a Boolean field for this. To follow some high quality principle we need to create constant for age of dying, let's call it AGE_DEATH. Then we need to define a method which increments the age if it's alive and takes care a dragon to die when it's necessary.



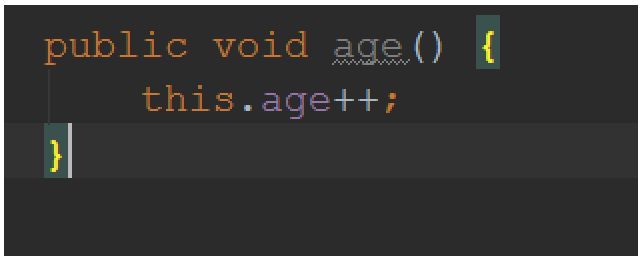
```
public void age() {
    if (this.isAlive)
        this.age++;
    if (this.age == AGE_DEATH)
        this.isAlive = false;
}
```

2. The dragons lay eggs on certain conditions not only when the era starts. They lay eggs when they are 3 or 4 years old. Let's create constants for these age ranges and call them AGE_LAY_EGGS_START and AGE_LAY_EGGS_END. Then overload the lay() method to not accept arguments and check its age instead.



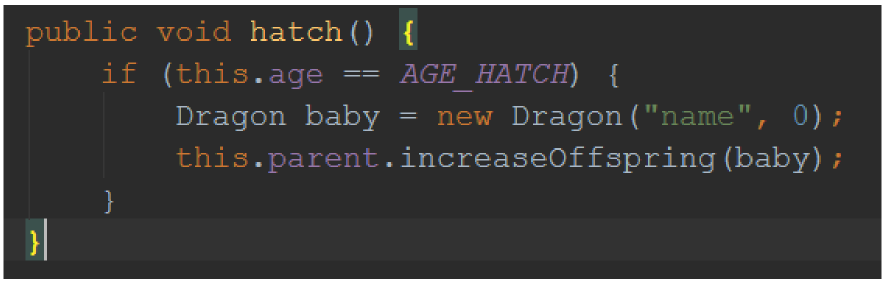
```
public void lay() {
    if (this.age >= AGE_LAY_EGGS_START
            && this.age <= AGE_LAY_EGGS_END) {
        Egg egg = new Egg(0, this);
        this.eggs.add(egg);
    }
}
```

3. The eggs age too, but they don't die. They either hatch on 2 years or get very old and hatch never. Let's make them age.



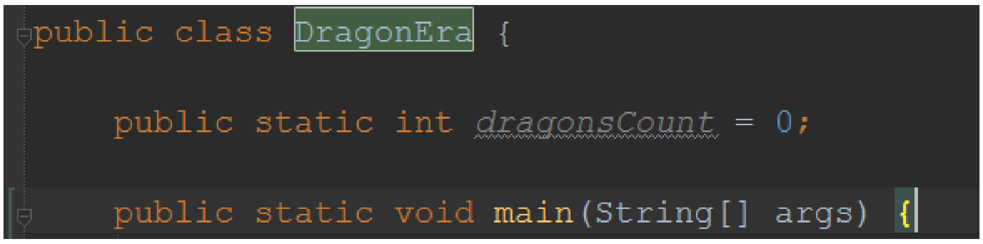
```
public void age() {
    this.age++;
}
```

4. Now they need to hatch. Let's declare a constant for age of hatching called AGE_HATCH and a method that checks the age and hatch the egg. A hatched egg produces a new baby dragon which is a child of the egg's parent. Thus, each dragon needs to keep its children in a separate collection. Make this collection and let's call it "children" and a method that adds to the collection called "increaseOffspring()". Now let's make the hatch method in the Egg class.



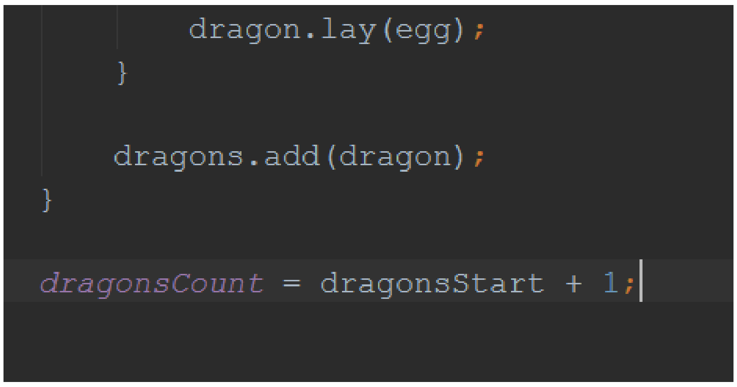
```
public void hatch() {
    if (this.age == AGE_HATCH) {
        Dragon baby = new Dragon("name", 0);
        this.parent.increaseOffspring(baby);
    }
}
```

