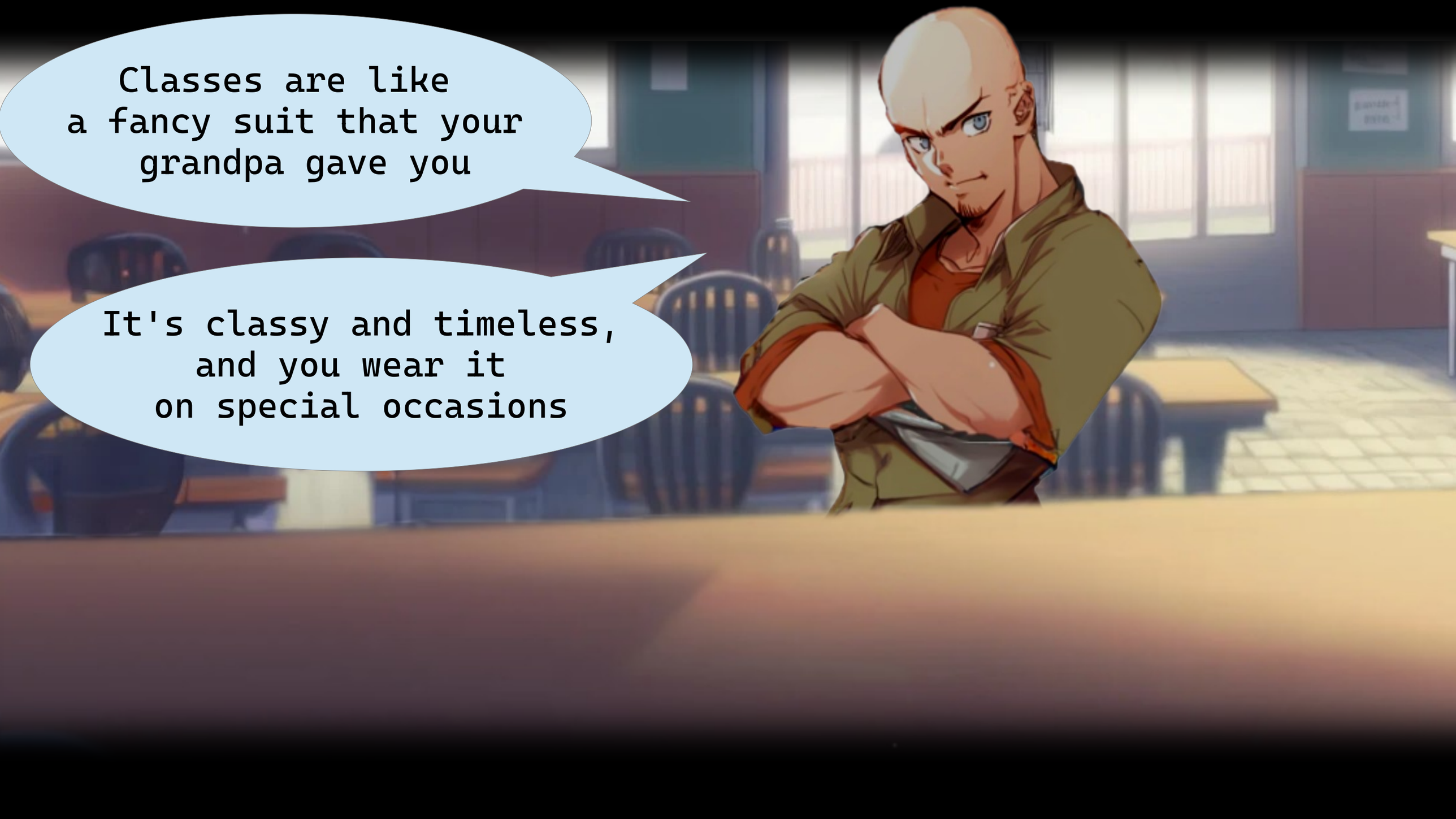
The day after, it is tutoring time!

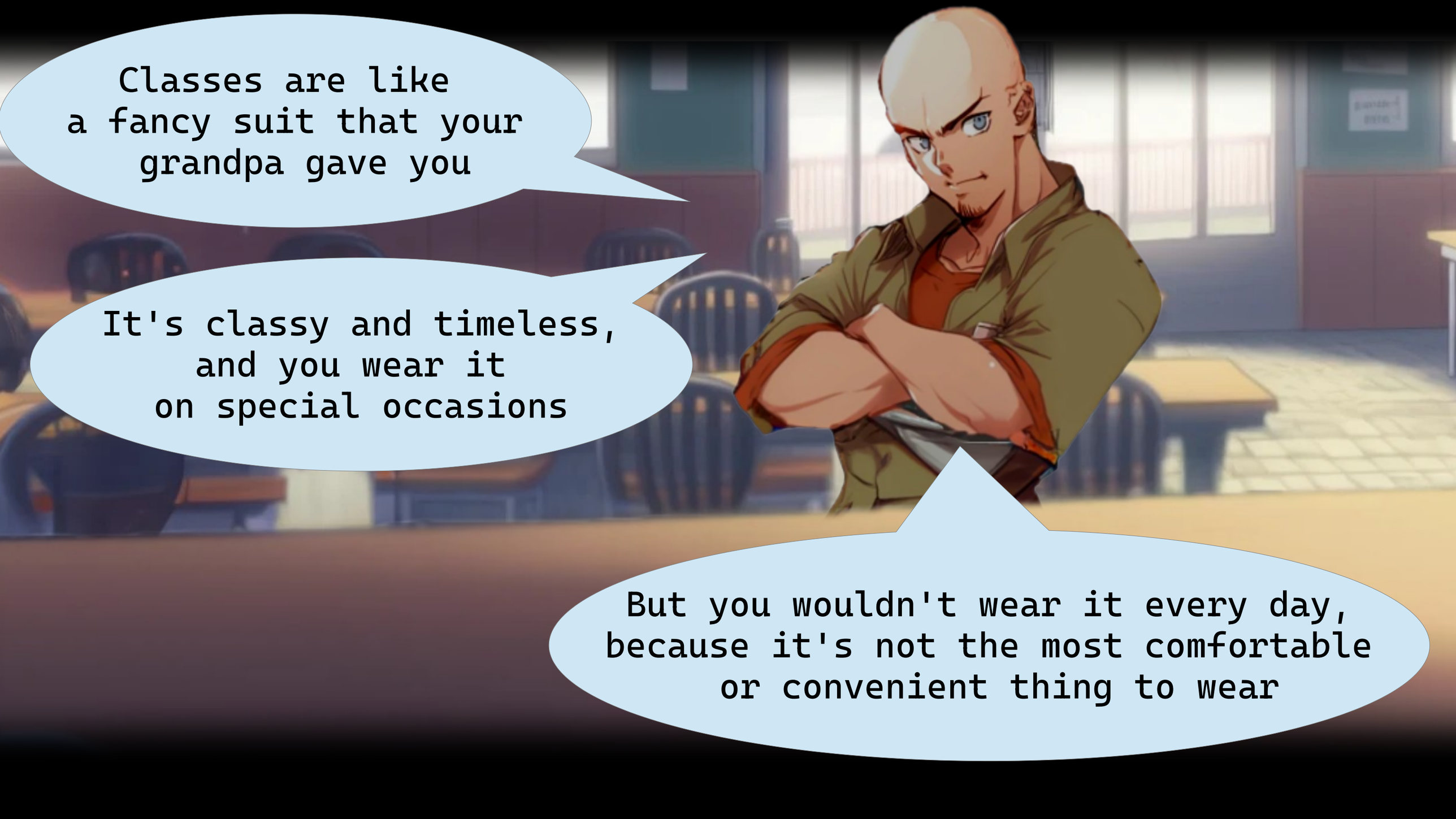

The day after, it is tutoring time!
So, classes and records


The day after, it is tutoring time!
So, classes and records
Classes have been in Java since the very beginning


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

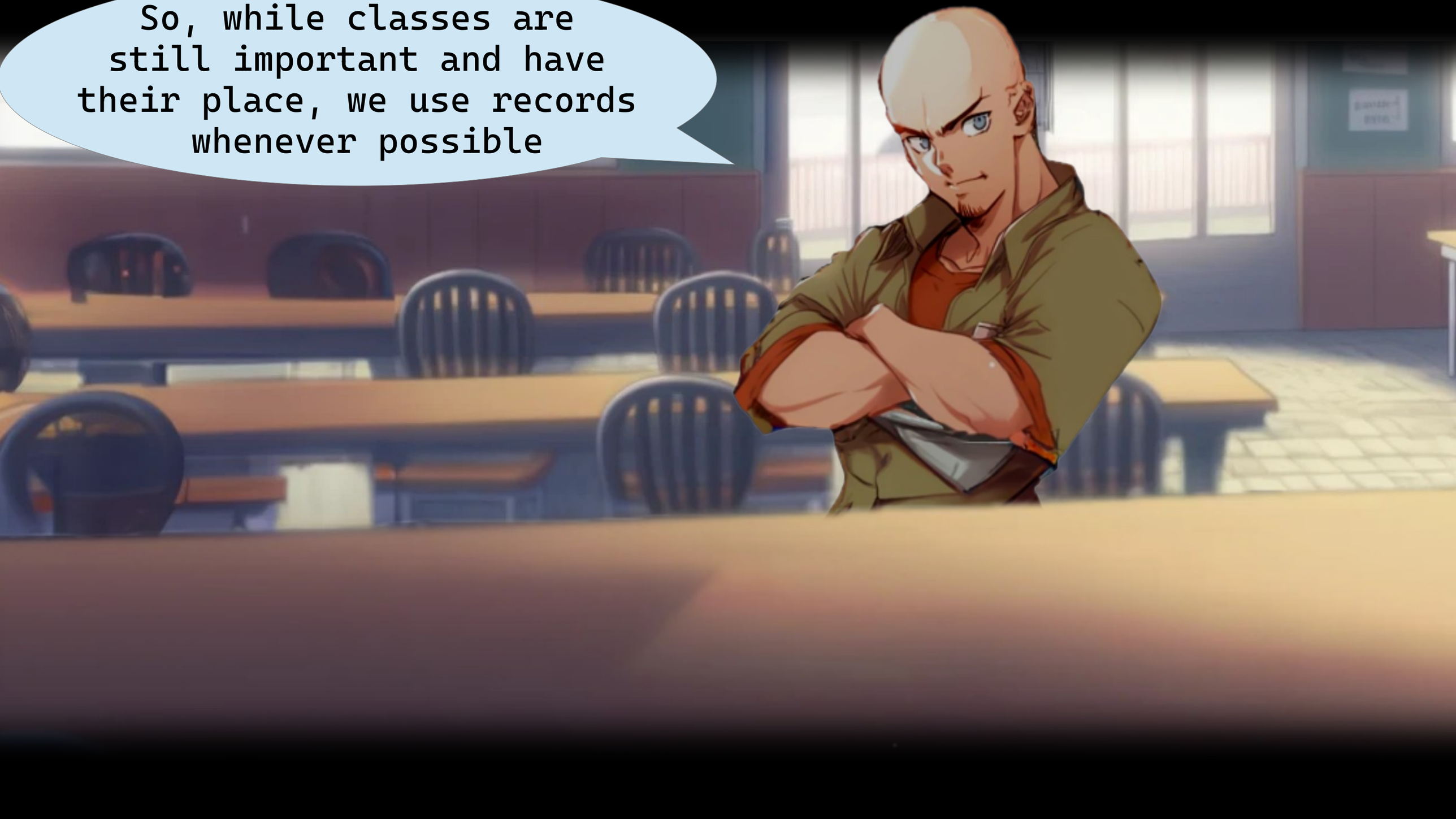

Classes are like a fancy suit that your grandpa gave you
It's classy and timeless, and you wear it on special occasions

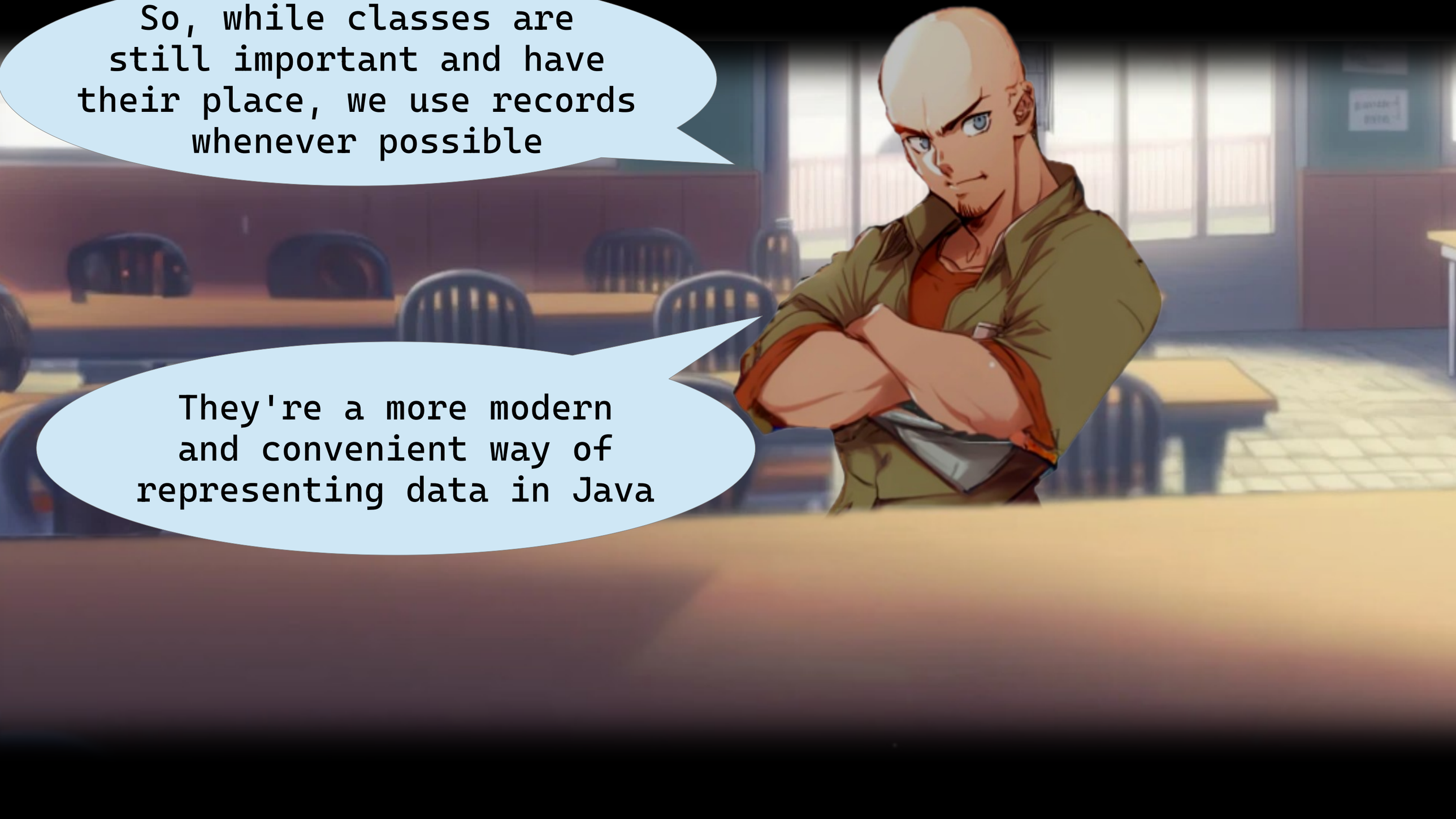

Classes are like a fancy suit that your grandpa gave you
It's classy and timeless, and you wear it on special occasions
But you wouldn't wear it every day, because it's not the most comfortable or convenient thing to wear


On the other hand, records are like the cool new outfit that all the kids in town are wearing


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


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


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?


