Inside of Hanton room


The notes!
The forbidden tome
of Hanton's knowledge


Oh, NO !
I… I cannot believe it!


Are they so unbelievably good?
No... this is just...
a list of timestamps

Pupon: Lec1 – 9:40 :
“It can not be done with only two variables,
It is mathematically proven.”

Nope: Just use this trivial one-liner
b = a + (a=b)*0;

Pupon: Lec3 – 9:10
“A try block always throws at most one exception,
and all exceptions are propagated to the caller.”

Nope:
In Java we have suppressed exceptions now!

```
class MyResource implements AutoCloseable {
  public void doSomething(){
    throw new IllegalArgumentException();
  }
  public void close(){
    throw new NullPointerException();
  }
  static void main(String[] a){
  try(var r = new MyResource()){ r.doSomething(); }
  }
}
```

This main throws an exception e of type
IllegalArgumentException, with
e.getSuppressed()[0] instanceof NullPointerException

Pupon: Lec4 – 9:23 :
“We can never be sure if assertions are enabled”

Nope: Just add this at the start of your main

```
try{ assert false; throw new Error("..."); }
catch(AssertionError){}
```

Pupon: Lec4 – 15:16 :
“Assertions need to be enabled from the command line with option –ea”

Nope: we can enable them programmatically via

```
ClassLoader.getSystemClassLoader()
    .setDefaultAssertionStatus(true);
```

This also shows the nature of class loading:

```
public class Main {
  public static void main(String[]a){
    //First the class Main is loaded
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
    assert false:"ignored, no error";
    System.out.println("Hello");//Prints no problem
    Other.method();       //Loading class ‘Other’
  }
}
class Other{
  static void method(){
    assert false:"Error here";
    System.out.println("World"); //Not printed
  }
}
```


This is so confusing
Wow, those are pretty advanced remarks. But... why the time stamps?
Is it possible that …

Pupon: Lec5 - 9:37 :
“Java does not have the comma operator of C/C++”

Nope: the comma operator is present in Java, but only in the third component of the ‘for’:

```
for(int x=0,y=1;x<10;x+=1,y=x+y+1){/*..*/}
```

However, using it is often considered bad style.

Pupon: Lec5 - 9:55 :
“In Java, Lists are serializable”

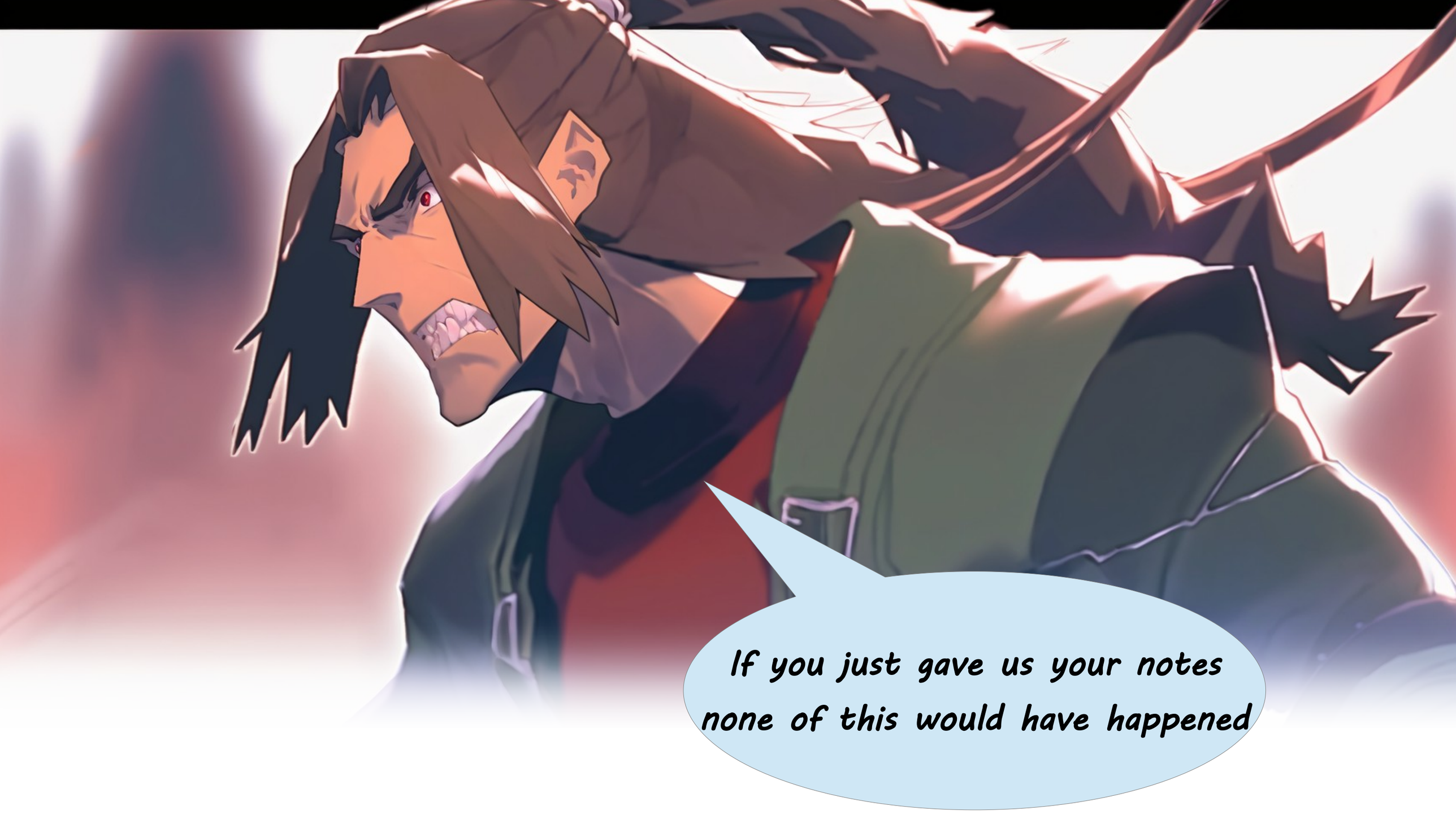
Nope: Not only we can define our own non serializable list, but even the common java.util.ArrayList$SubList is not serializable

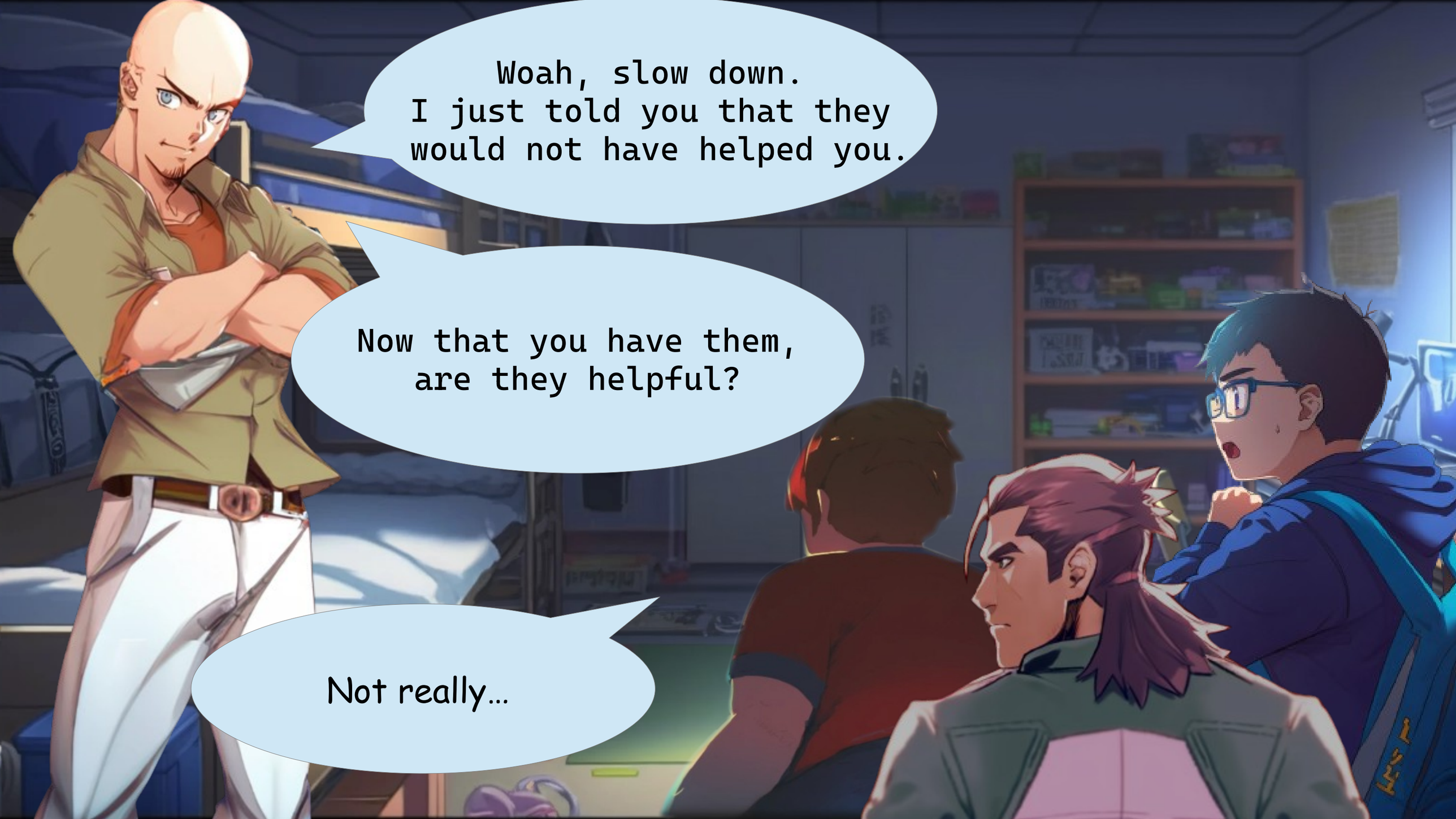
Those are obtained by calling the .subList(..) method.

However, many kinds of lists are serializable:
- ArrayList,
- List.of(),
- List.copyOf()
- Stream.toList()


The timestamps are all and only the moments when Pupon was wrong.
Oh, no, he's got us!
Hanton!