4.1. There is a little problem here. "name" is no good name. Our constraints say that we need to call our child Parent_id/Dragon_id. We can have the parent name easily exposing the "name" field, but we don't know how many dragons we have so far. The easiest way is to have a static field that can be accessed from everywhere in order to keep track of which dragon we need to hatch. Let's go to our main class and declare the static field



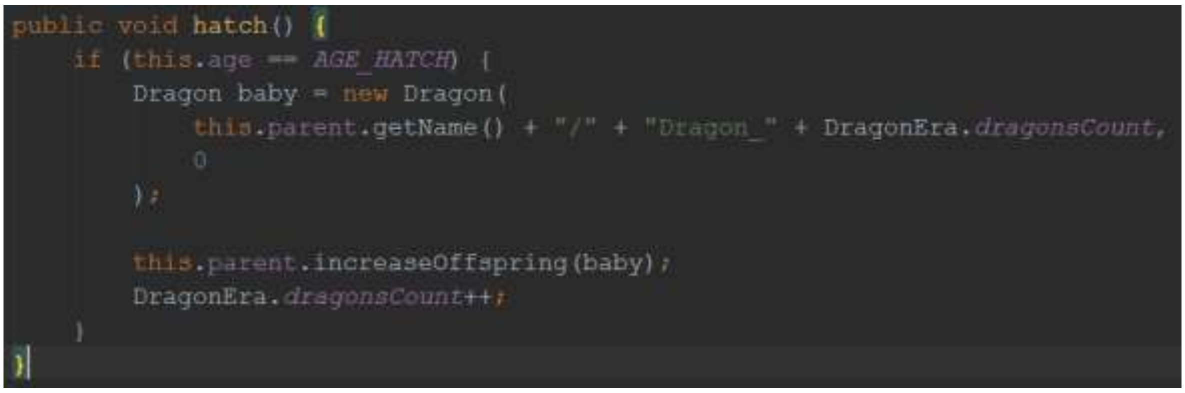
```java
public class DragonEra {

    public static int dragonsCount = 0;

    public static void main(String[] args) {
```

The field needs to increase everytime a dragon is born. It first will be the dragons we have created from the input, and the next dragon should be + 1. Let's say we received from the input 12 dragons. The next dragon we need to create is the 13-th dragon so. So let's assign "dragonsCount" to the count received from the input + 1



```java
            dragon.lay(egg);
        }

        dragons.add(dragon);
    }

    dragonsCount = dragonsStart + 1;
```

Now we can use this static field in our hatch method to name our new baby dragon. We need to increase the counter as well.

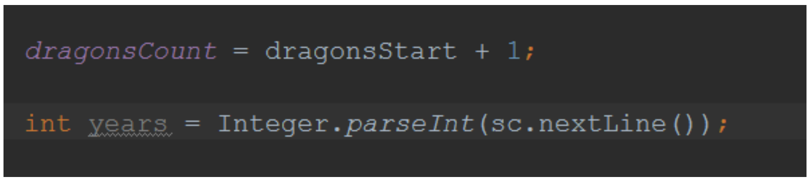


```java
public void hatch() {
    if (this.age == AGE_HATCH) {
        Dragon baby = new Dragon(
            this.parent.getName() + "/" + "Dragon_" + DragonEra.dragonsCount,
            0
        );

        this.parent.increaseOffspring(baby);
        DragonEra.dragonsCount++;
    }
}
```

Task 7. The year factor

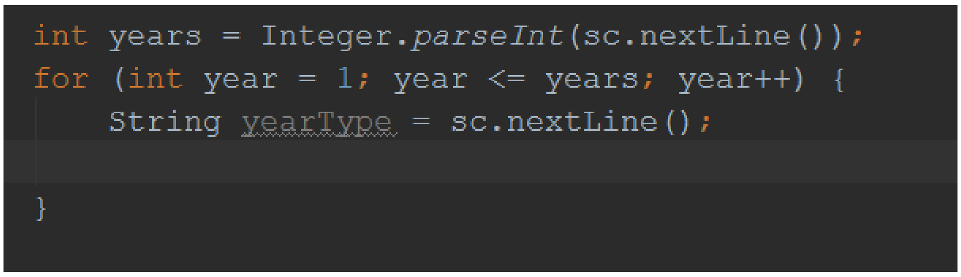
Now we are about to parse the years. When we stop receiving input for eggs we will receive a number – how many years the epoch will be. Then as many years as we have we will receive lines for each year, whether it's Normal, Bad or Good. Remember, when an egg is ready to hatch, at the year is Normal, it hatches one dragon. If it's Good year it hatches two dragons and if it's bad year it does not hatch.