Can you show us some code examples?
Some code where it is better to use a record?

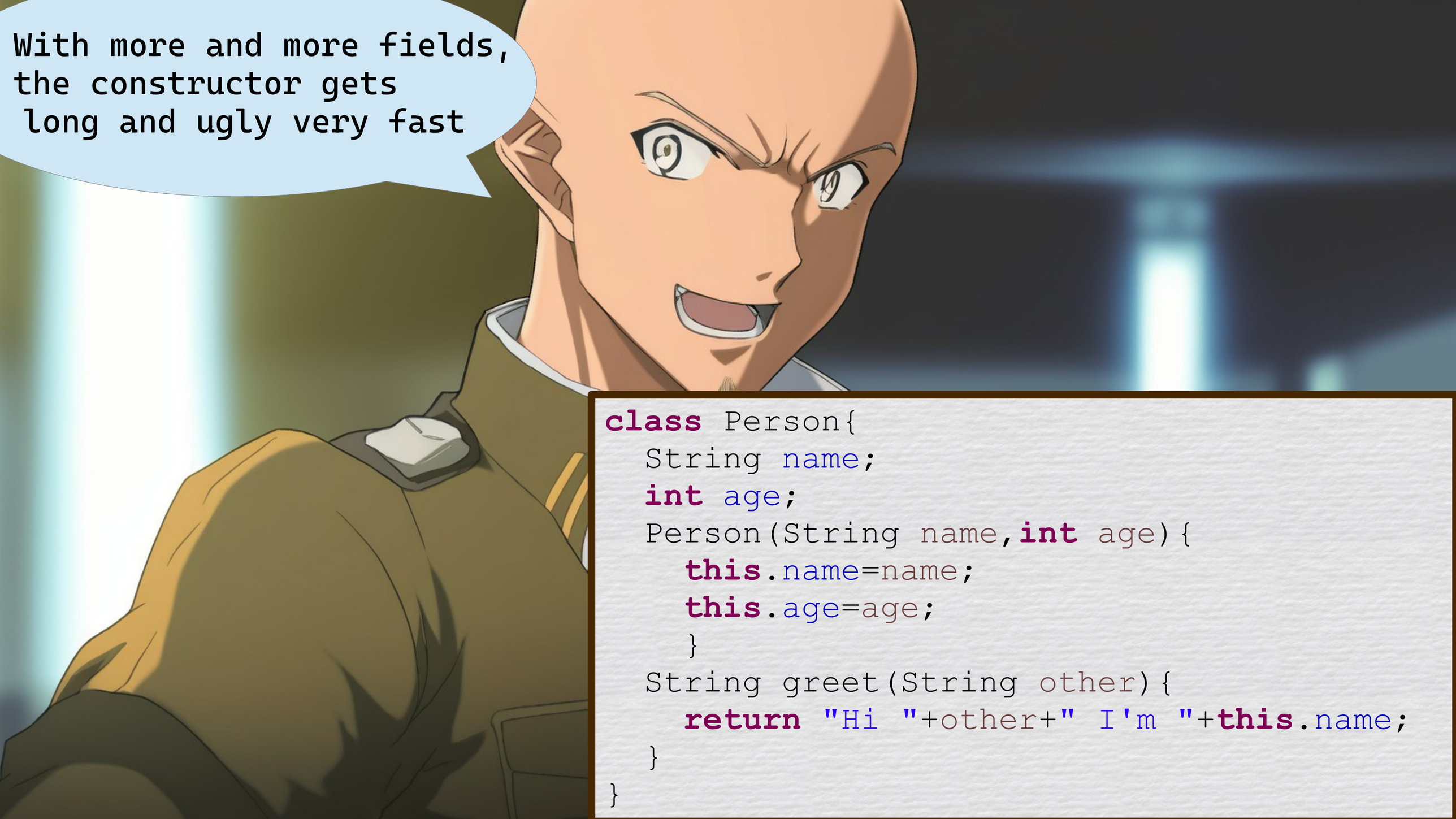

Can you show us some code examples?
Some code where it is better to use a record?
And some code where we should dress up and use a class?


Sure. Consider a simple Person class with name and age


Sure. Consider a simple Person class with name and age
class Person{
String name;
int age;
String greet(String other){
return "Hi "+other+" I'm "+this.name;
}
}


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


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;
}
}




```
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;
  }
}
```


```
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;
  }
}
```

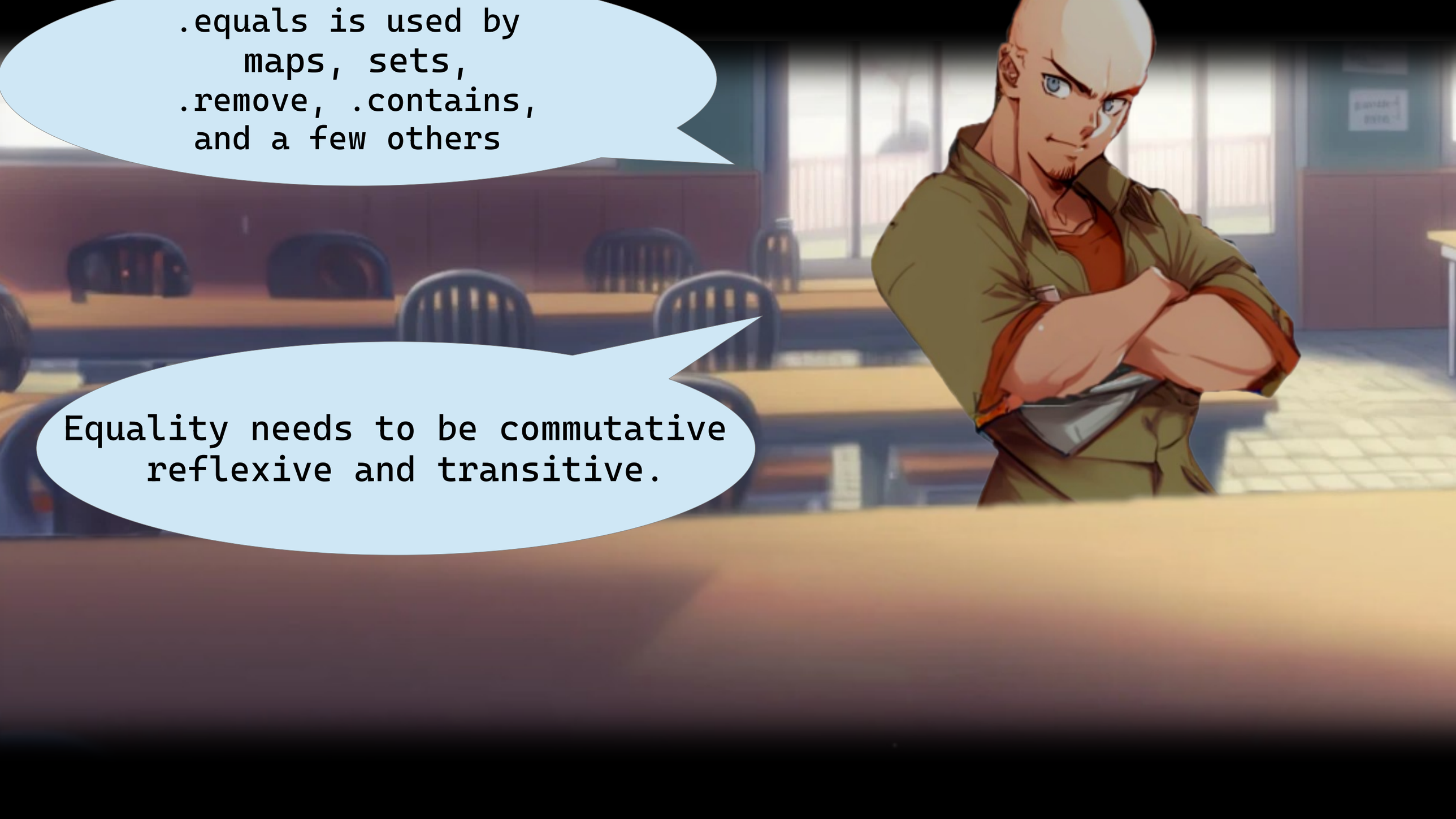
With more and more fields, the constructor gets long and ugly very fast