Why are you
in my dorm room?
Wait, we can explain …
What, you broke my window
to get in?


If you just gave us your notes
none of this would have happened


Woah, slow down.
I just told you that they would not have helped you.
Now that you have them, are they helpful?
Not really...


Told you so!


Now,
is there something
ELSE
I can do for you?


Hey Hanton … sorry, we should not have broken in your room.
Tonight
I’m going to be cold.
They will not fix the window until tomorrow!


Yea, we messed up big time.
But we are so desperate.
We can not understand
records and classes.
Are we going to just fail?


So, … why did you bunch not just ask for help? I could just have tutored you.
As I said: “is there something else I can do for you?”

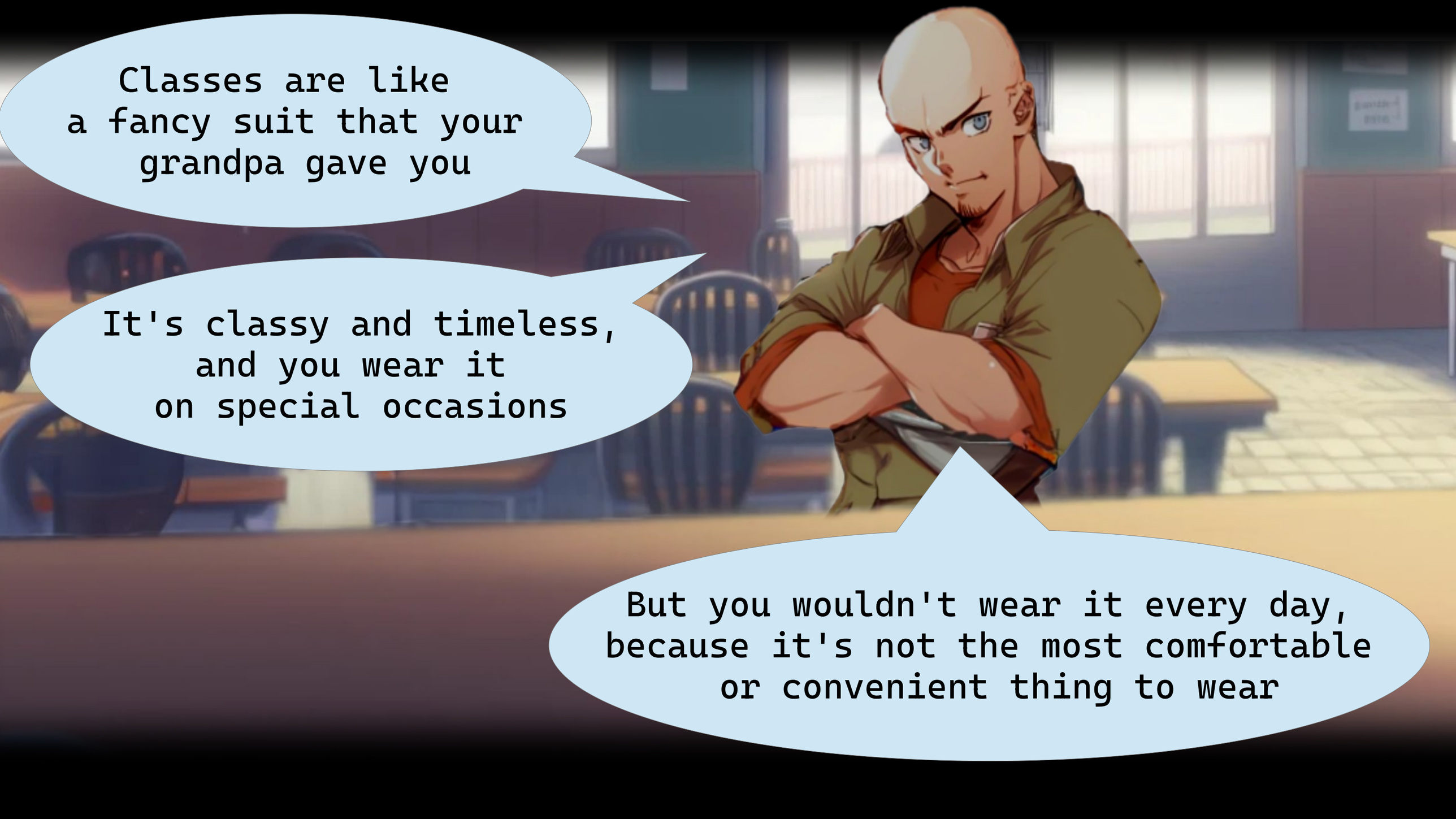

Are you... for real?
Will you tutor us even after we broke into your room?


Of course!
Your stupidity
does not reach me.
See you all tomorrow
in the cafeteria.
But,
you will have to pay me.
What about 245 dollars?

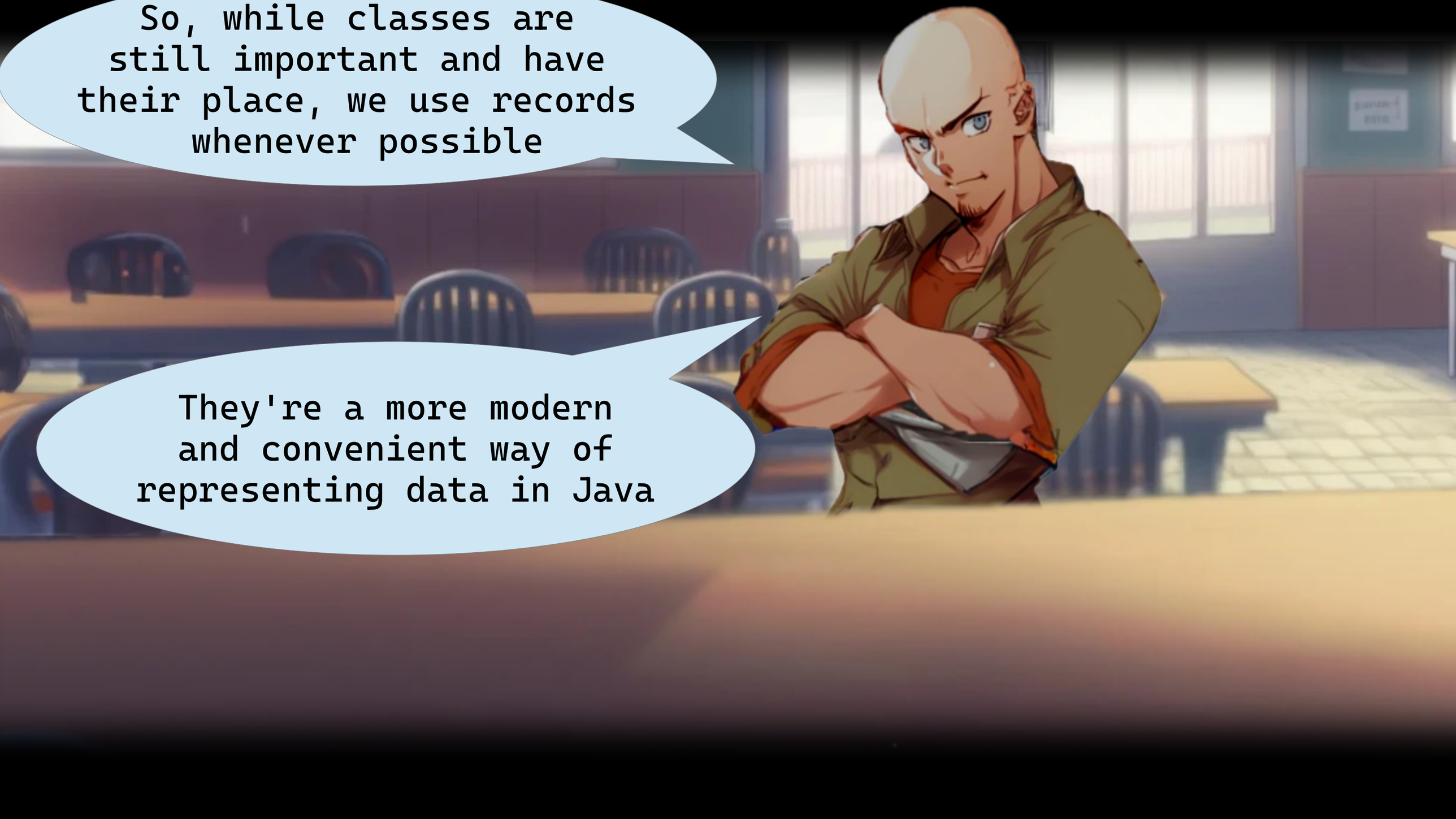

245 dollars? Are you insane?


If you asked earlier,
I would have done
it for free.
But fixing
a window in the dorm, …
it costs 245 dollars!