Let's read how many years the epoch will last



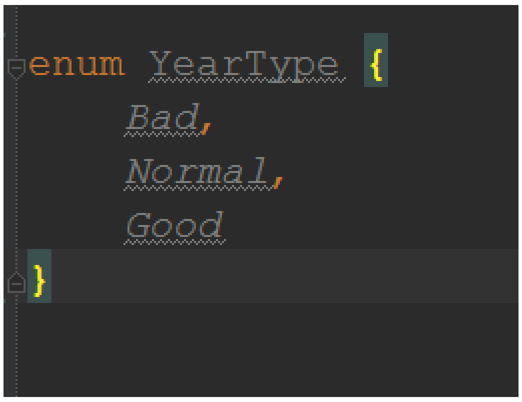
```java
dragonsCount = dragonsStart + 1;

int years = Integer.parseInt(sc.nextLine());
```

Now we need to check each year type and execute the actions.



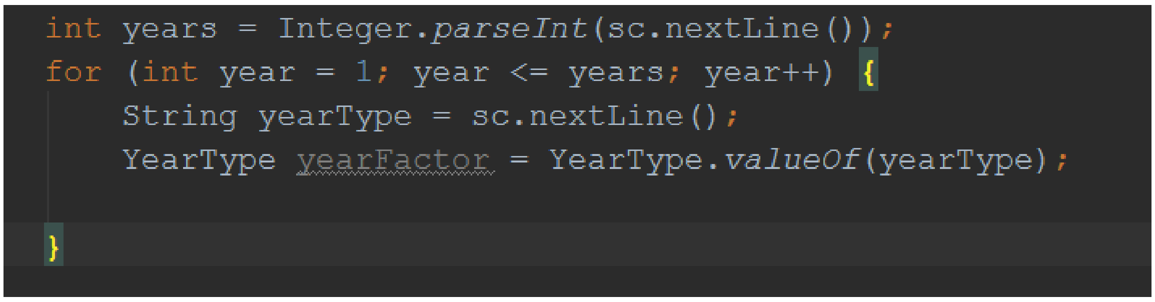
```java
int years = Integer.parseInt(sc.nextLine());
for (int year = 1; year <= years; year++) {
    String yearType = sc.nextLine();

}
```

Cool story. Now we can switch(yearType) and execute operation relevant to the year type. But it's too expensive from the matter of codebase and we just need to repeat code such as hatch(). Let's create enumeration. As we have a pattern – bad year -> 0 eggs, normal year -> 1 egg and good year -> 2 eggs we can make enumeration corresponding to these integers.



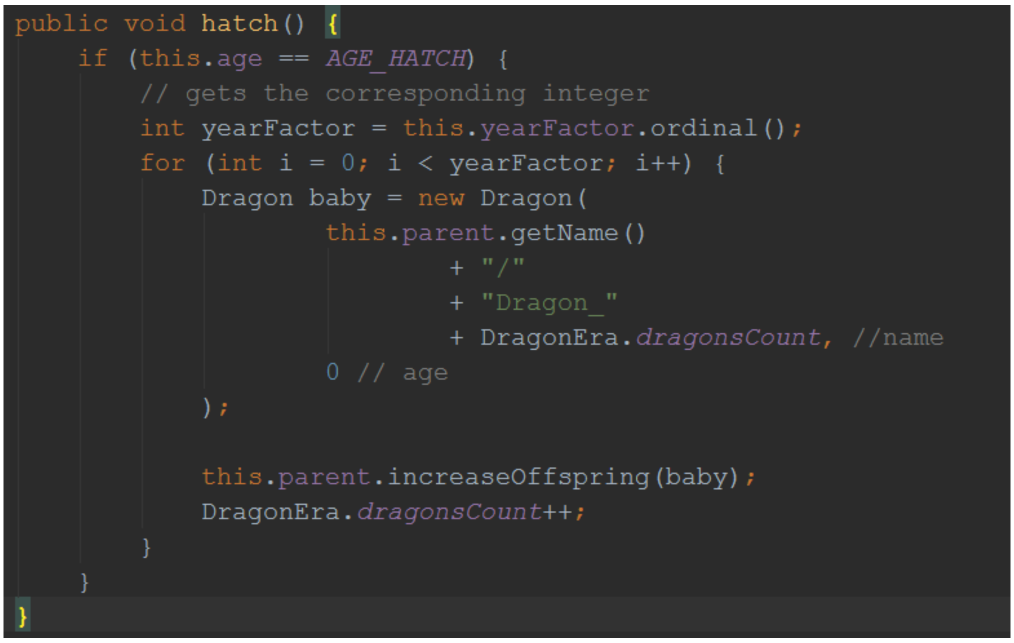
```java
enum YearType {
    Bad,
    Normal,
    Good
}
```