But the constructor is just the beginning

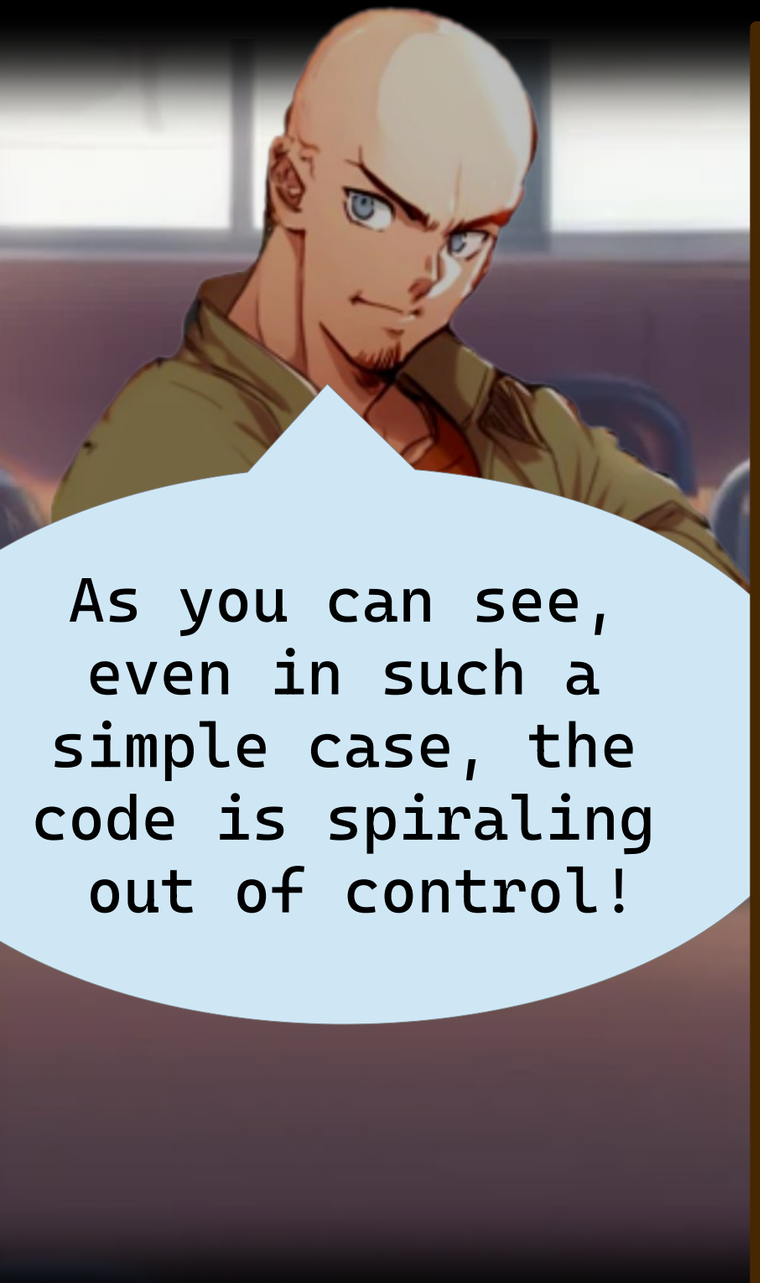
```
class Person{
  String name;
  int age;
  Person(String name,int age){
    this.name=name;
    this.age=age;
    }
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```

To make Persons work with collections, we should add equals, hashCode and toString

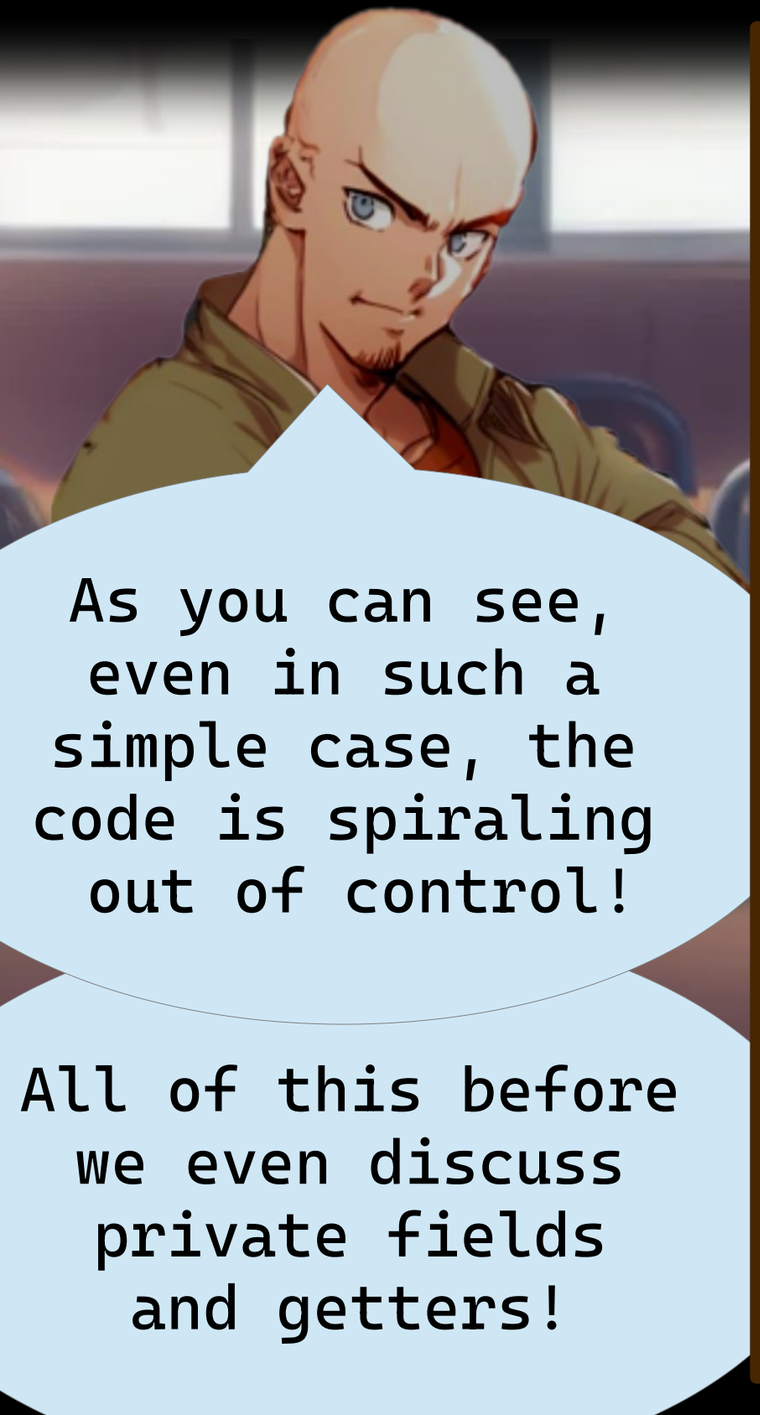

.equals is used by
maps, sets,
.remove, .contains,
and a few others


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

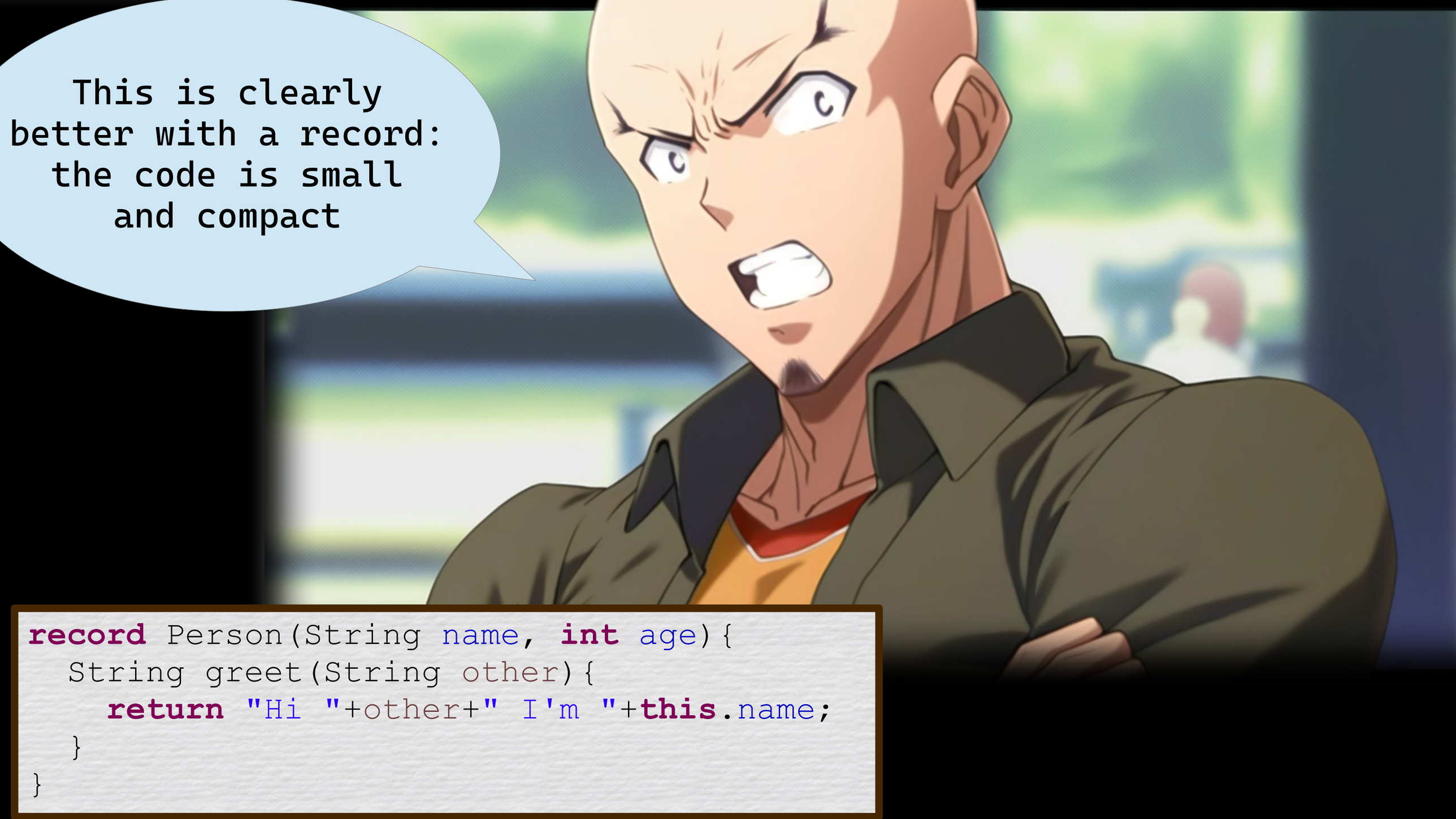
```java
class Person{
  String name;  int age;
  Person(String name,int age){
    this.name=name;  this.age=age;
    }
  public int hashCode(){
    return Objects.hash(age, name);
  }
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return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




```
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{
  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;
  }
}
```

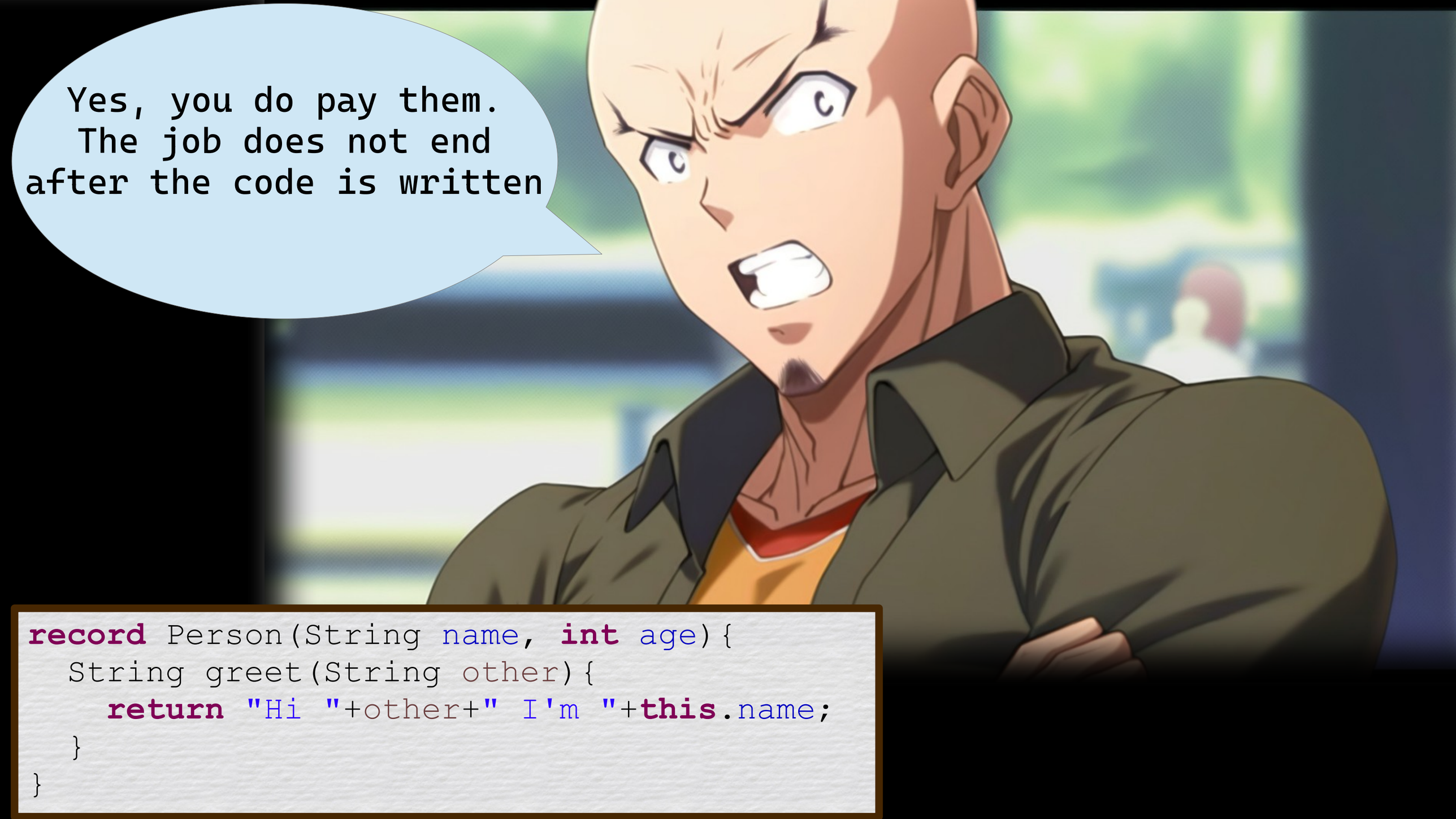


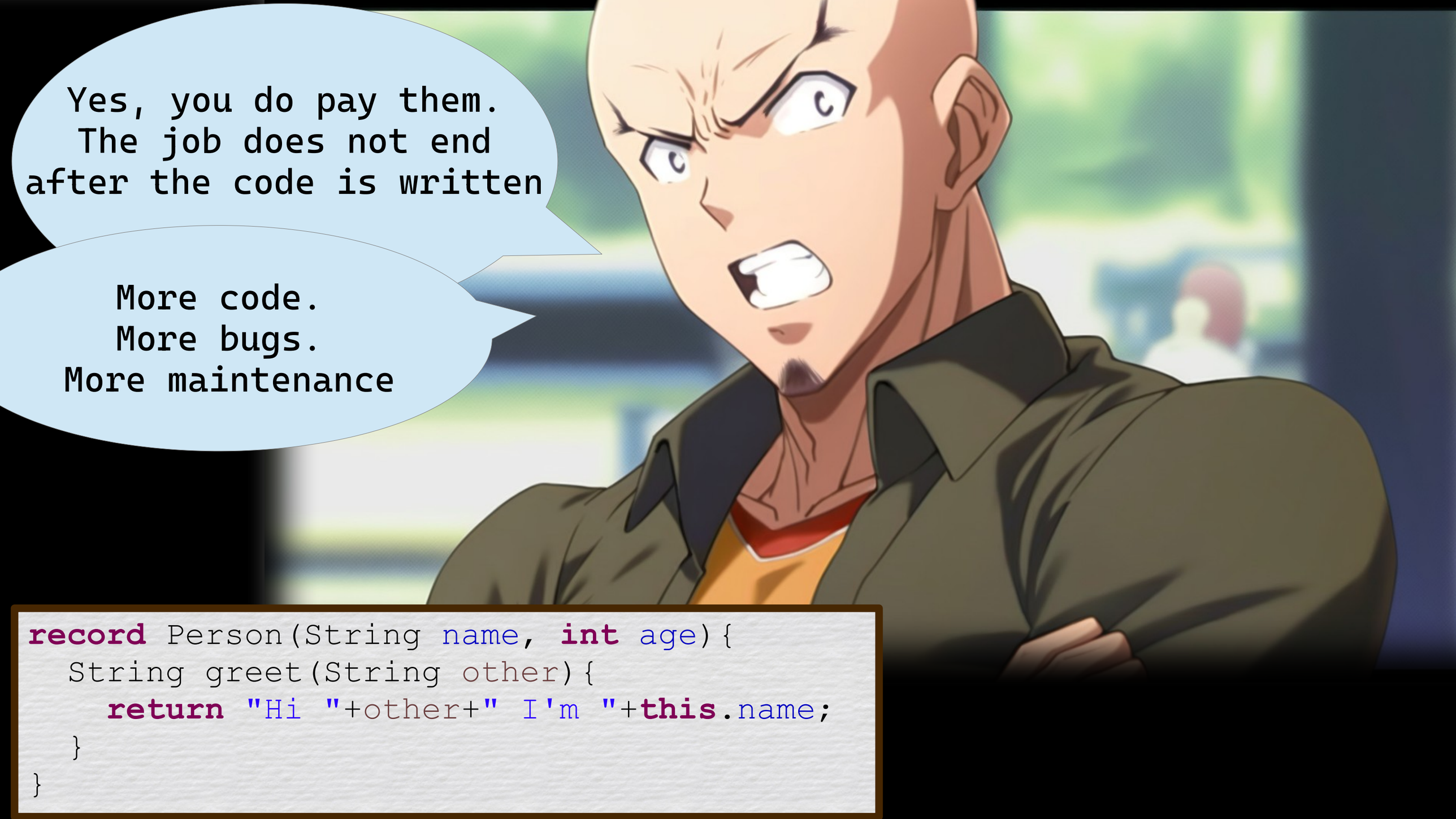

```
record Person(String name, int age){
  String greet(String other){
    return "Hi "+other+" I'm "+this.name;
  }
}
```


Is it really that significant? Yes, the class definitely has more code, ...