The day after, it is tutoring time!
So, classes and records
Classes have been in Java since the very beginning
Records are the new cool kid on the block. They went out of preview in Java 16

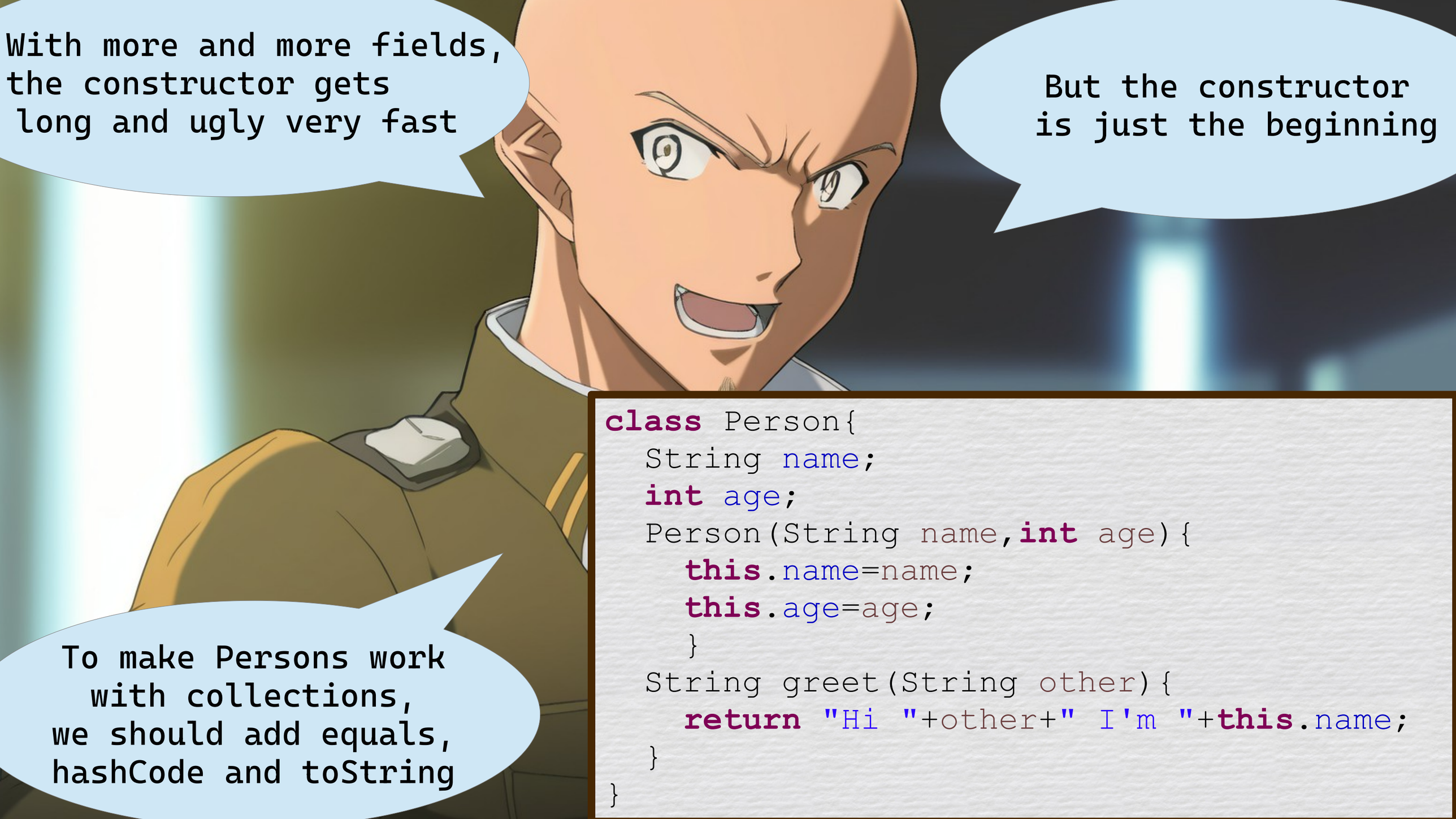

Classes are like
a fancy suit that your
grandpa gave you
It's classy and timeless,
and you wear it
on special occasions
But you wouldn't wear it every day,
because it's not the most comfortable
or convenient thing to wear


On the other hand, records are like the cool new outfit that all the kids in town are wearing
It's modern, stylish, and designed for everyday wear
It's comfortable and easy to move around in, and it still looks great

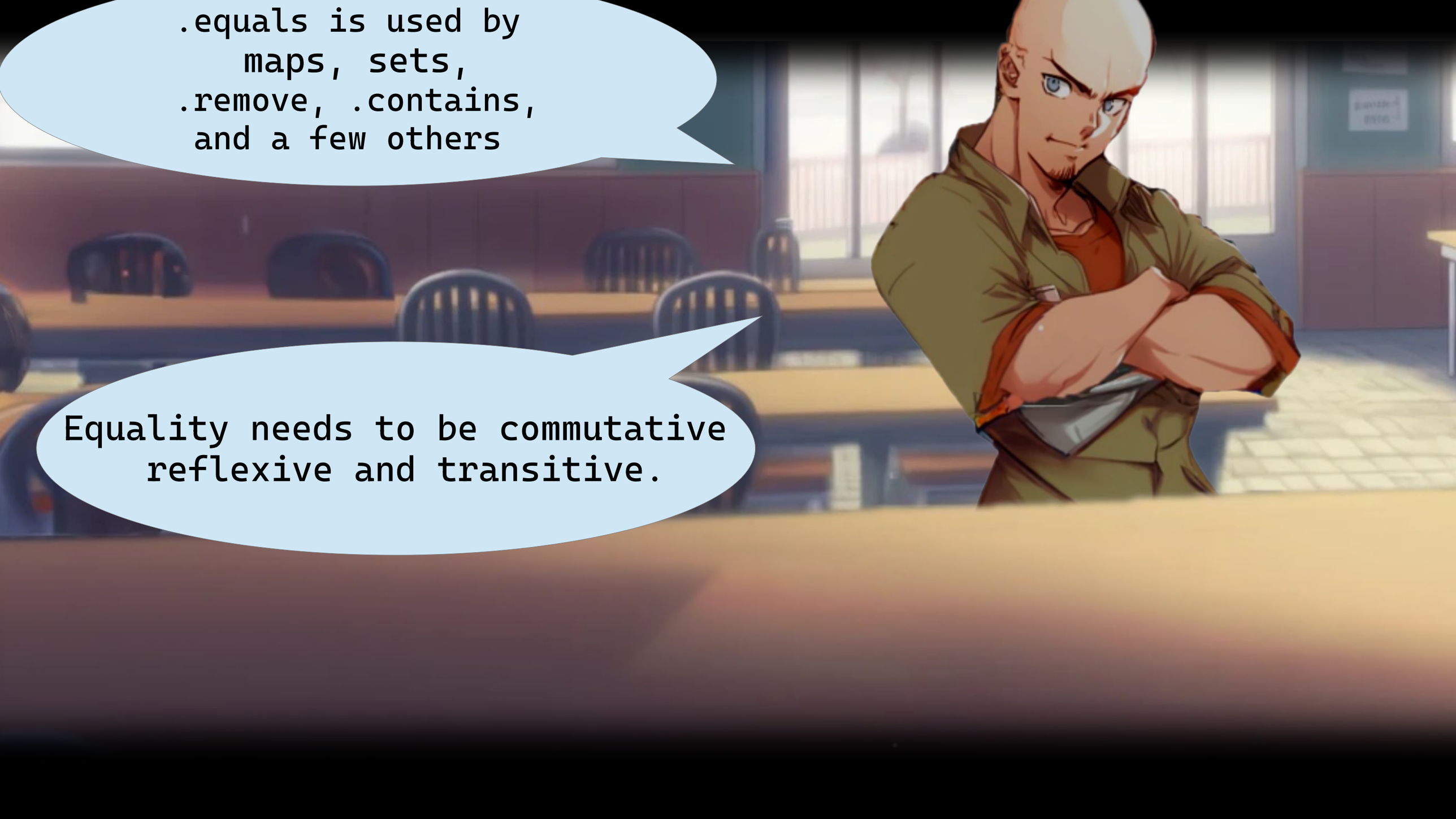

So, while classes are still important and have their place, we use records whenever possible
They're a more modern and convenient way of representing data in Java


Can you show us some code examples?
Some code where it is better to use a record?
And some code where we should dress up and use a class?


Sure. Consider a simple Person class with name and age
This code would be basically unusable
We should add a constructor
class Person{
String name;
int age;
String greet(String other){
return "Hi "+other+" I'm "+this.name;
}
}