Luckily, Java allows us easily to parse string to enum if it corresponds to it. Let's change the String we receive to parsed enum type.



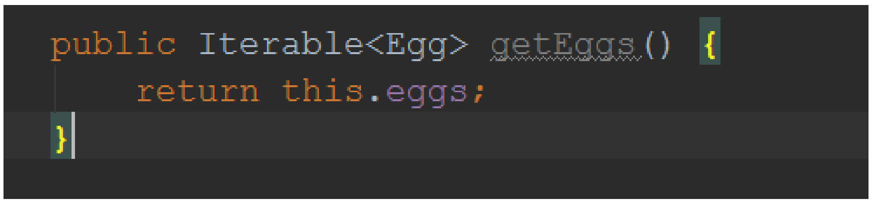
```java
int years = Integer.parseInt(sc.nextLine());
for (int year = 1; year <= years; year++) {
    String yearType = sc.nextLine();
    YearType yearFactor = YearType.valueOf(yearType);

}
```

Afterwards the Egg needs to know about the current YearType in order to perform different logic. Create a YearType field called "yearFactor" and an exposed setter so on each year we can tell the egg the type – called "setYearFactor". Then in the hatch method we can just make a loop from zero to the year type corresponding number. If it's a bad year it will perform the operations from 0 to 0 times e.g. it will not perform anything. If it's a normal year it will perform the operations from 0 to 1 times e.g. it will perform it only one time and if it's a good year it will perform it from 0 to 2 e.g. 2 times.



```java
public void hatch() {
    if (this.age == AGE_HATCH) {
        // gets the corresponding integer
        int yearFactor = this.yearFactor.ordinal();
        for (int i = 0; i < yearFactor; i++) {
            Dragon baby = new Dragon(
                    this.parent.getName()
                            + "/"
                            + "Dragon_"
                            + DragonEra.dragonsCount, //name
                    0 // age
            );

            this.parent.increaseOffspring(baby);
            DragonEra.dragonsCount++;
        }
    }
}
```

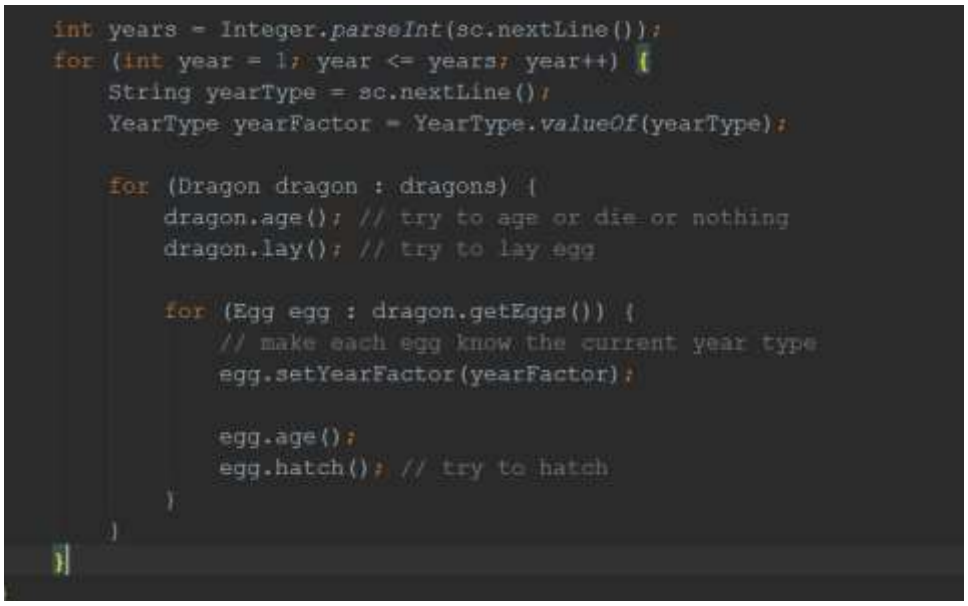
Now we are ready to execute some actions per each year. We need to iterate over our dragons collection and make each of them age and try to leg eggs. Also we need to iterate over their eggs and try them to hatch. In order to do this, we need to expose the collection of eggs. But just for iteration. Luckily there's an interface that exposes only the iterator and not the add/remove method – it's called Iterable<T> (corresponding interface in .NET Framework is IEnumerable<T>)



```java
public Iterable<Egg> getEggs() {
    return this.eggs;
}
```