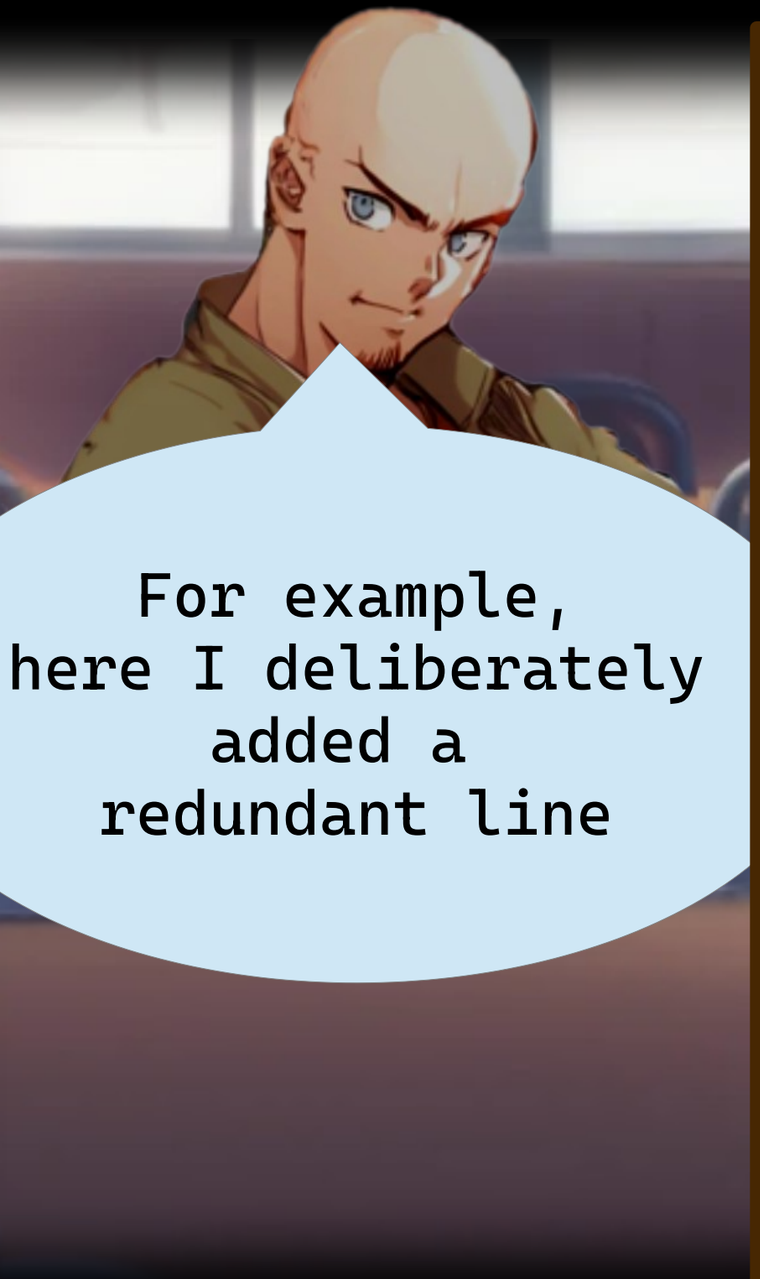

Is it really that significant? Yes, the class definitely has more code, ...
but it is not like we are paying for those extra lines

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

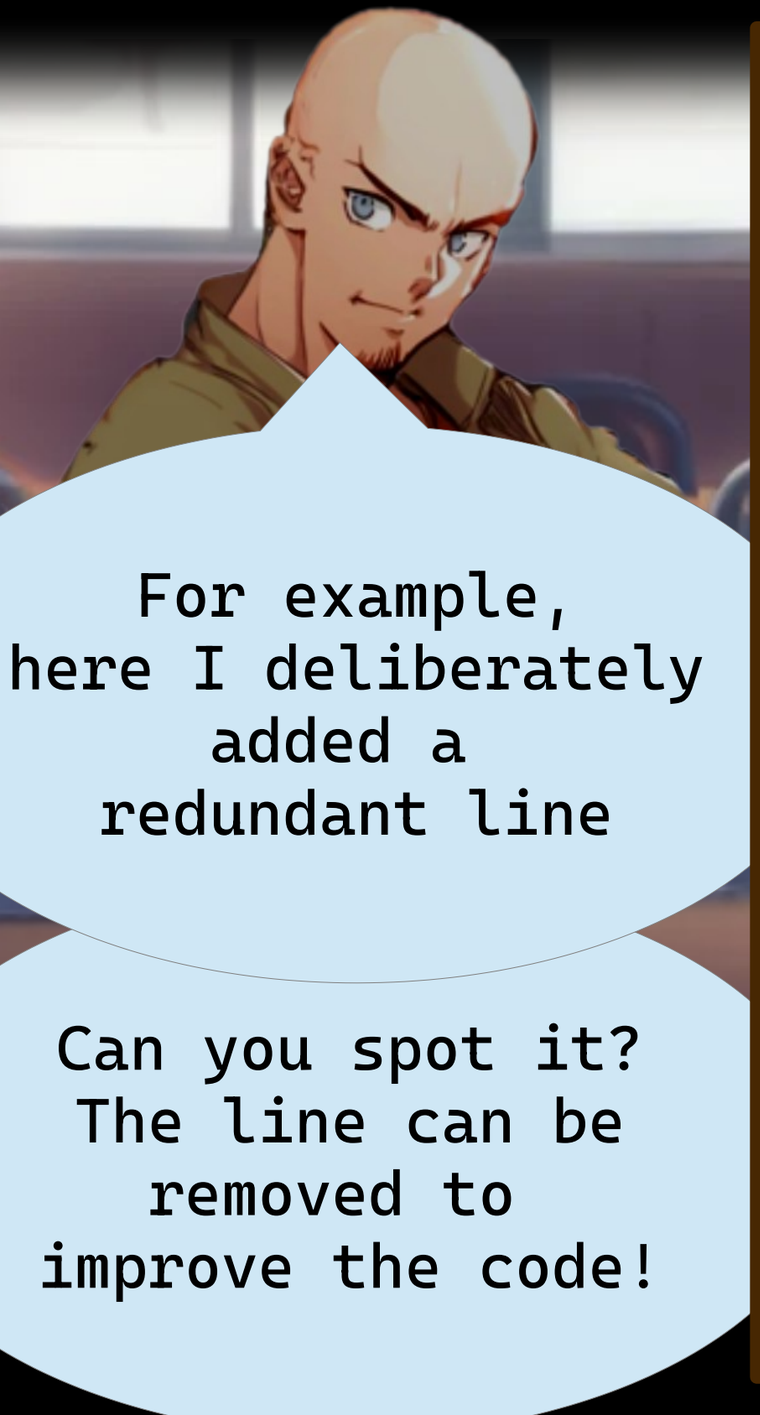

Yes, you do pay them.
The job does not end
after the code is written
record Person(String name, int age){
String greet(String other){
return "Hi "+other+" I'm "+this.name;
}
}


Yes, you do pay them.
The job does not end
after the code is written
More code.
More bugs.
More maintenance
record Person(String name, int age){
String greet(String other){
return "Hi "+other+" I'm "+this.name;
}
}

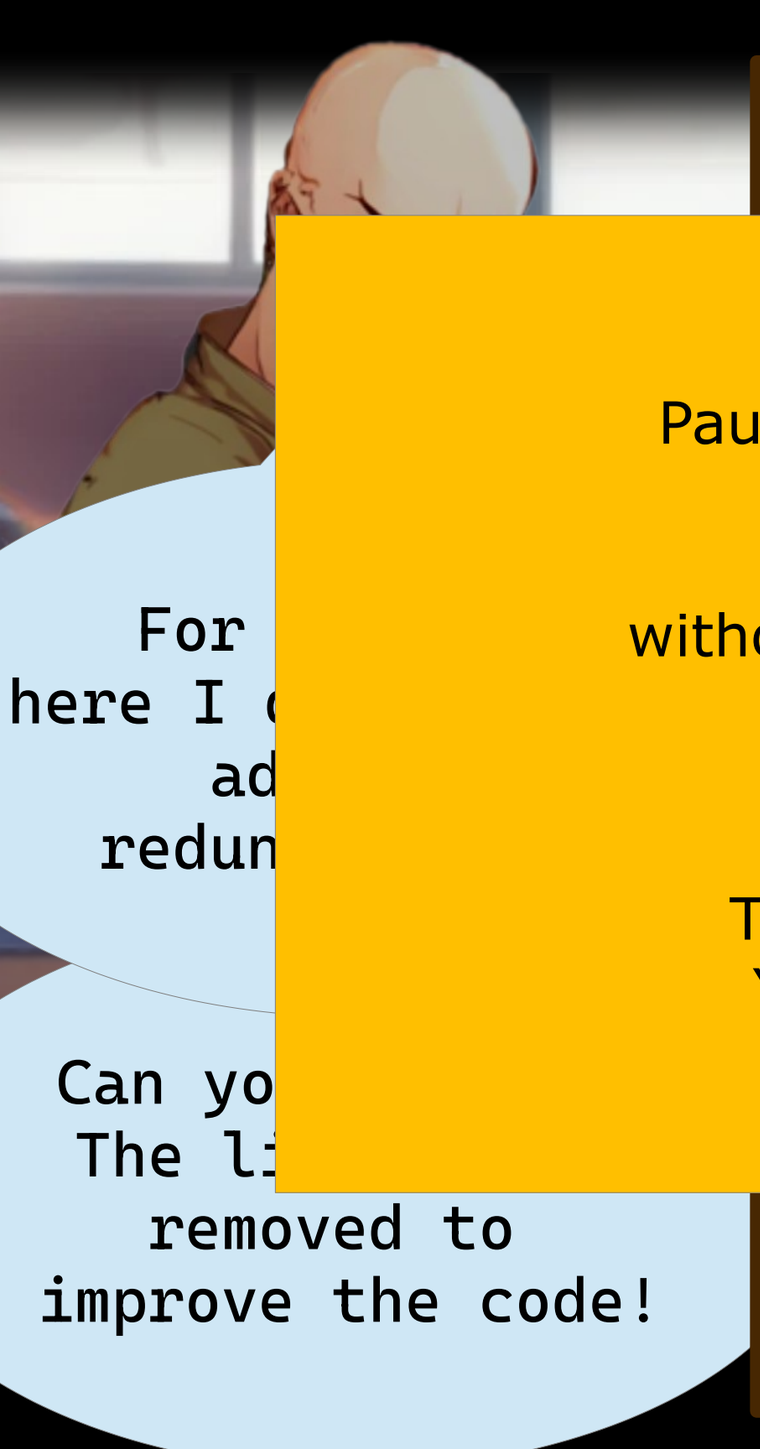
```java
class Person{
  String name;  int age;
  Person(String name,int age){
    this.name=name;  this.age=age;
    }
  public int hashCode(){
    return Objects.hash(age, name);
  }
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return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




```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;
  }
}
```




```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;
  }
}
```



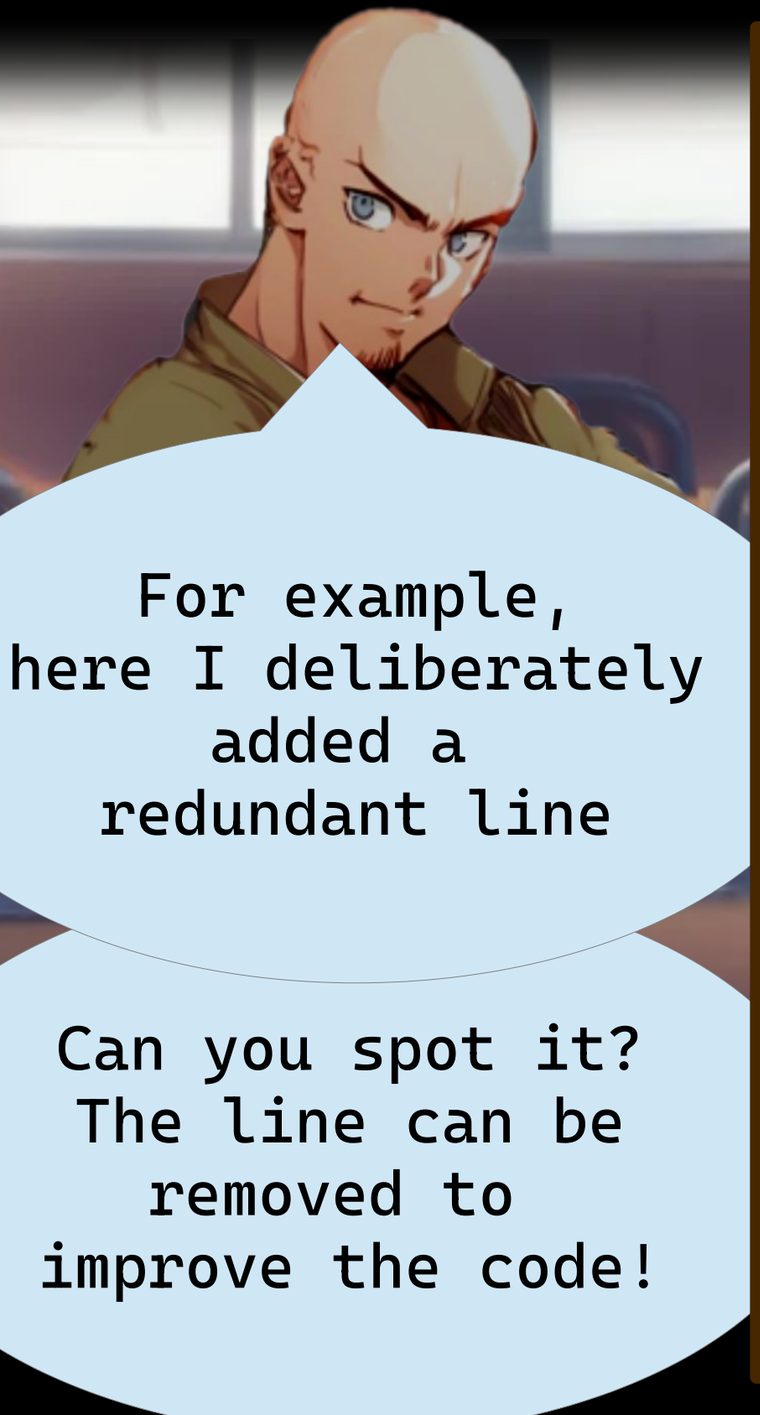