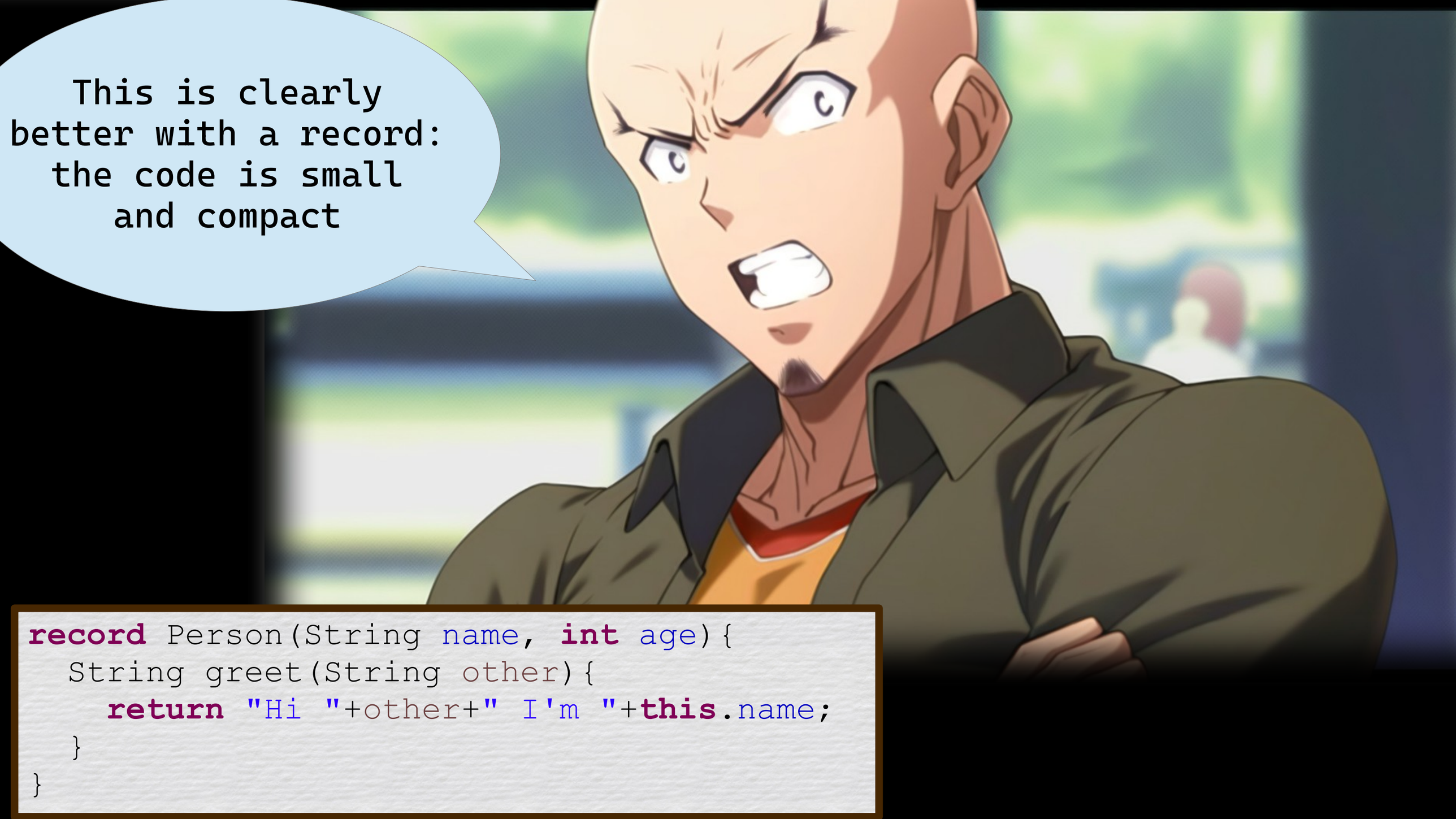

```java
class Person{
  String name;
  int age;
  Person(String name,int age){
    this.name=name;
    this.age=age;
    }
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```


.equals is used by
maps, sets,
.remove, .contains,
and a few others
Equality needs to be commutative
reflexive and transitive.

As you can see, even in such a simple case, the code is spiraling out of control!

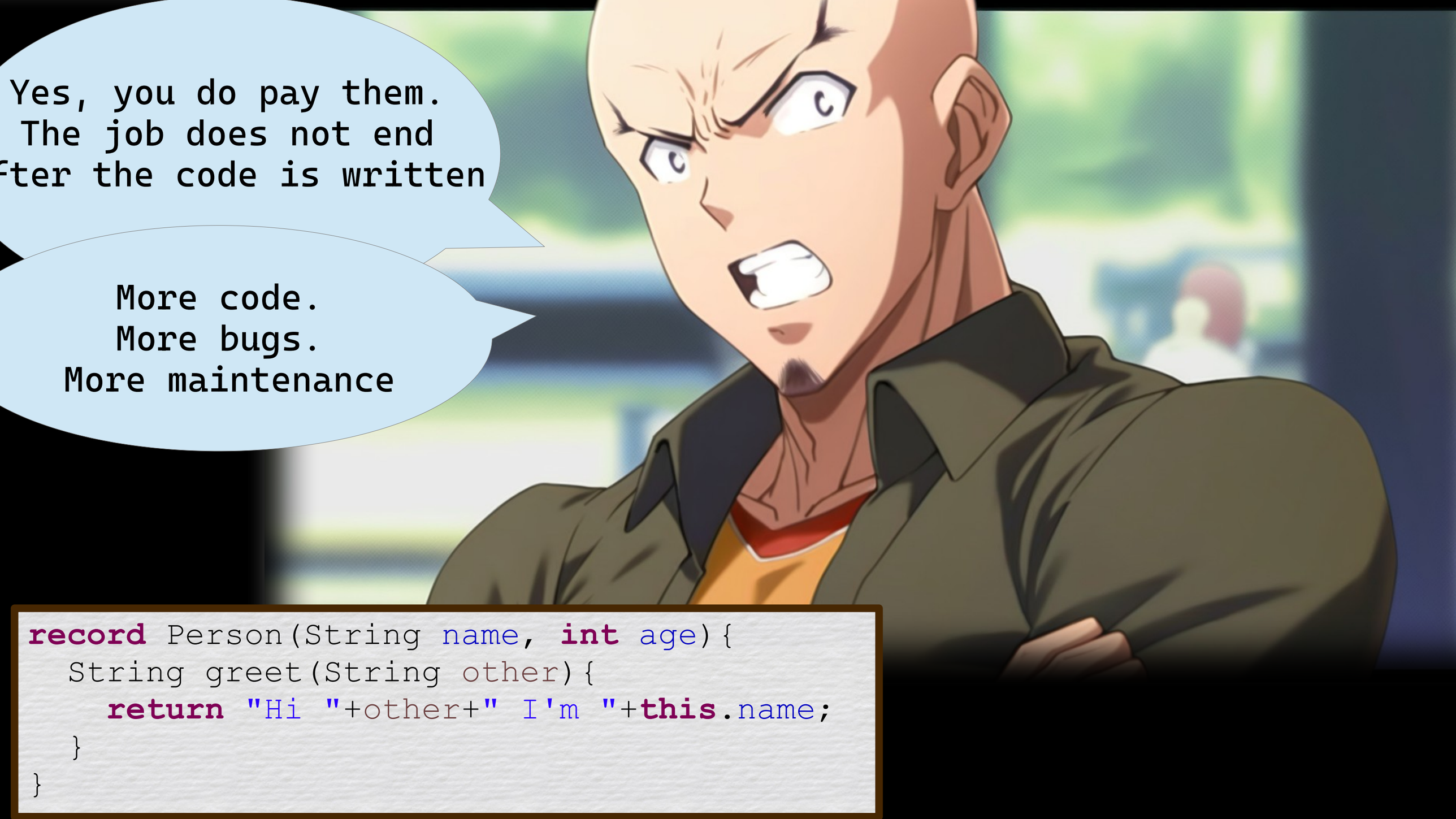
All of this before we even discuss private fields and getters!

```
class Person{
   String name;  int age;
   Person(String name,int age){
      this.name=name;  this.age=age;
      }
   public int hashCode(){
      return Objects.hash(age, name);
   }
   public boolean equals(Object obj){
      if(obj == null){ return false; }
      if(!(obj instanceof Person p)){ return false; }
      return age == p.age
         && getClass() == p.getClass()
         && Objects.equals(name, p.name);
   }
   String greet(String other){
      return "Hi "+other+" I'm "+this.name;
   }
}
```




```
record Person(String name, int age){
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```


Is it really that significant?
Yes, the class definitely has more code, ...
but it is not like we are paying for those extra lines

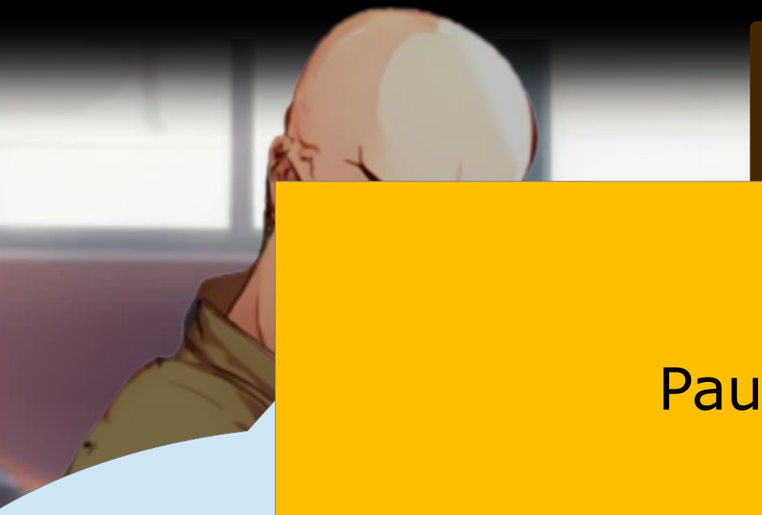



```
record Person(String name, int age){
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```

For example, here I deliberately added a redundant line

Can you spot it? The line can be removed to improve the code!

```
class Person{
   String name;  int age;
   Person(String name,int age){
      this.name=name;  this.age=age;
      }
   public int hashCode(){
      return Objects.hash(age, name);
   }
   public boolean equals(Object obj){
      if(obj == null){ return false; }
      if(!(obj instanceof Person p)){ return false; }
      return age == p.age
         && getClass() == p.getClass()
         && Objects.equals(name, p.name);
   }
   String greet(String other){
      return "Hi "+other+" I'm "+this.name;
   }
}
```



For
here I c
ad
redun

Can yo
The li
removed to
improve the code!

```
class Person{
  String name;  int age;

                                    false; }

  }
}
```

Pause the video. Can you solve this one?

Exactly one line can be removed
without changing the behavior of the code.

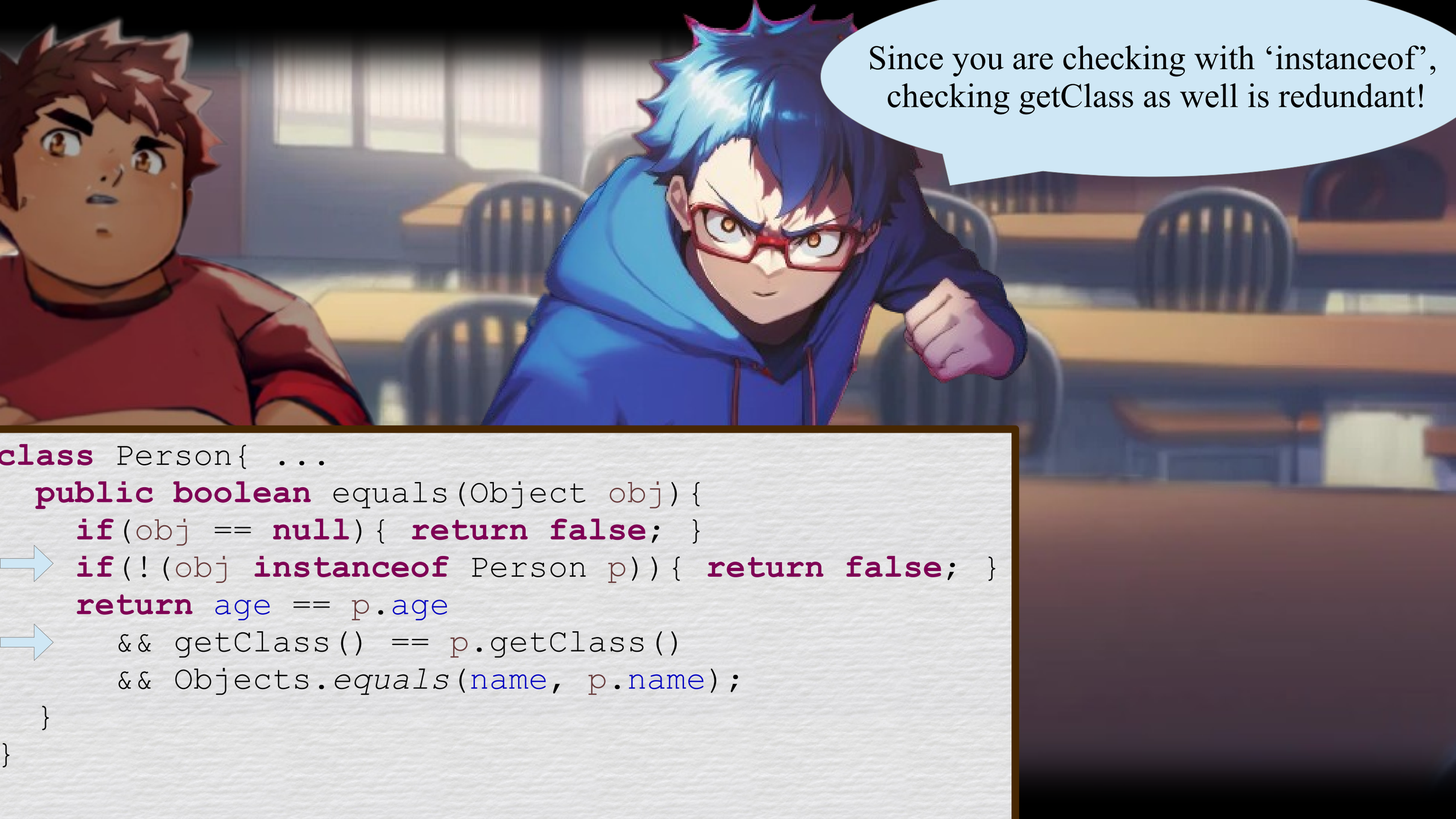
Can you find that line?

This message will disappear shortly.
You can pause the video after that.

For example, here I deliberately added a redundant line

Can you spot it? The line can be removed to improve the code!