Task 8. Make them age, lay and hatch!

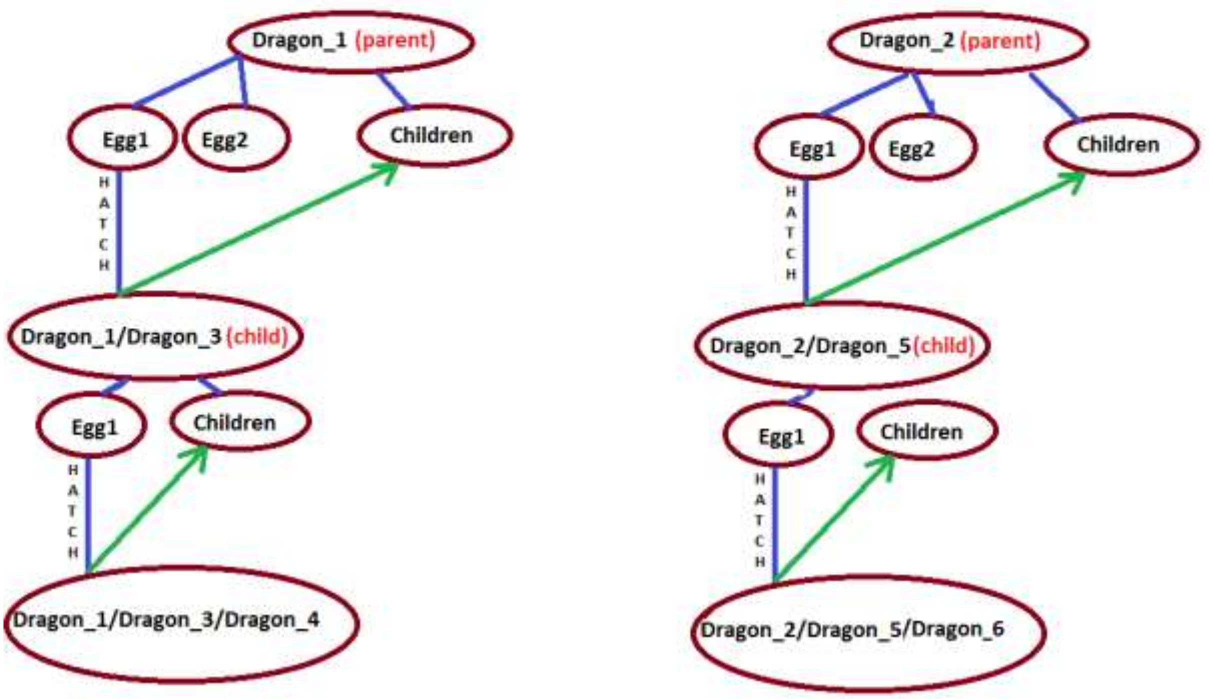
We are about to make the ecosystem move one step. All our dragons and their eggs will age one year, try to lay eggs and the eggs will try to hatch dragon.