```
class Person{
  String name;  int age;
                                                  false;  }
  }
}
```

Pause the video. Can you solve this one?

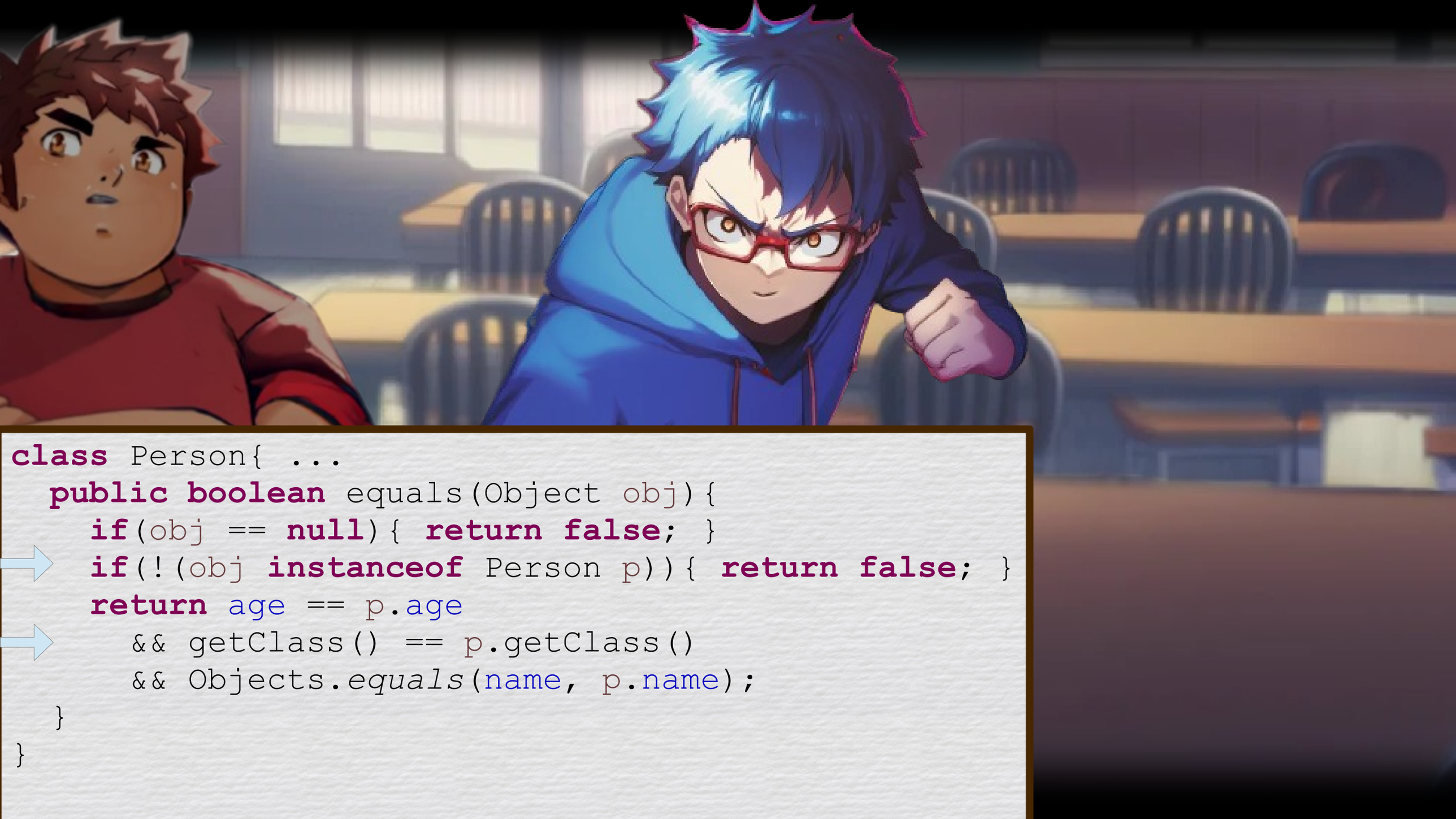
Exactly one line can be removed
without changing the behavior of the code.

Can you find that line?

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




```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;
  }
}
```



```
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);
  }
}
```

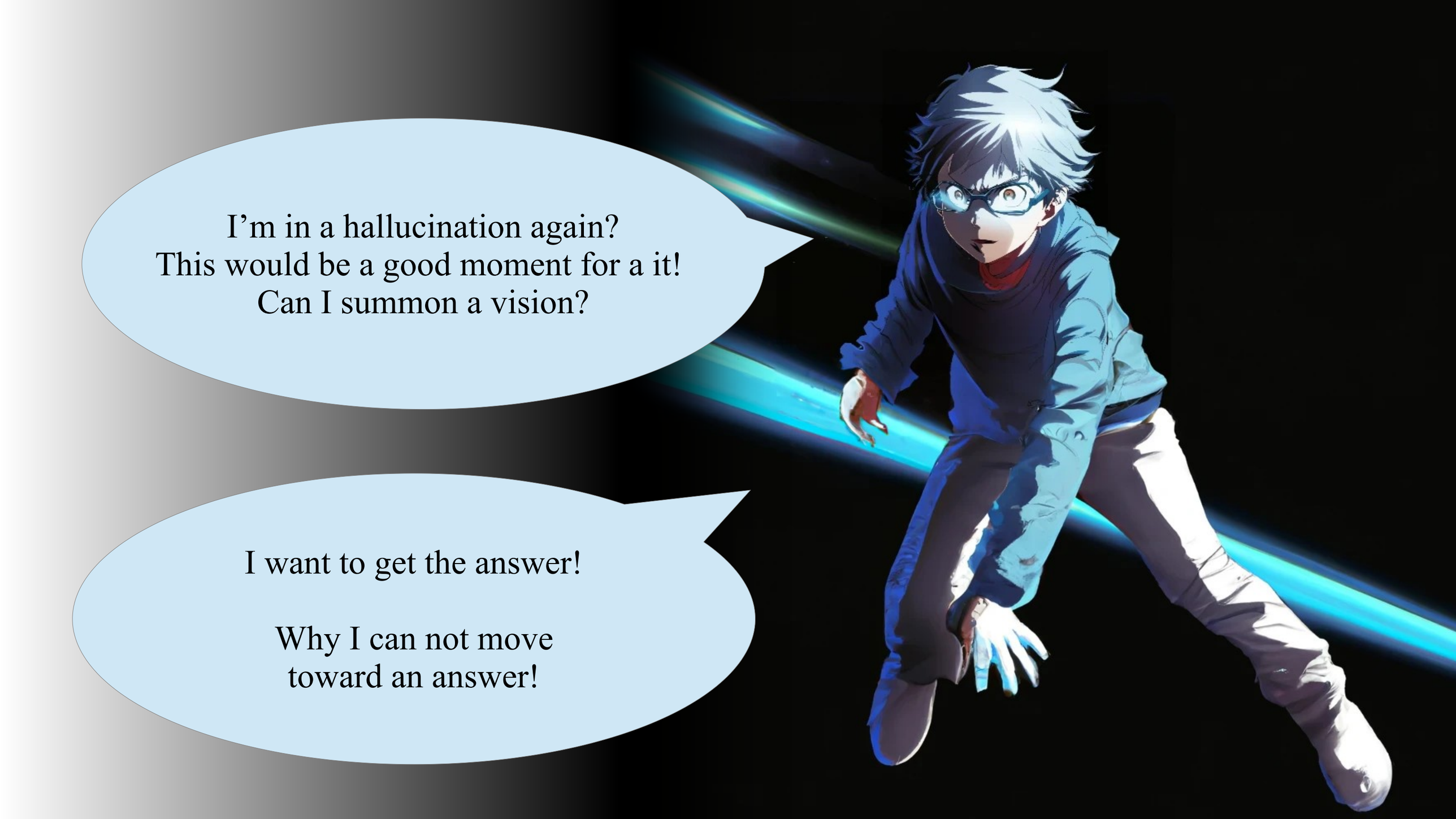

```
class Person{ ...
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return false; }
    return age == p.age
      && getClass() == p.getClass()
      && Objects.equals(name, p.name);
  }
}
```


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


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’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?

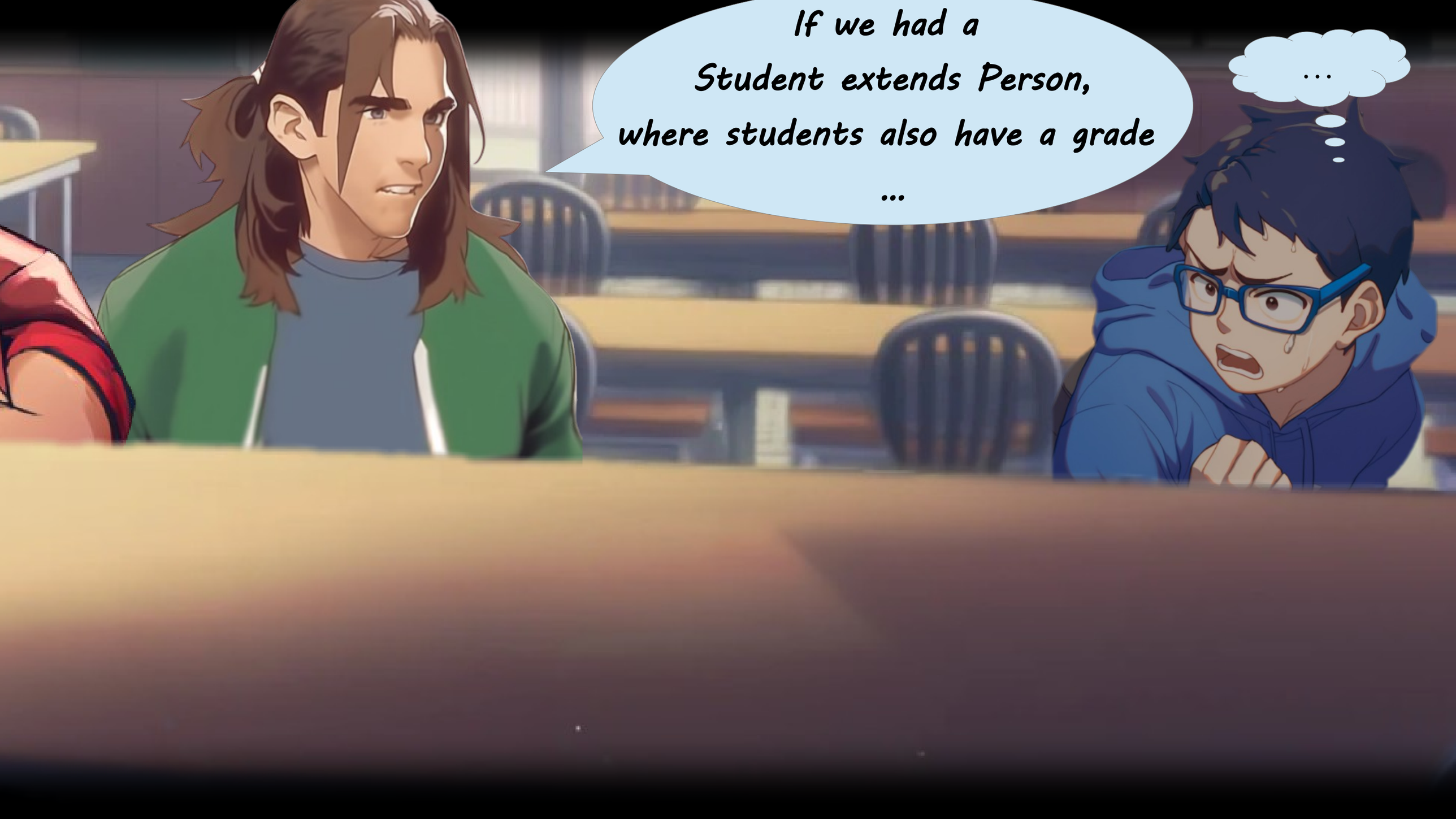

Can you at least tell me …

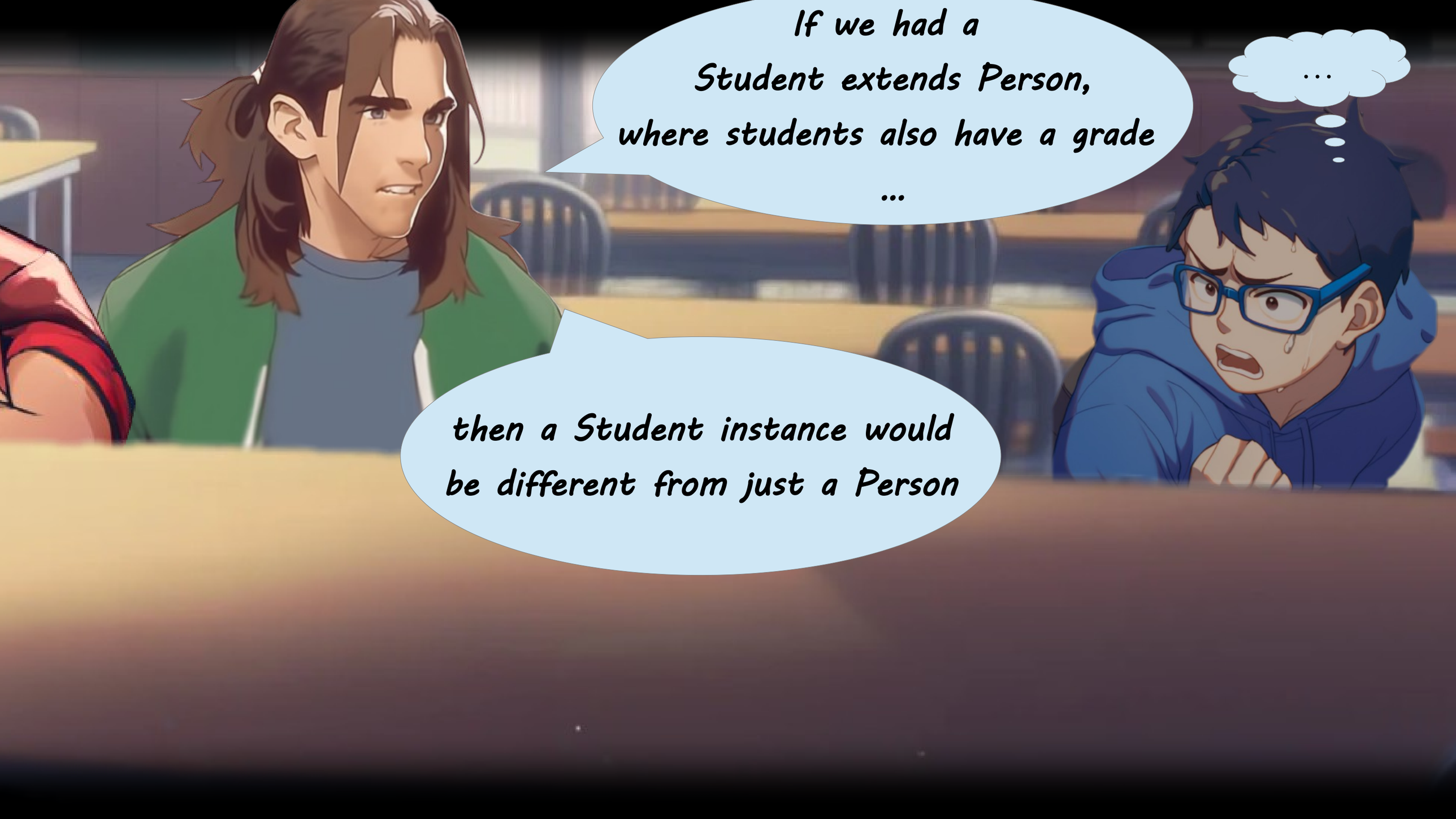

Can you at least tell me …
why it is wrong to remove the class check?

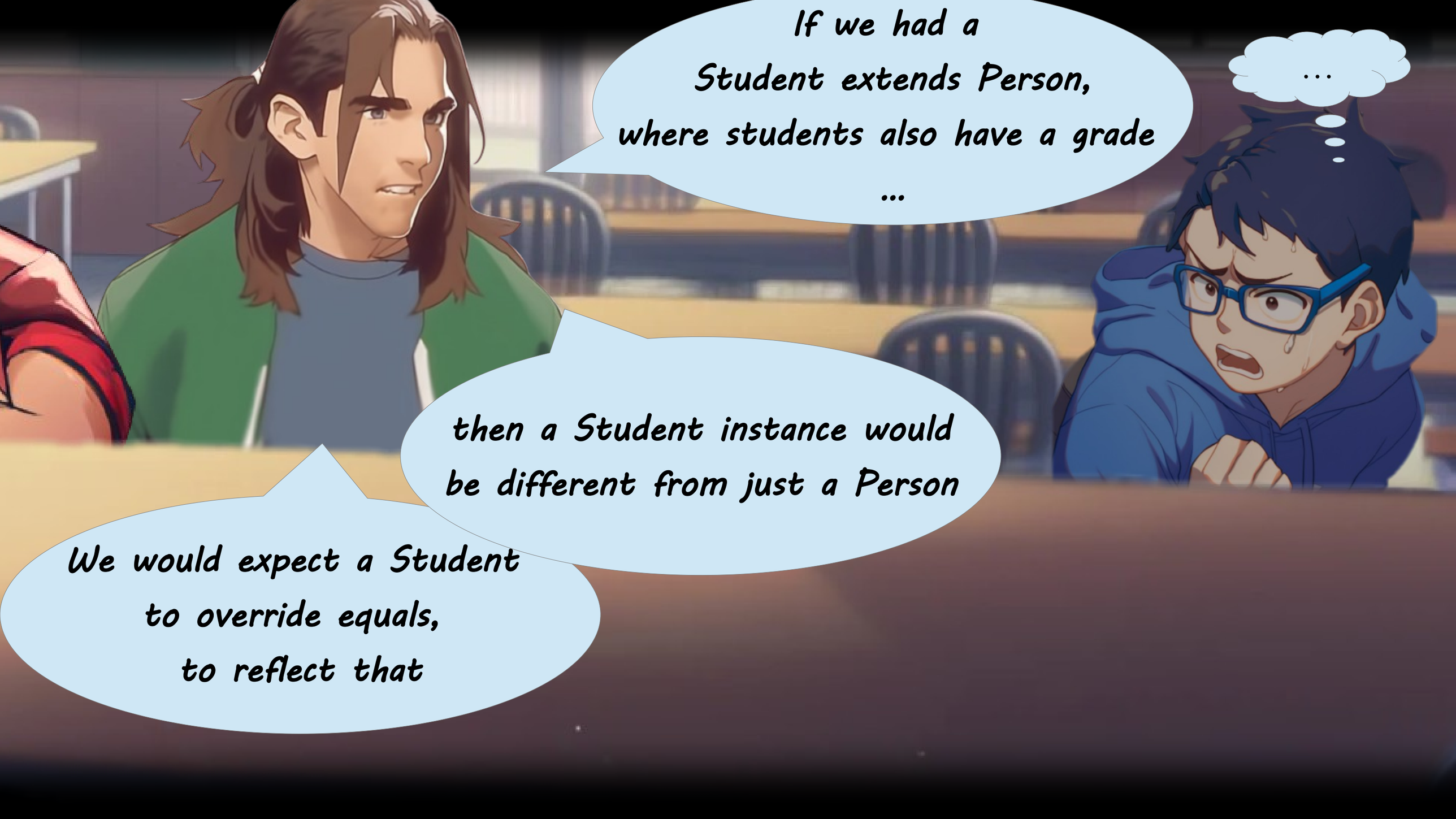

You said equality
has to be reflexive, commutative
and transitive


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


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
...
...


If we had a
Student extends Person,
where students also have a grade
...
then a Student instance would
be different from just a Person
...


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