```
class Person{
   String name;  int age;
   Person(String name,int age){
      this.name=name;  this.age=age;
      }
   public int hashCode(){
      return Objects.hash(age, name);
   }
   public boolean equals(Object obj){
      if(obj == null){ return false; }
      if(!(obj instanceof Person p)){ return false; }
      return age == p.age
         && getClass() == p.getClass()
         && Objects.equals(name, p.name);
   }
   String greet(String other){
      return "Hi "+other+" I'm "+this.name;
   }
}
```




```
class Person{ ...
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return false; }
    return age == p.age
      && getClass() == p.getClass()
      && Objects.equals(name, p.name);
  }
}
```

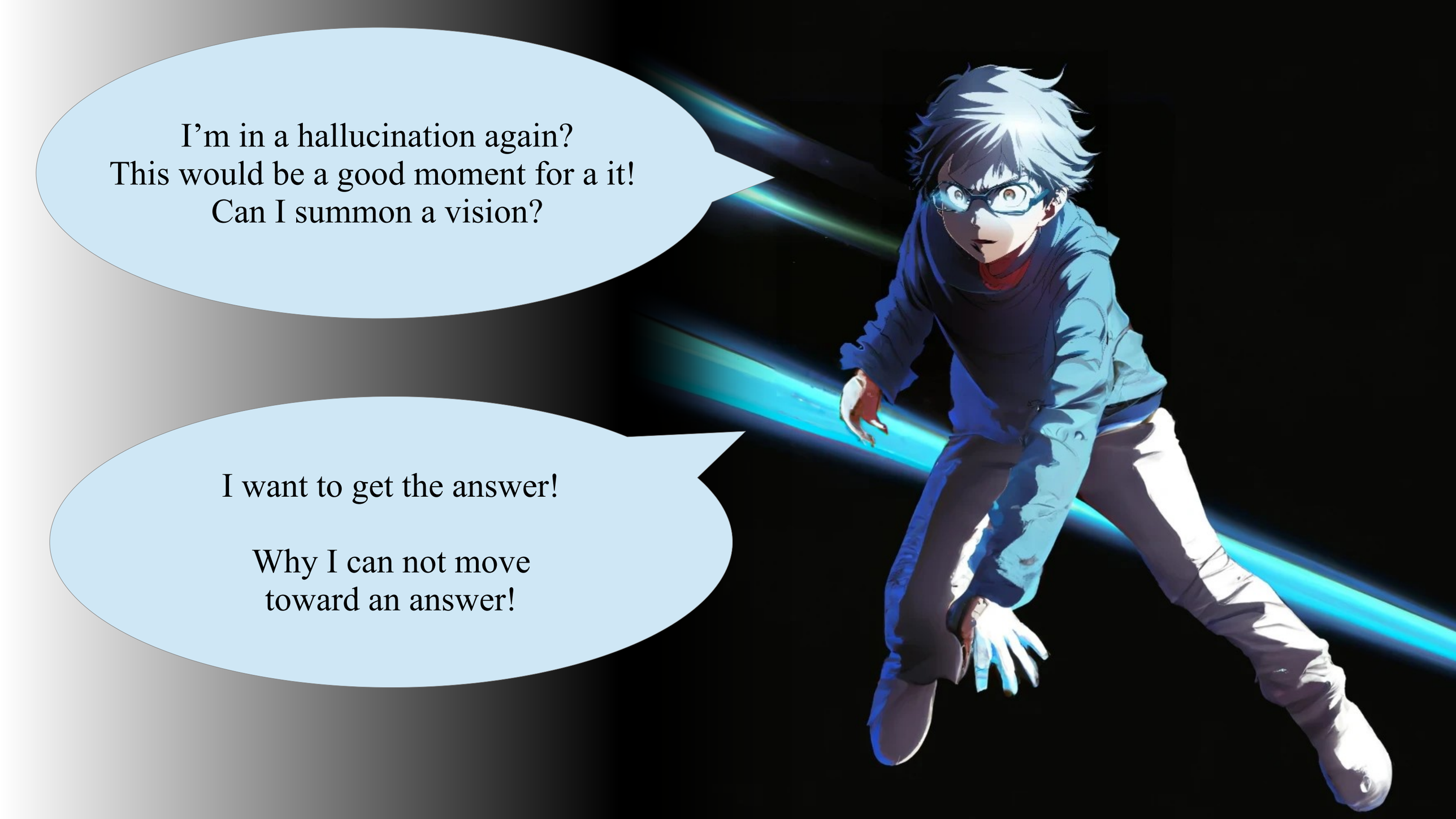

I knew you were going to answer like that, but I did not expect such a level of confidence
For a wrong answer, that is!

Wrong?
Removing that test would be … wrong?

I’m in a hallucination again?