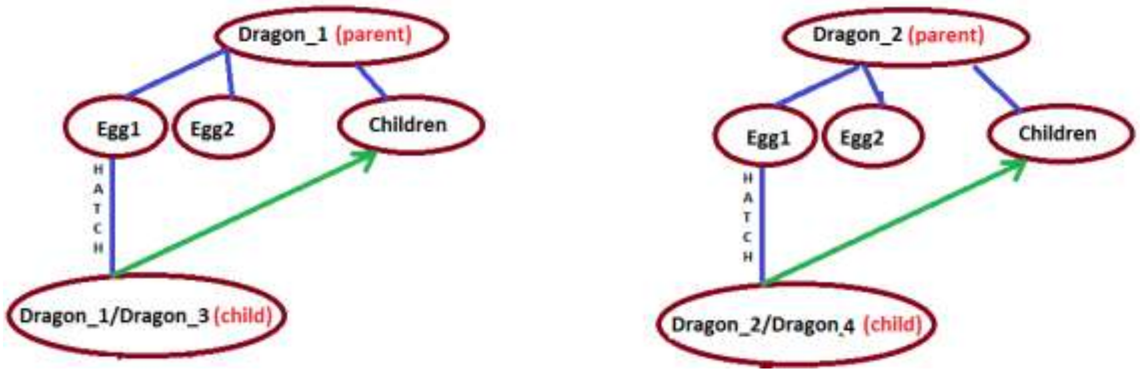
```java
int years = Integer.parseInt(sc.nextLine());
for (int year = 1; year <= years; year++) {
    String yearType = sc.nextLine();
    YearType yearFactor = YearType.valueOf(yearType);

    for (Dragon dragon : dragons) {
        dragon.age(); // try to age or die or nothing
        dragon.lay(); // try to lay egg

        for (Egg egg : dragon.getEggs()) {
            // make each egg know the current year type
            egg.setYearFactor(yearFactor);

            egg.age();
            egg.hatch(); // try to hatch
        }
    }
}
```

Cool story, eh? Our parent dragons age each year until they die, their lay eggs and their eggs age each year and maybe hatch baby dragons, but these baby dragons never age, nor lay eggs and the ecosystem easily stops and everyone dies young.

The ancestry should look like this:



Instead it now looks like this



Task 9. Recursion!

To go to the deepest level of children we need to use the so called recursion. A recursion is the process of repeating items in a self-similar way. In programming it's explained by a method calling itself.

```
public static void recursion() {
  System.out.println("a");
  recursion();
}
```

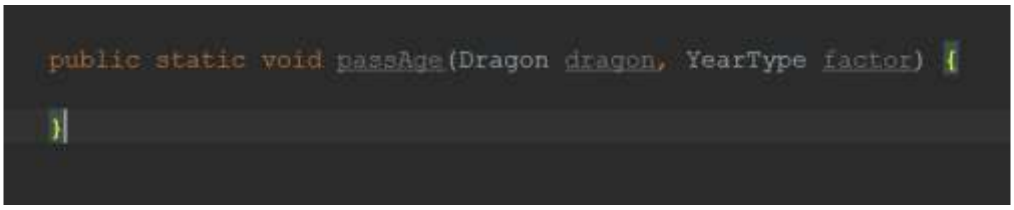
The “recursion()” method once called prints “a”, then it calls “recursion()” method (self) which prints “a”, then it calls “recursion()” method and so over until the world ends (or so your computer’s memory). It’s very important when using recursion to reach the end of the recursion when everything pops out.

In our case we need to recursively get each child of our dragons and make it age, get its children too and each child of its children to age, and each child of its children until we reach dragon that does not have any children so we will pop out the recursive calls.

The model should look like

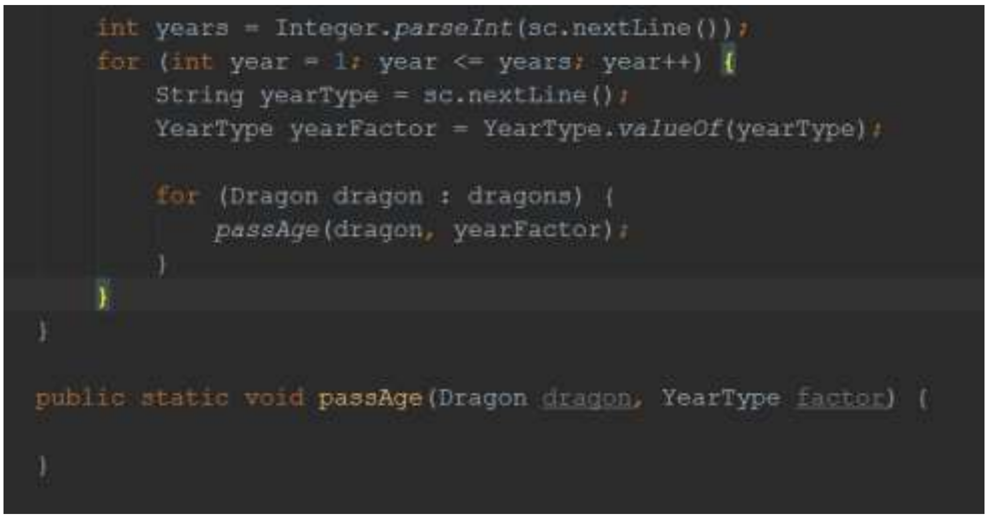
```
void recursion(Dragon dragon) {
    dragon.age();
    For Each Child (called dragonChild) in dragon.getChildren()
        recursion(dragonChild)
}
```