```java
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);
  }
}
```

Now a student is different from a person

```
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);
  }
}
```

Now a student is different from a person

However ...

```
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);
  }
}
```

a person can still be equal to a student

Now a student is different from a person

However ...

```
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);
  }
}
```

Now a student is different from a person

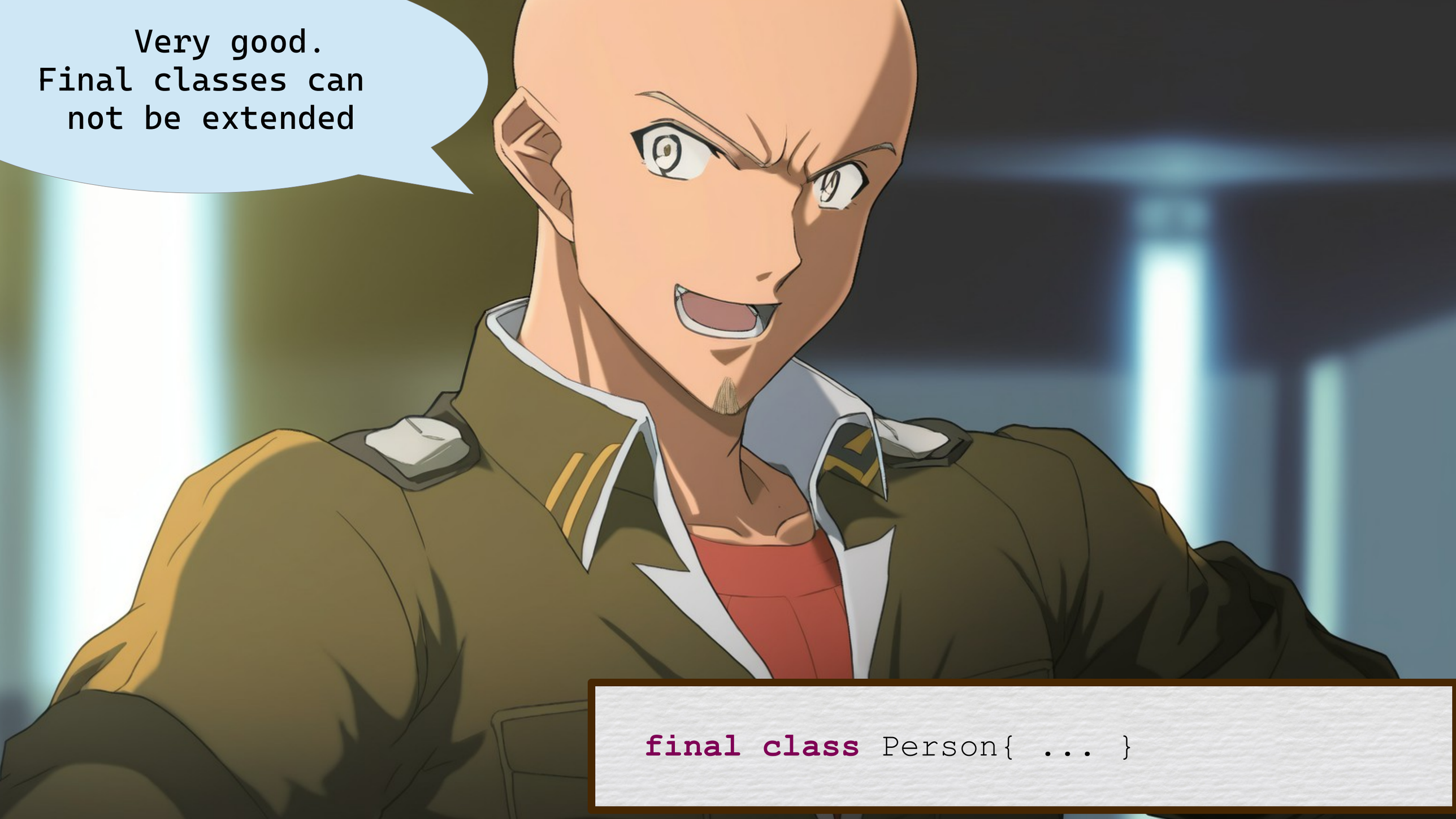
However ...

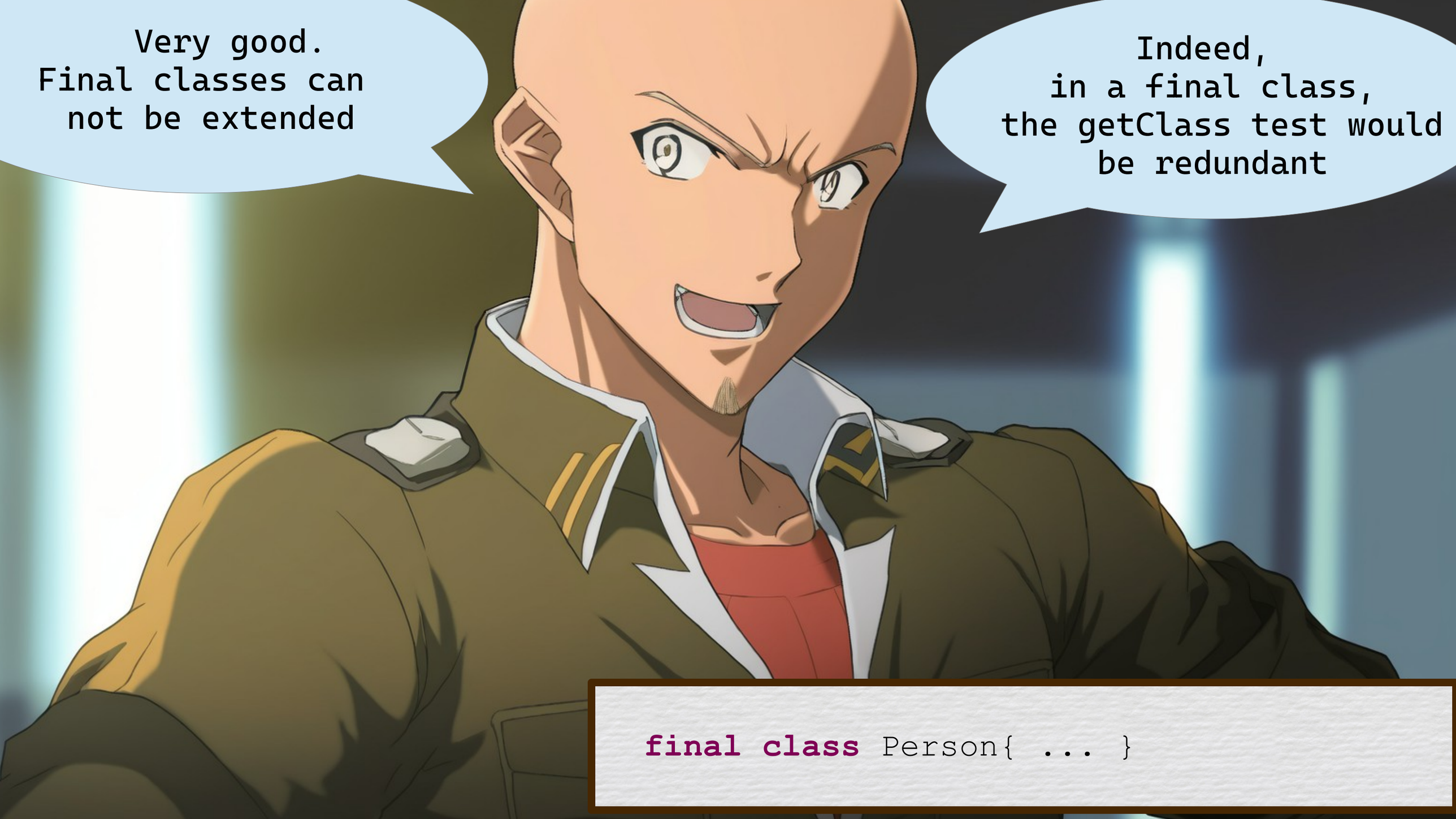
```
class Person{ ...
  public boolean equals(Object obj){
    if(obj == null){ return false; }
    if(!(obj instanceof Person p)){ return false; }
    return age == p.age
      && Objects.equals(name, p.name);
  }
}
class Student extends Person{ ...
  int grade;
  public boolean equals(Object obj){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