This would be a good moment for a it!
Can I summon a vision?

I want to get the answer!

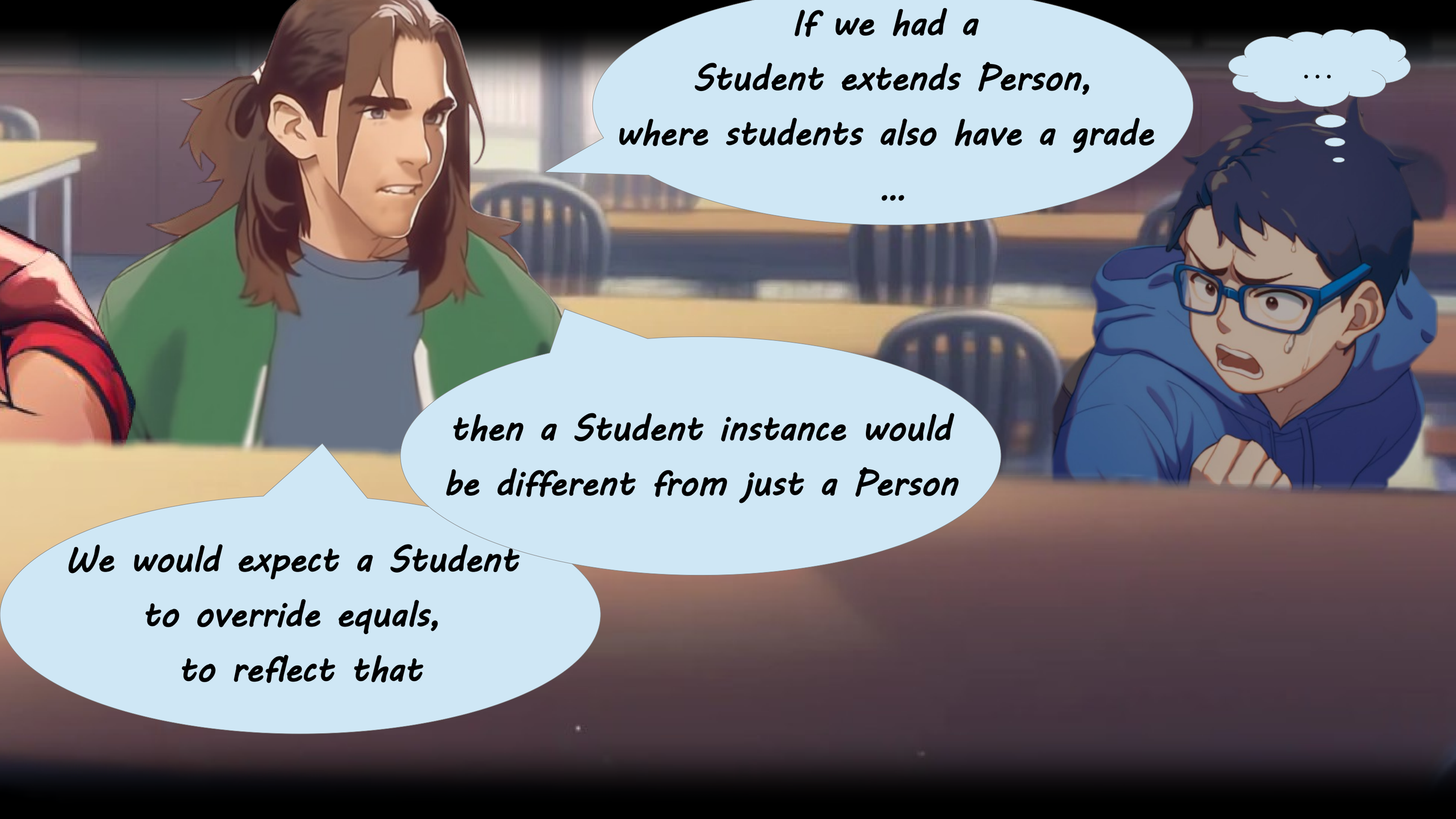
Why I can not move
toward an answer!




Nothing? I see nothing …
Why I can not reason my
way toward an answer?
Is there some limit
to my power?


Can you at least tell me …
why it is wrong to remove the class check?


You said equality
has to be reflexive, commutative
and transitive
Reflexivity seams to be ok, …
however, commutativity …


If we had a
Student extends Person,
where students also have a grade
...
....
then a Student instance would
be different from just a Person
We would expect a Student
to override equals,
to reflect that

Now a student is different from a person

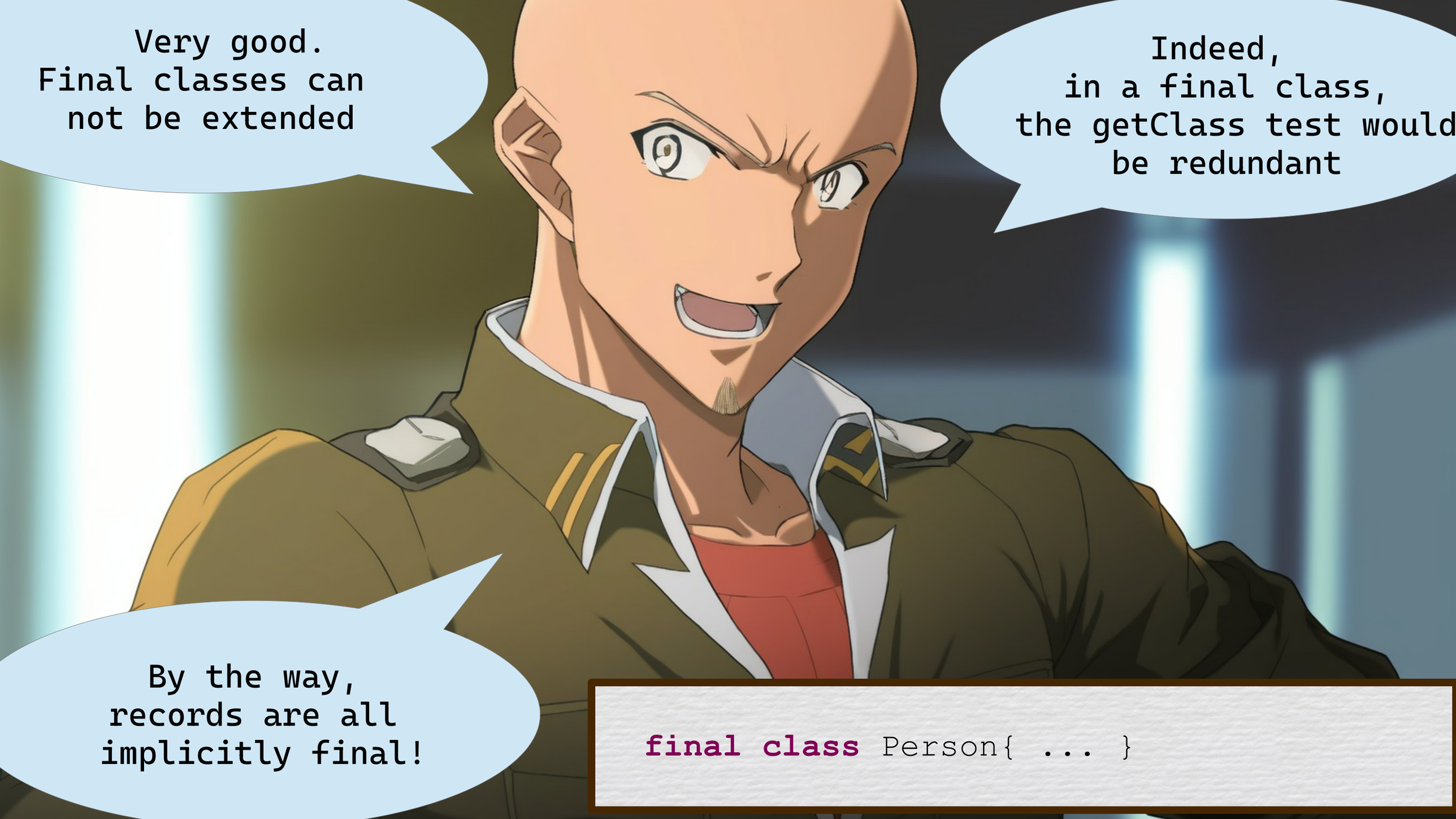
However ...

a person can still be equal to a student

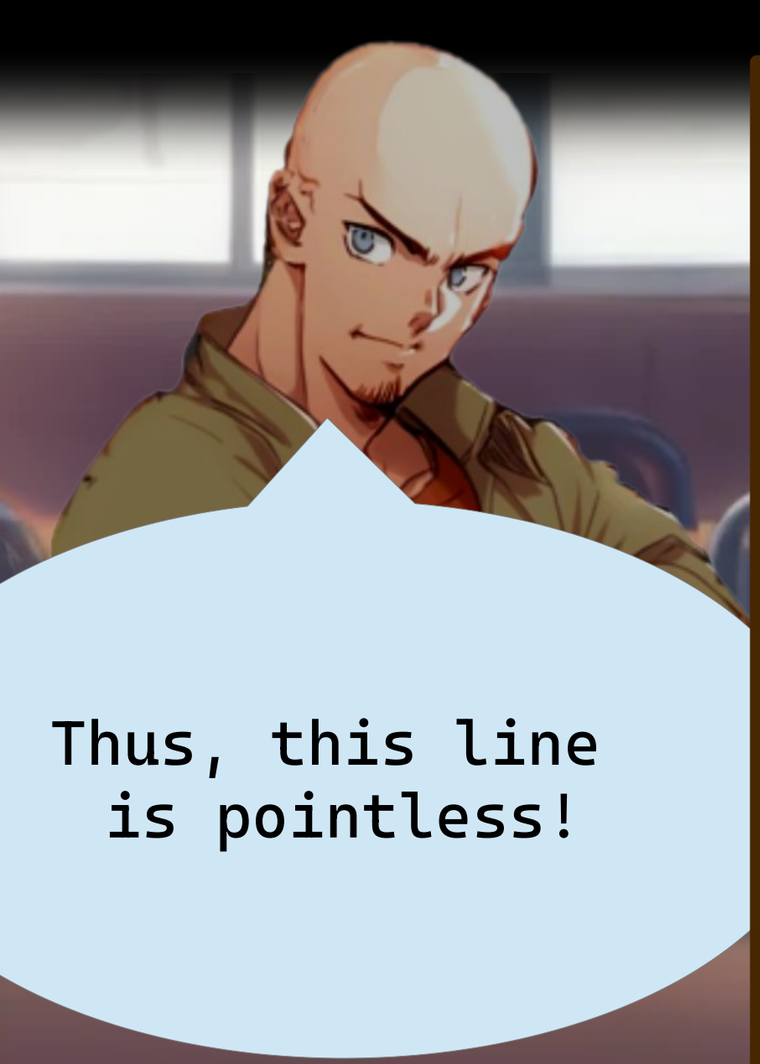
Since Student extends Person and Person.equals would not look at the grade field