Let’s make in our main class a static method which we will use for recursive calls. It will be called each year and it will traverse the ancestry all the way down. Because of that we will name it “passAge()”, because when it exits from the recursion one year passes and everyone is aging. It receives a dragon and the current parsed year factor:



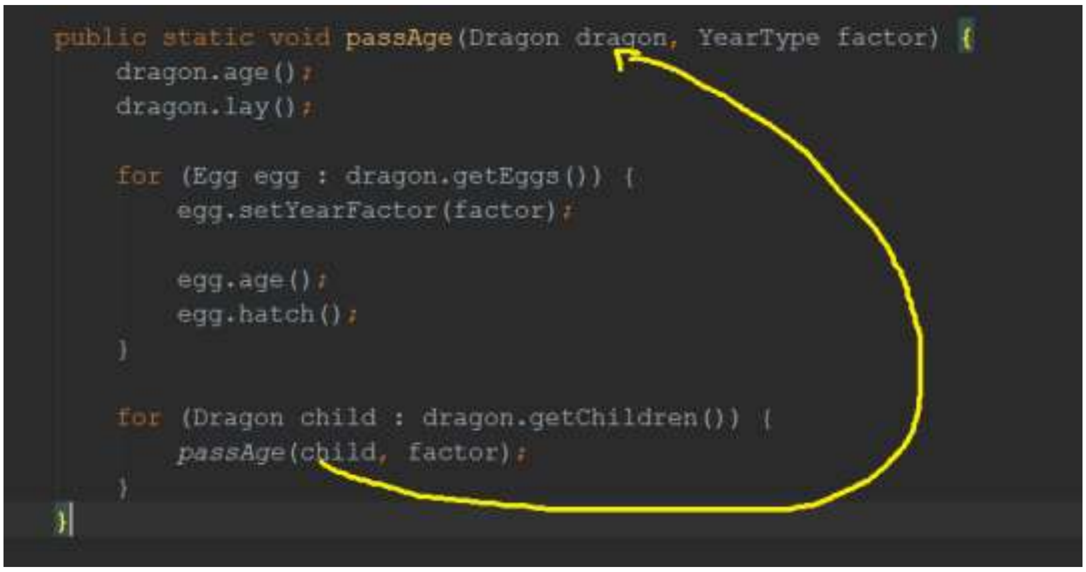
```
public static void passAge(Dragon dragon, YearType factor) {
}
```

Then we will initially call it in our loop for each year:



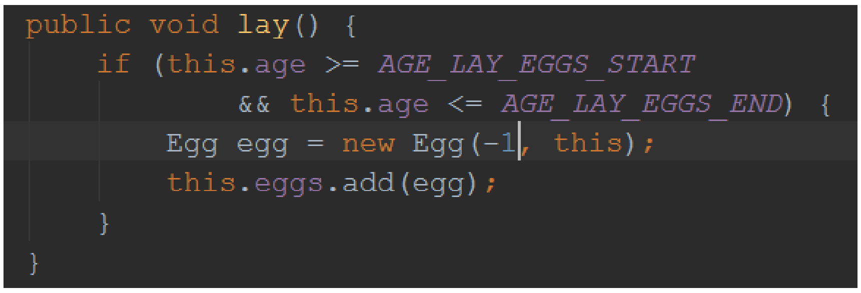
```
    int years = Integer.parseInt(sc.nextLine());
    for (int year = 1; year <= years; year++) {
        String yearType = sc.nextLine();
        YearType yearFactor = YearType.valueOf(yearType);

        for (Dragon dragon : dragons) {
            passAge(dragon, yearFactor);
        }
    }
}

public static void passAge(Dragon dragon, YearType factor) {

}
```

Let’s now move the recursive logic in “passAge()”.