```
class Person{ ...
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return false; }
    return age == p.age
      && Objects.equals(name, p.name);
  }
}
class Student extends Person{ ...
  int grade;
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Student s)){ return false; }
    return grade == s.grade
      && super.equals(this,obj);
  }
}
```


So,
either we keep that line,
or we make Person final

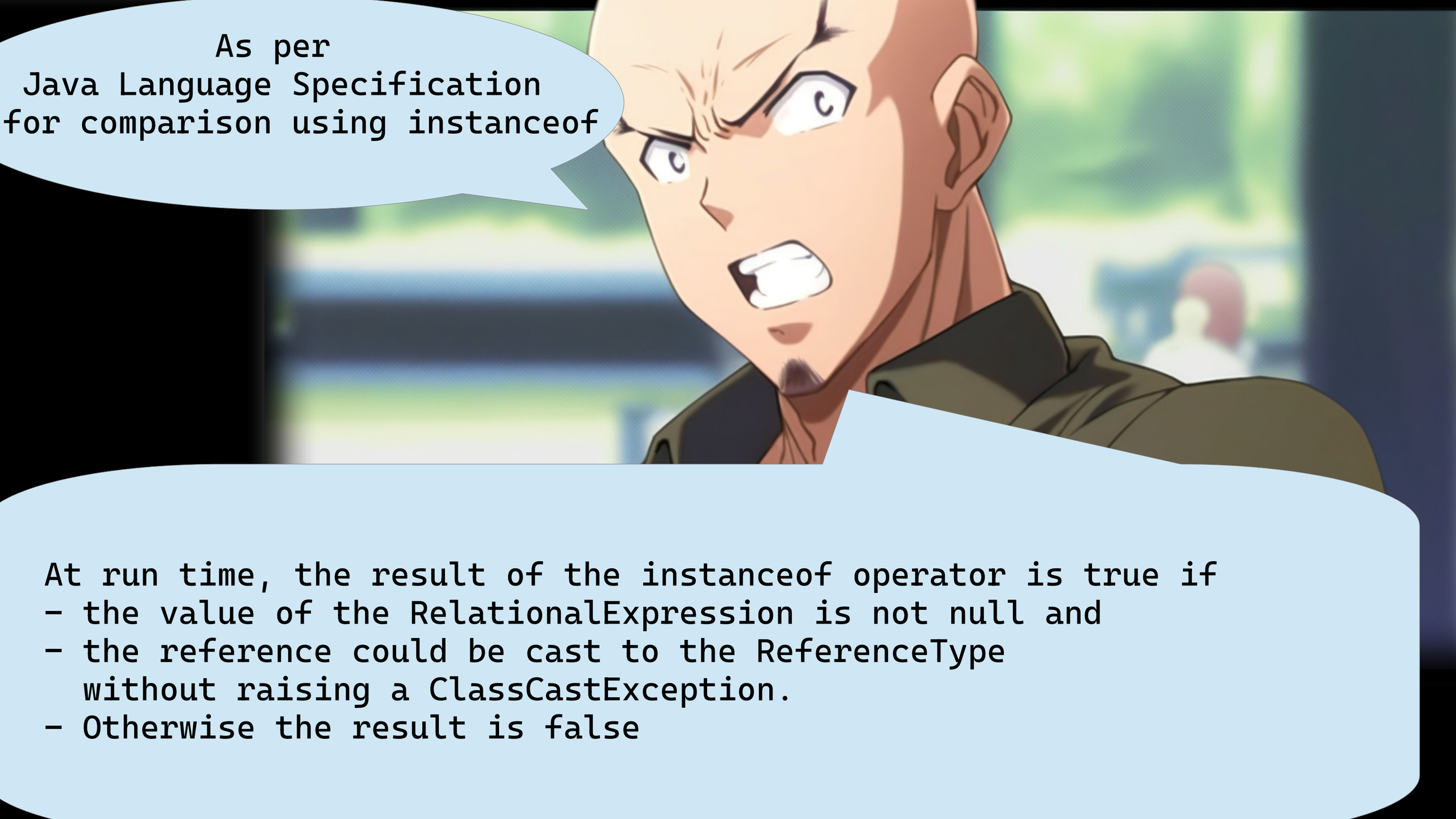

Very good.
Final classes can
not be extended
Indeed,
in a final class,
the getClass test would
be redundant
By the way,
records are all
implicitly final!
final class Person{ ... }


Now, time to reveal the redundant line:
It is the null check!
instanceof subsumes a null check!

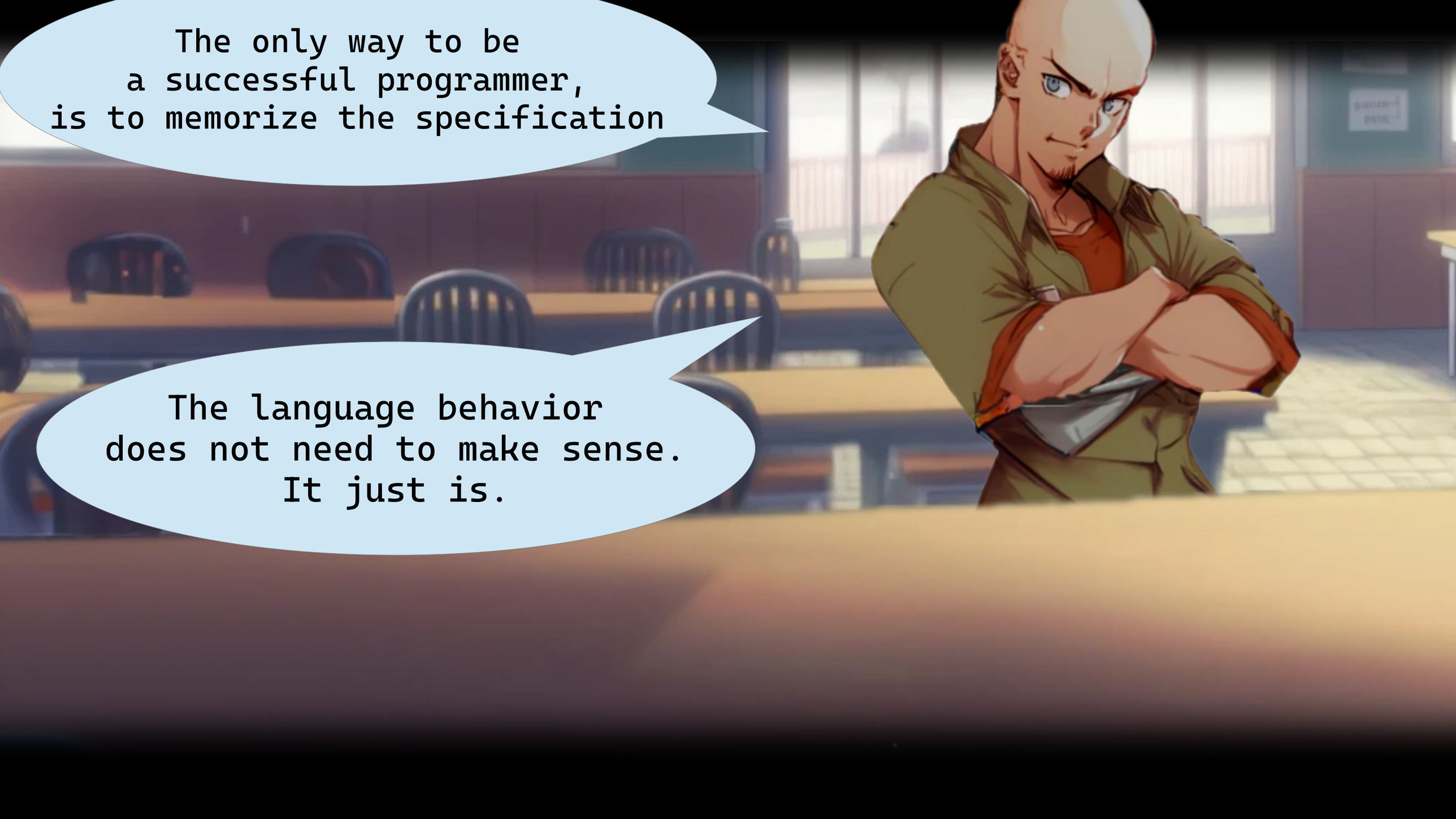


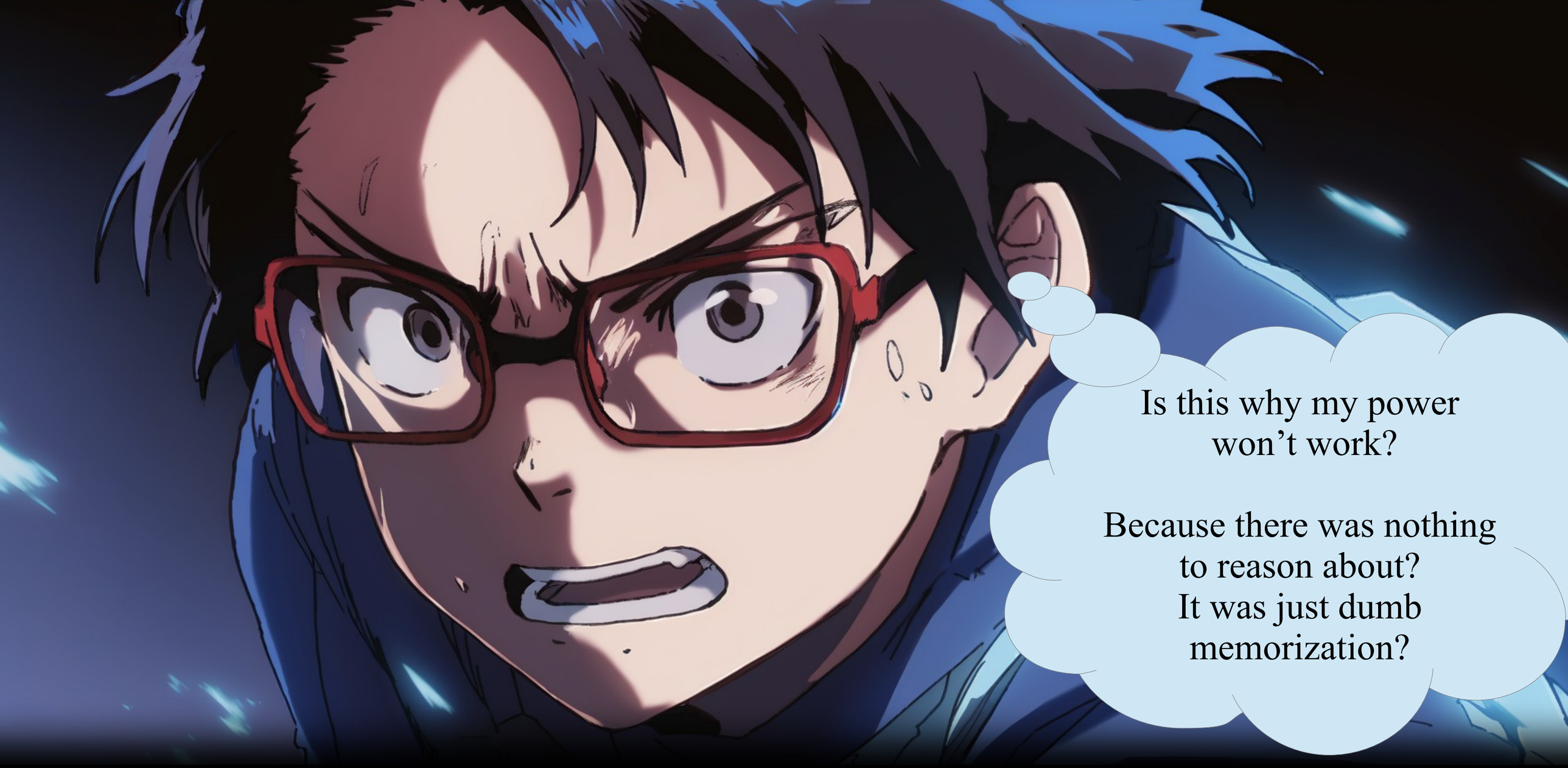

```java
class Person{
  String name;  int age;
  Person(String name,int age){
    this.name=name;  this.age=age;
    }
  public int hashCode(){
    return Objects.hash(age, name);
  }
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return false; }
    return age == p.age
      && getClass() == p.getClass()
      && Objects.equals(name, p.name);
  }
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```

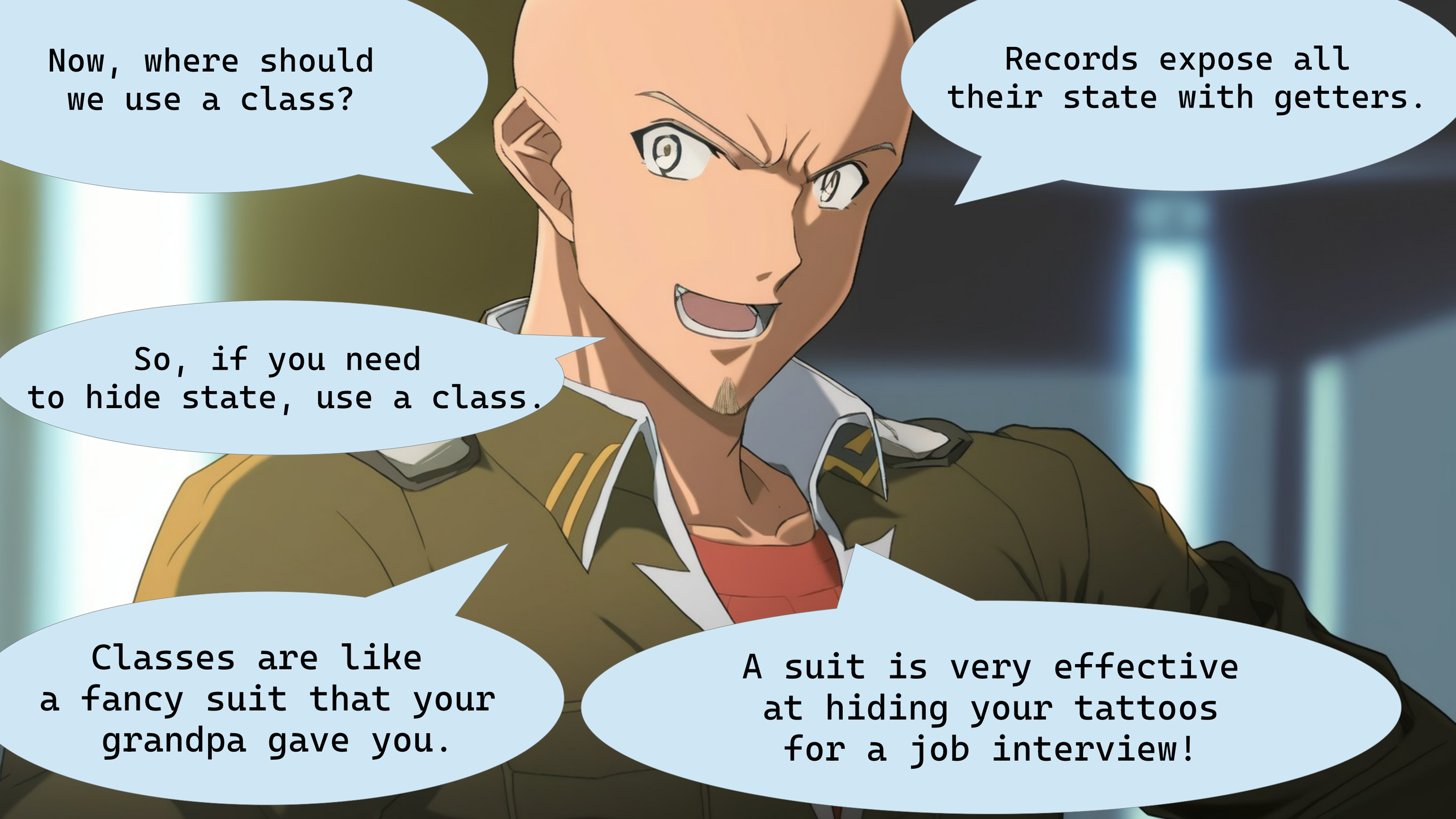

Yes, null can be assigned to a Person. It can be cast to Person too.
But, … assigning null to a Person would work, thus null is an instance of Person.
So, null must be an instance of Person!
It would be insane otherwise!

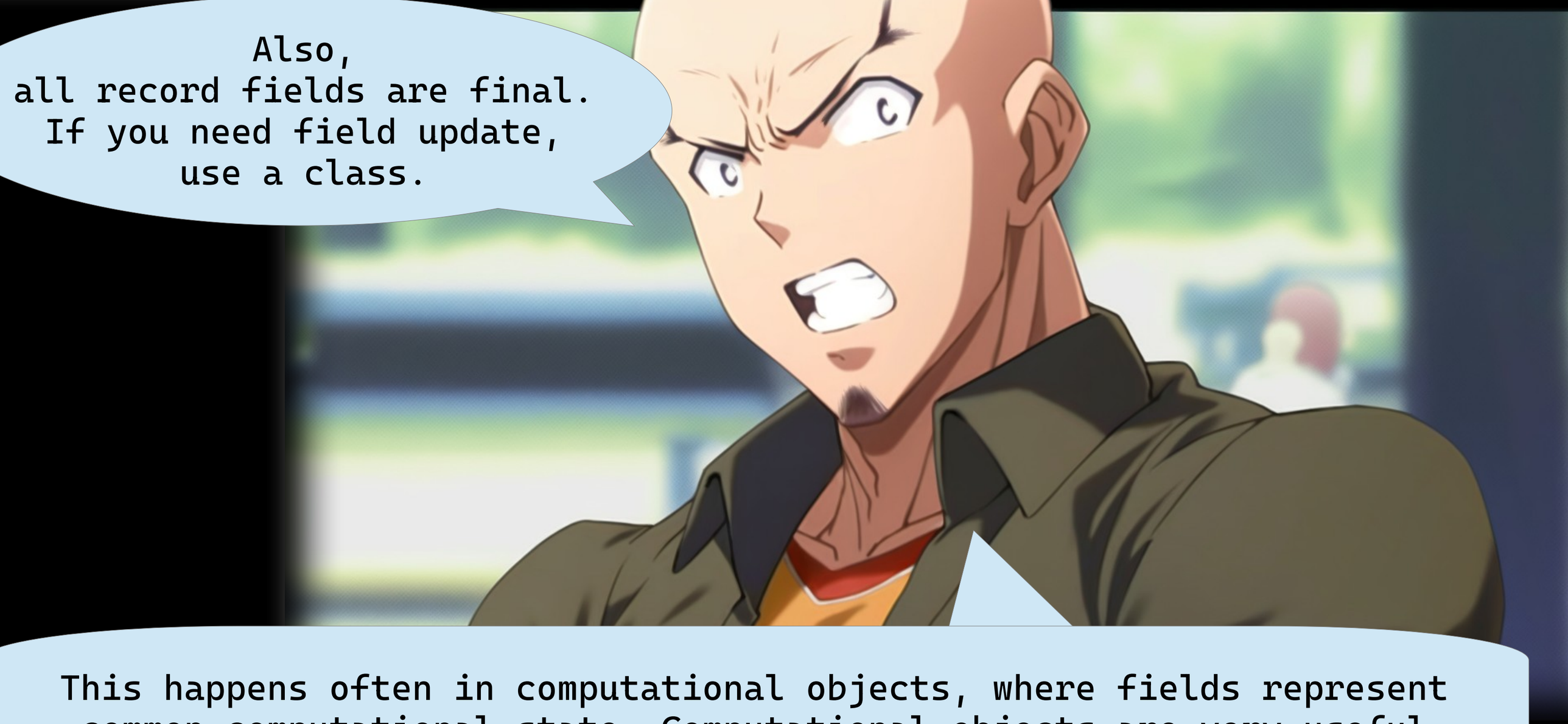


At run time, the result of the instanceof operator is true if
- the value of the RelationalExpression is not null and
- the reference could be cast to the ReferenceType
  without raising a ClassCastException.
- Otherwise the result is false

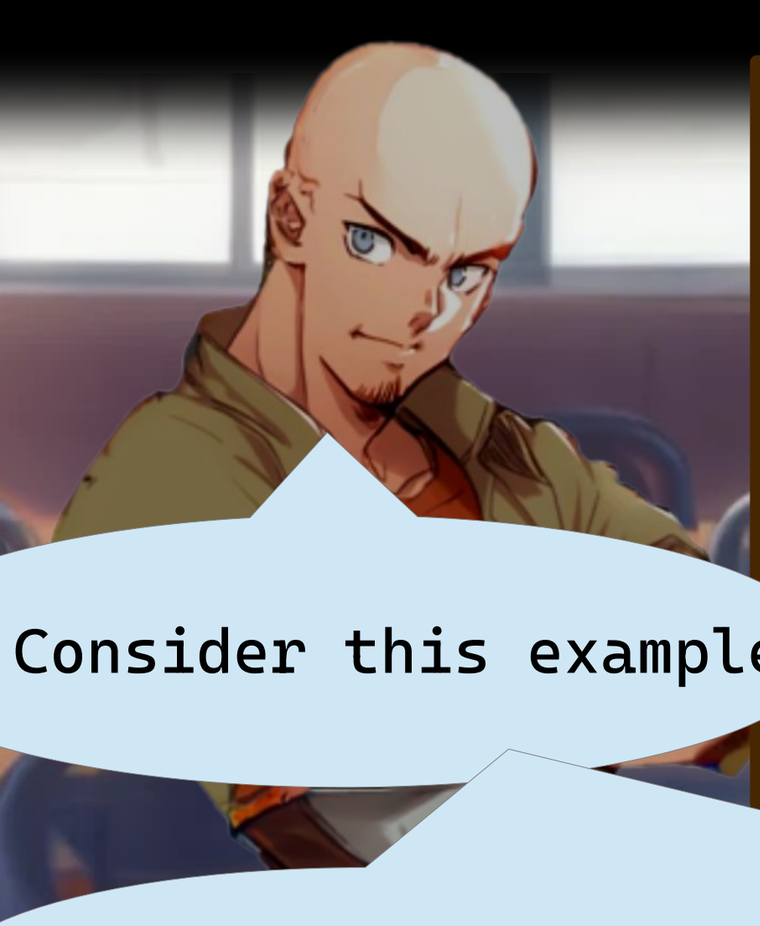

The only way to be
a successful programmer,
is to memorize the specification
The language behavior
does not need to make sense.
It just is.


Is this why my power won't work?
Because there was nothing to reason about?
It was just dumb memorization?


Now, where should we use a class?
Records expose all their state with getters.
So, if you need to hide state, use a class.
Classes are like a fancy suit that your grandpa gave you.
A suit is very effective at hiding your tattoos for a job interview!



This happens often in computational objects, where fields represent common computational state. Computational objects are very useful for splitting large methods into small readable chuncks.



Consider this example

```
//Given a String count how many digits and
//how many spaces in a single pass
final class StringInfo{
  public int digits = 0;
  public int spaces = 0;
  public StringInfo(String s){
    s.chars().forEach(c->{
    if(Character.isDigit(c)){ digits+=1; }
    if(Character.isWhitespace(c)){ spaces+=1; }
    });
  }
```

Here all fields have a default value; if they were local variables
we would not be able to update them inside the lambda.
Moreover, in this way the class StringInfo also works as
a tuple type, storing the two results we want.
If the logic to update those fields was more complex, we could
easily split it in multiple methods of StringInfo.