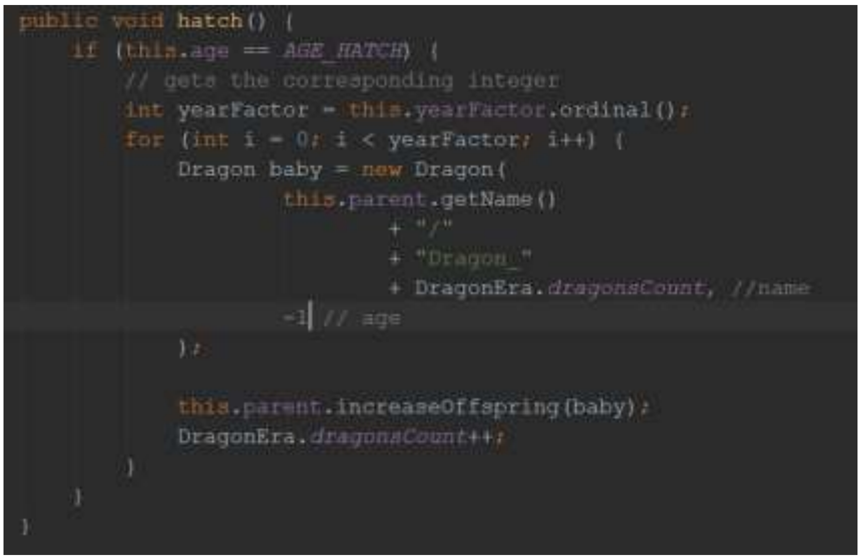


```java
public static void passAge(Dragon dragon, YearType factor) {
    dragon.age();
    dragon.lay();

    for (Egg egg : dragon.getEggs()) {
        egg.setYearFactor(factor);

        egg.age();
        egg.hatch();
    }

    for (Dragon child : dragon.getChildren()) {
        passAge(child, factor);
    }
}
```

We have a little problem here. If the dragon lay egg and then we loop over its eggs, the newly laid eggs will age one year, same will happen for the newly hatch dragon, as it will be attached to the children collection and after it's iterated it will age one year on the same year. The little hack we can do here is to go back to the hatch() and lay() methods and change the newly created objects to start from -1 year, because they will age immediately after that.

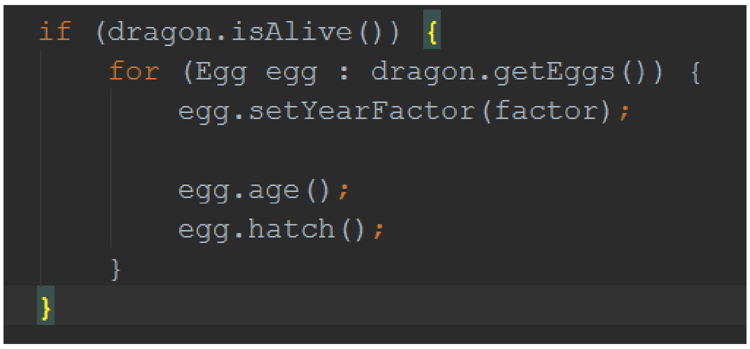


```java
public void lay() {
    if (this.age >= AGE_LAY_EGGS_START
            && this.age <= AGE_LAY_EGGS_END) {
        Egg egg = new Egg(-1, this);
        this.eggs.add(egg);
    }
}
```



```java
public void hatch() {
    if (this.age == AGE_HATCH) {
        // gets the corresponding integer
        int yearFactor = this.yearFactor.ordinal();
        for (int i = 0; i < yearFactor; i++) {
            Dragon baby = new Dragon(
                    this.parent.getName()
                            + "/"
                            + "Dragon_"
                            + DragonEra.dragonsCount, //name
                    -1 // age
            );

            this.parent.increaseOffspring(baby);
            DragonEra.dragonsCount++;
        }
    }
}
```

There's a little remark in the exercise. Dead dragons' eggs should not age, nor hatch. So iterating over the eggs in the recursive method should happen only if the dragon is not dead. We need to expose the value of isAlive field and use it in the "passAge()" method



```
if (dragon.isAlive()) {
    for (Egg egg : dragon.getEggs()) {
        egg.setYearFactor(factor);

        egg.age();
        egg.hatch();
    }
}
```

Task 10. Print the ancestry

To print the ancestry once it's ready we need to use, surprise, a recursion. You already know how to use it, so it's time to print them recursively. Make a method that prints each dragon's name, and calls recursively for its children. Each child should be printed on new line and 2 spaces indented more than its parent. Pass the indentation as an argument to the recursive function.