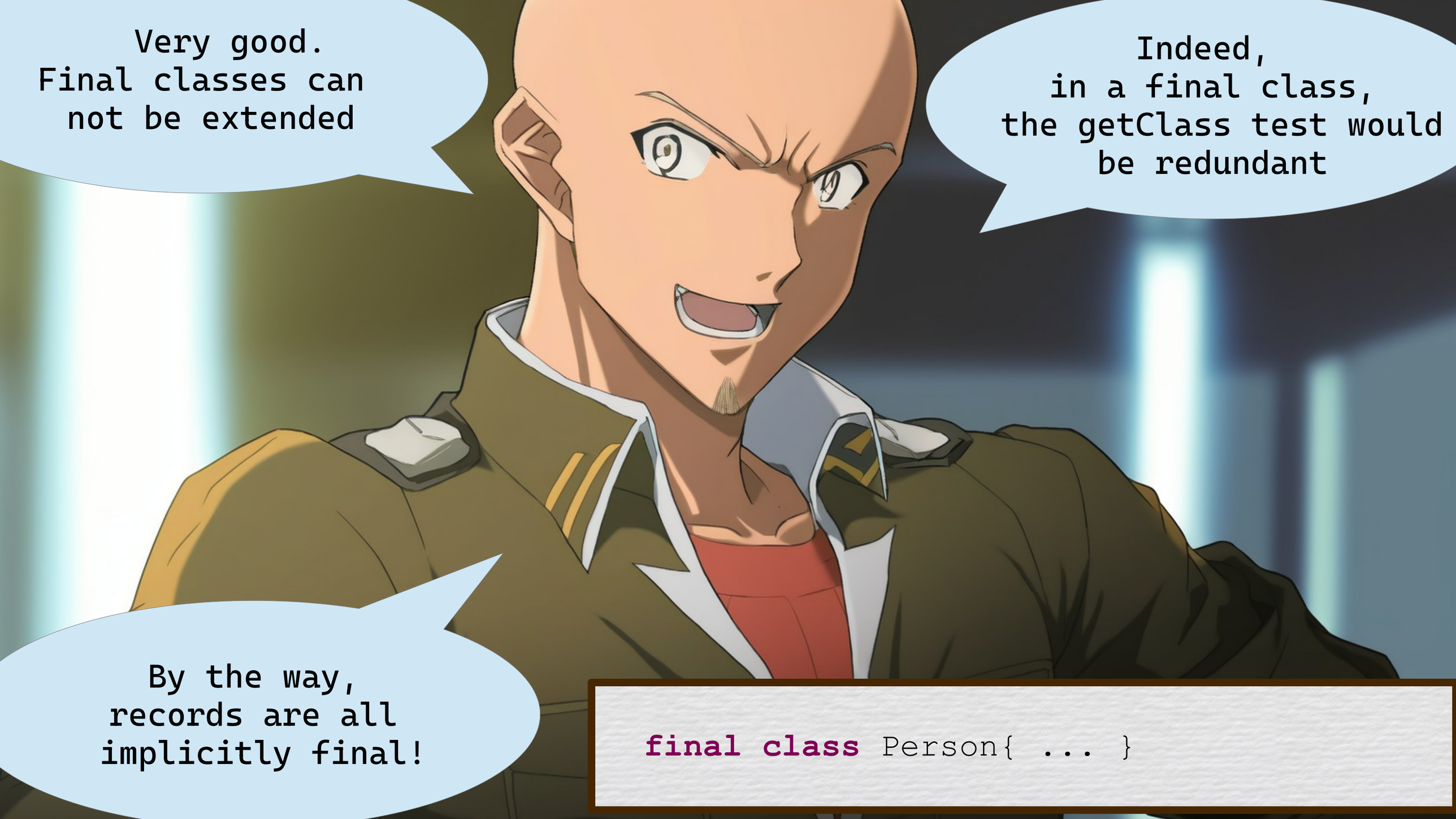
a person can still be equal to a student

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


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


Very good.
Final classes can
not be extended
final class Person{ ... }

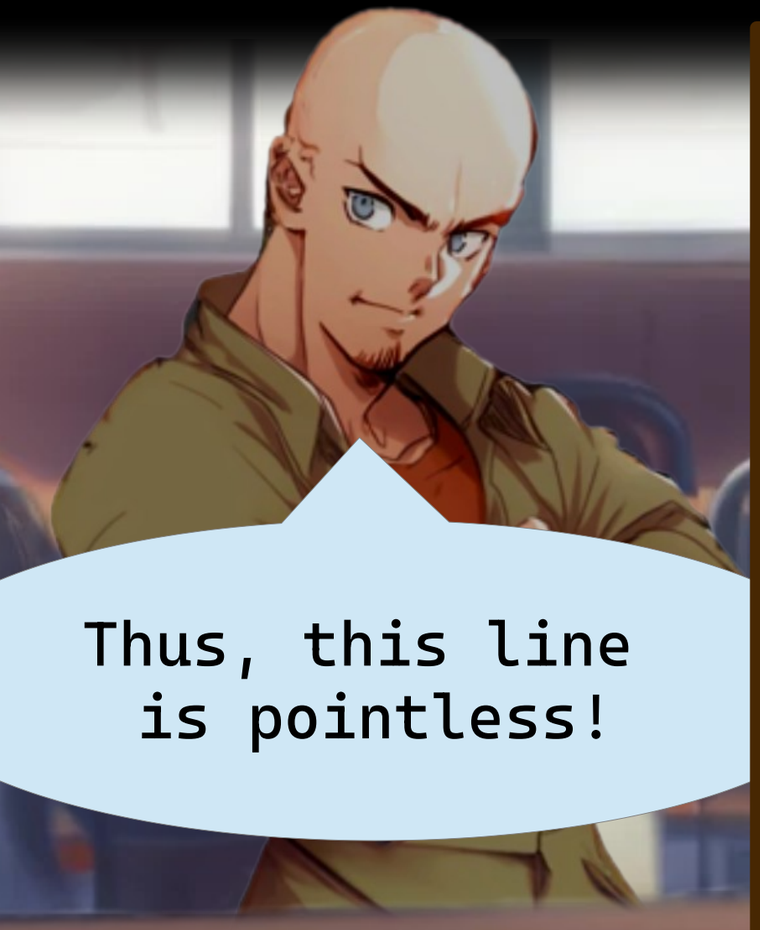

Very good.
Final classes can
not be extended
Indeed,
in a final class,
the getClass test would
be redundant
final class Person{ ... }


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:


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


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;
  }
}
```

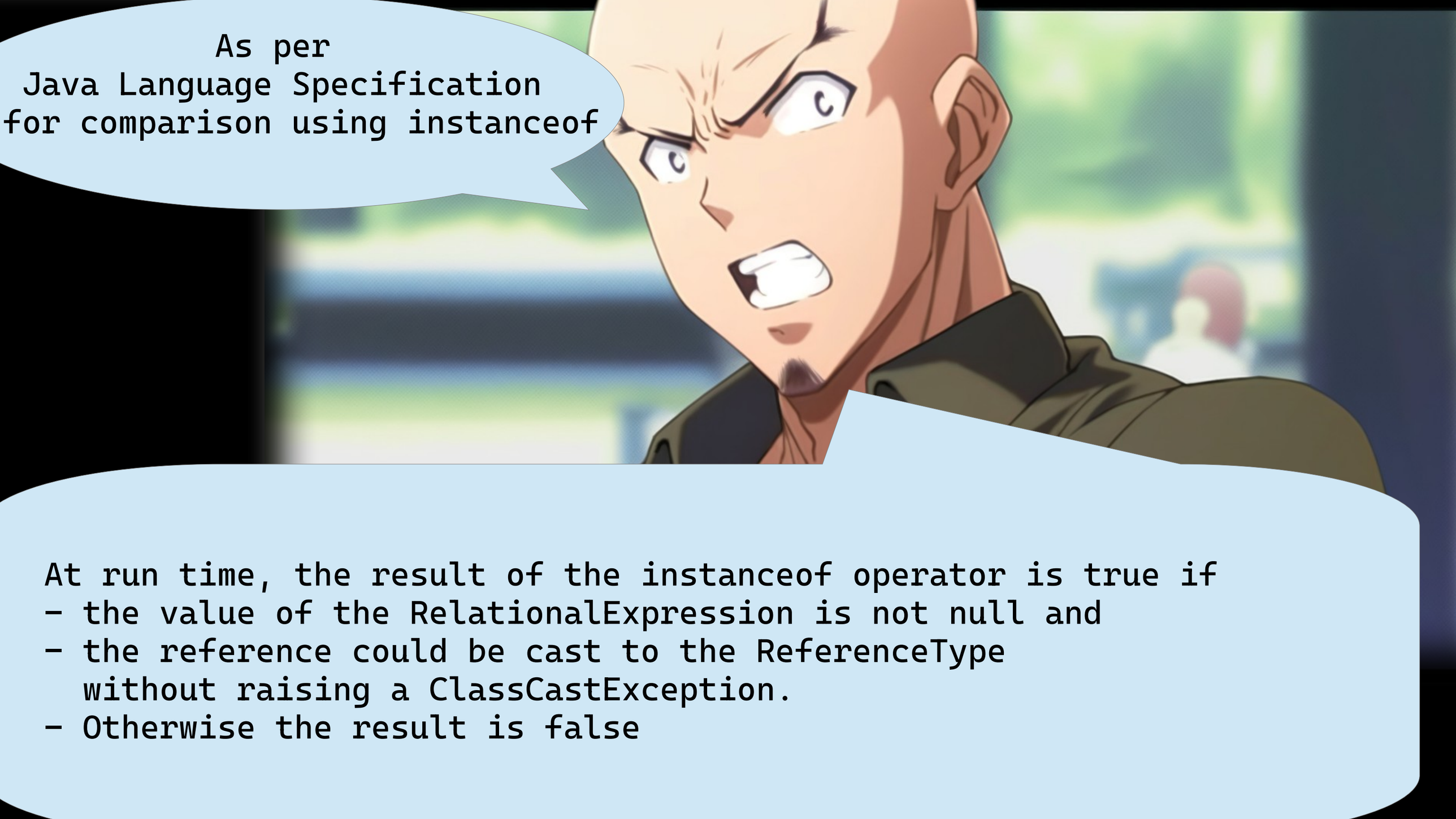

But, …
assigning null to a Person would work,
thus null is an instance of Person.


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.

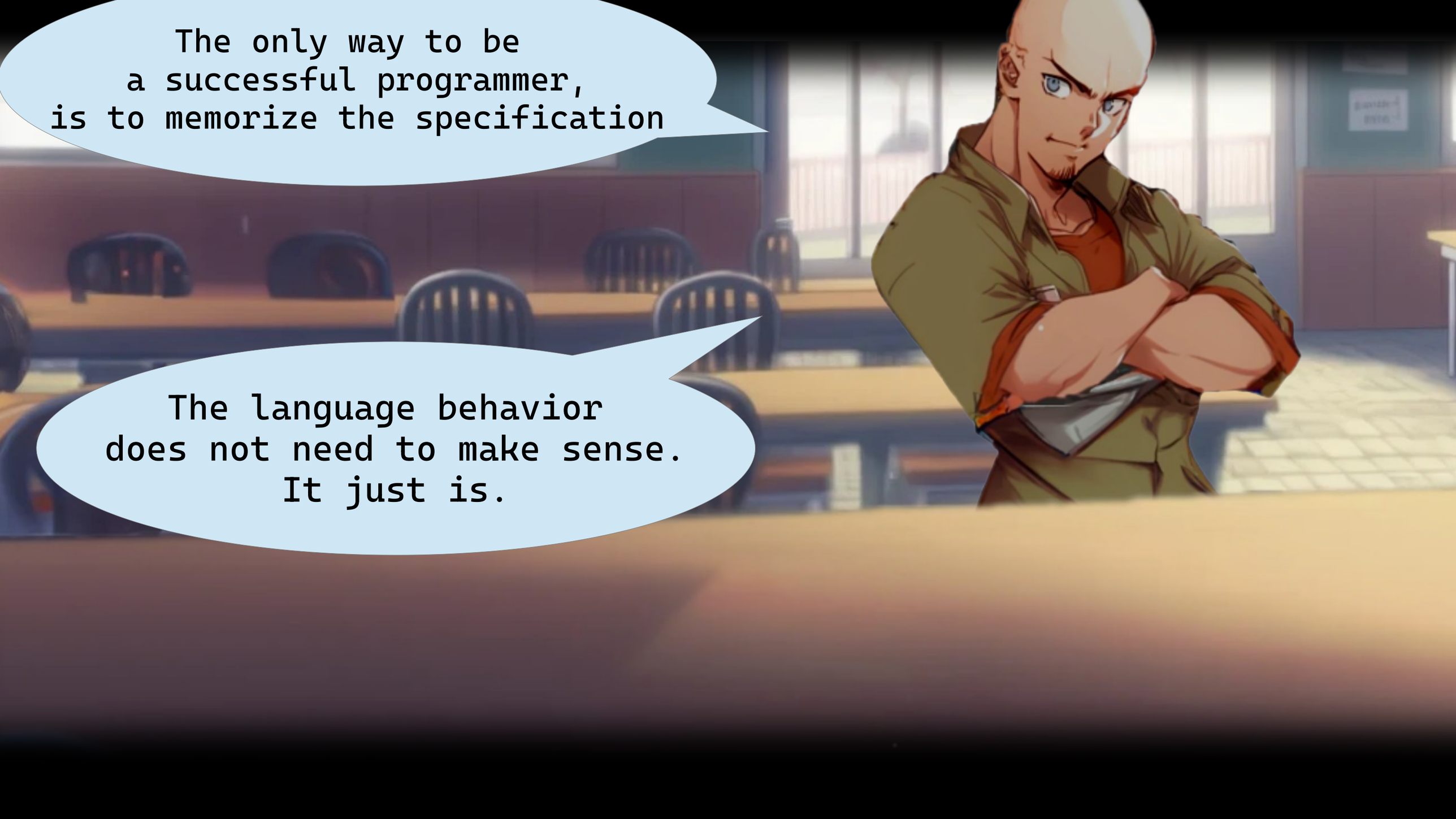

Yes, null can be assigned to a Person. It can be cast to Person too.
But, … assigning null to a Person would work, thus null is an instance of Person.
So, null must be an instance of Person!

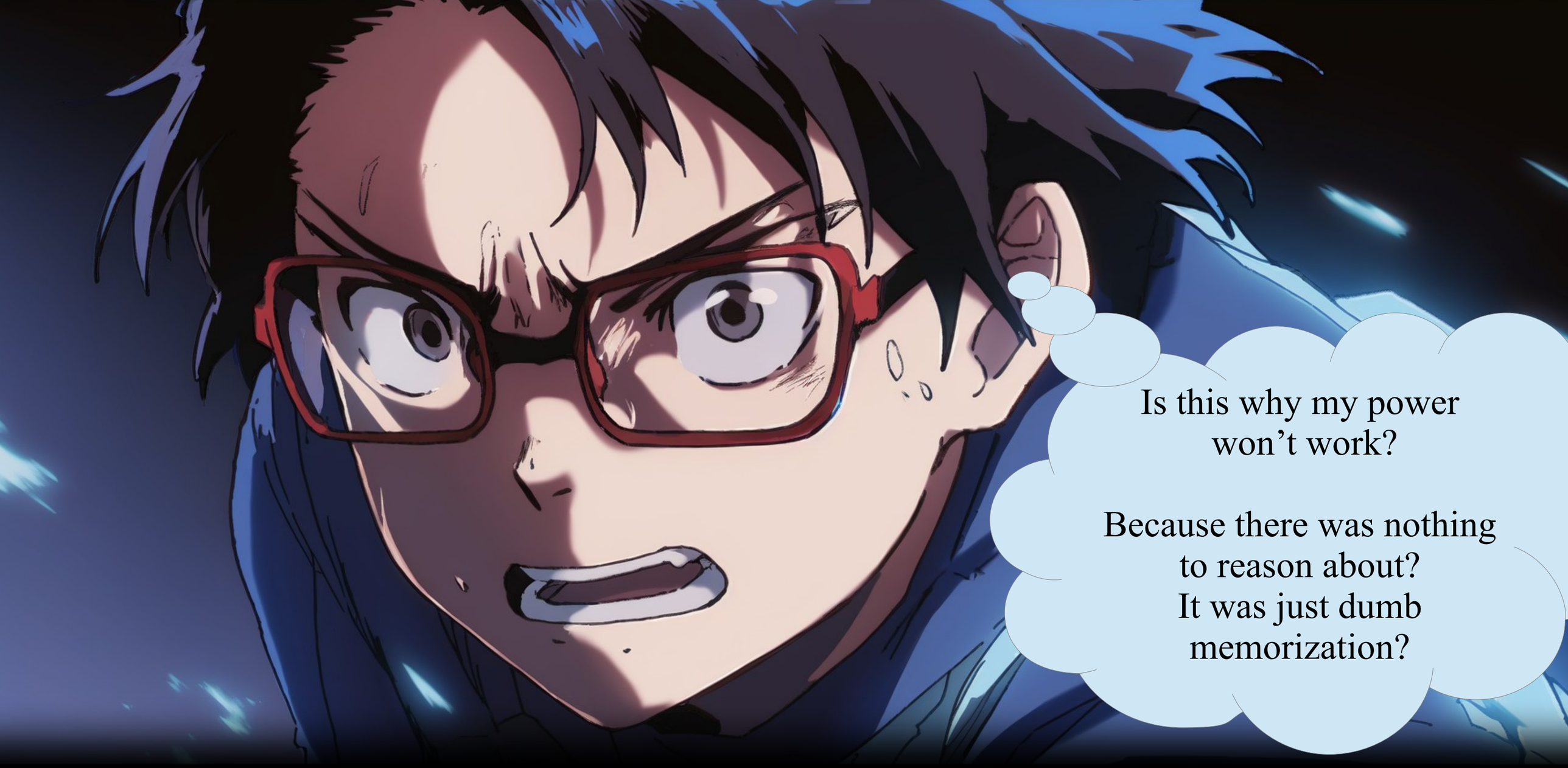

Yes, null can be assigned to a Person. It can be cast to Person too.
But, … assigning null to a Person would work, thus null is an instance of Person.
So, null must be an instance of Person!
It would be insane otherwise!


As per
Java Language Specification
for comparison using instanceof


As per
Java Language Specification
for comparison using instanceof
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 only way to be
a successful programmer,
is to memorize the specification
The language behavior
does not need to make sense.
It just is.

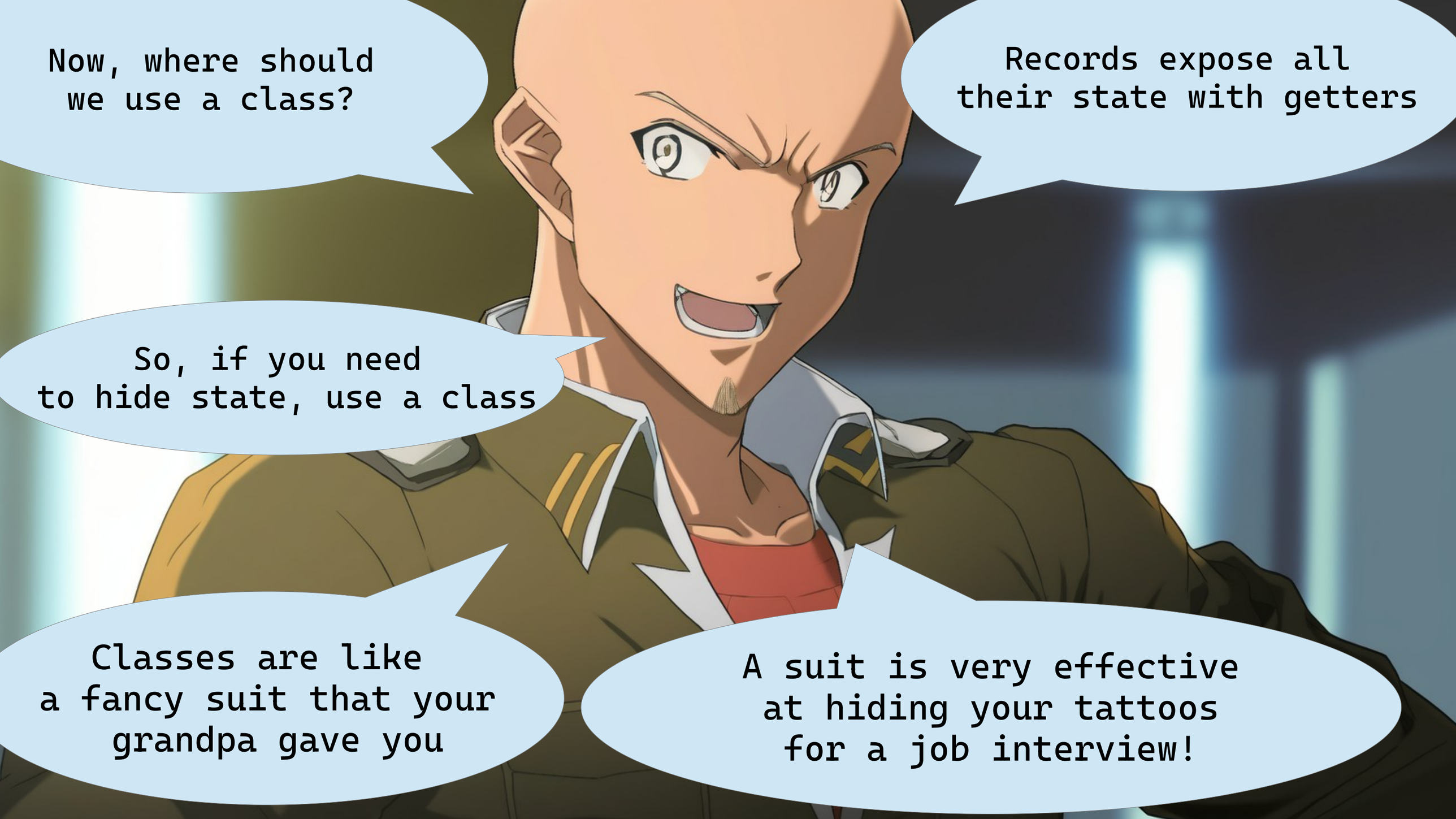

Is this why my power
won’t work?
Because there was nothing
to reason about?
It was just dumb
memorization?

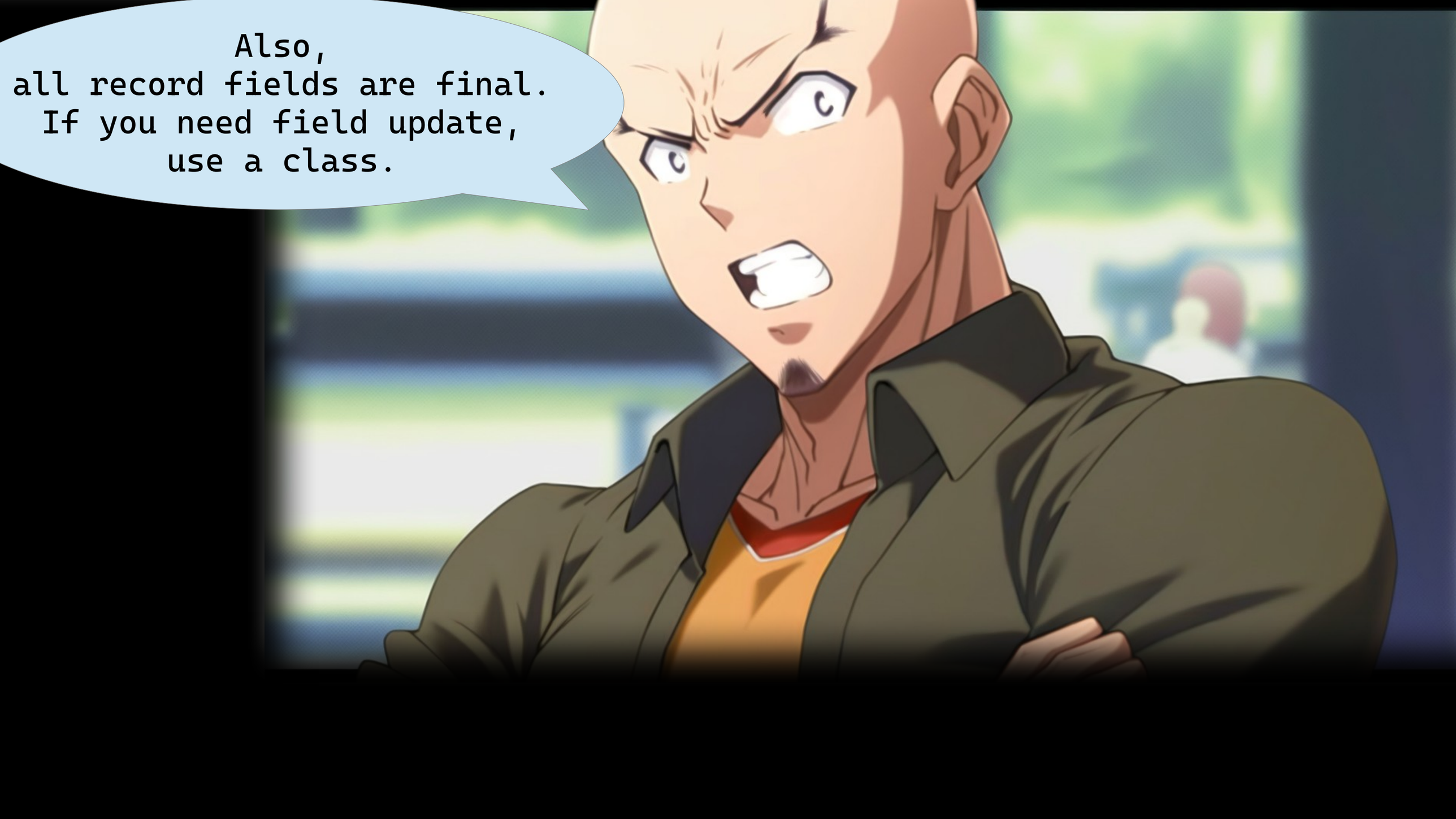

Now, where should we use a class?

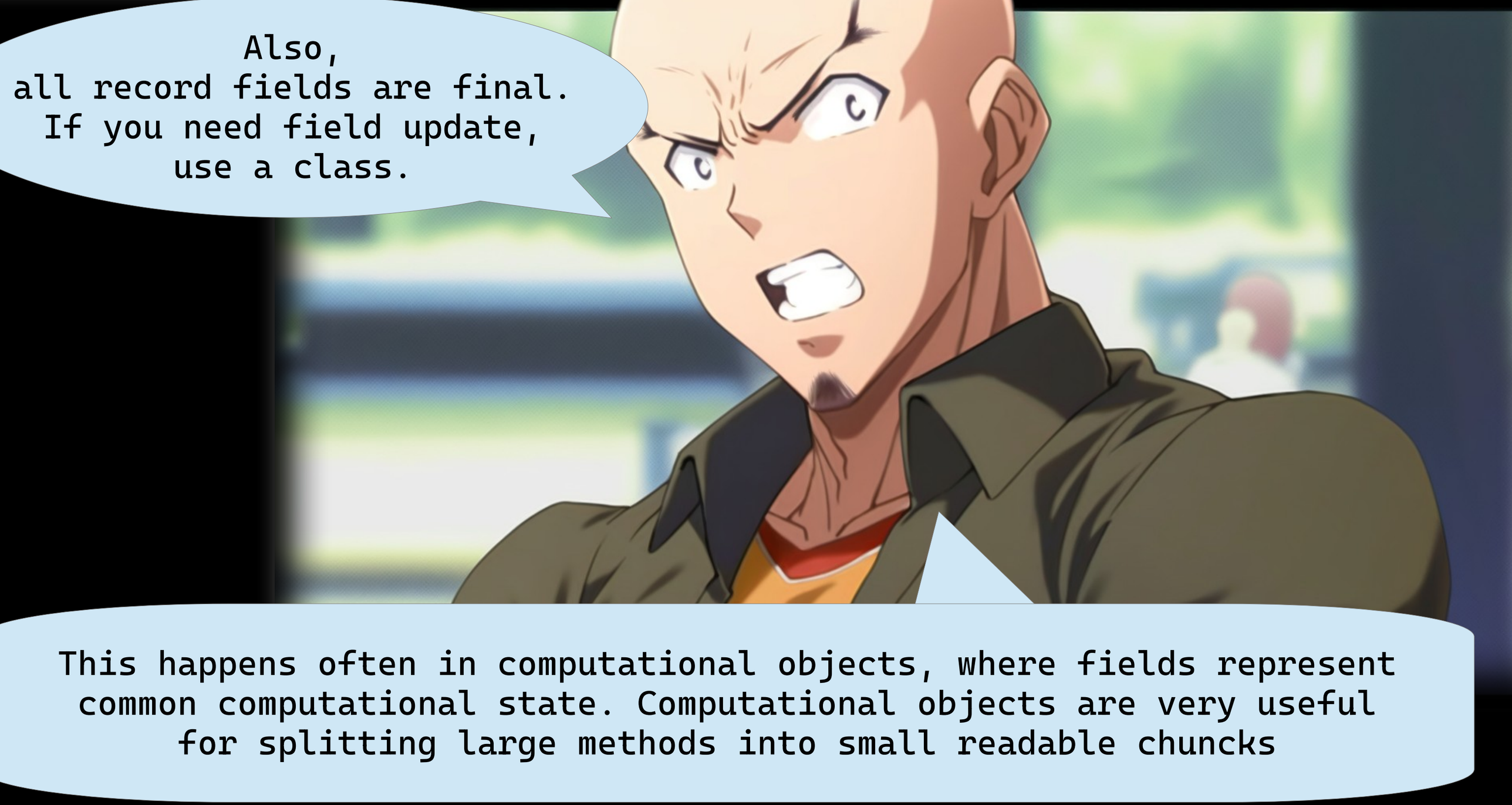

Now, where should we use a class?
Records expose all their state with getters


Now, where should we use a class?
Records expose all their state with getters
So, if you need to hide state, use a class

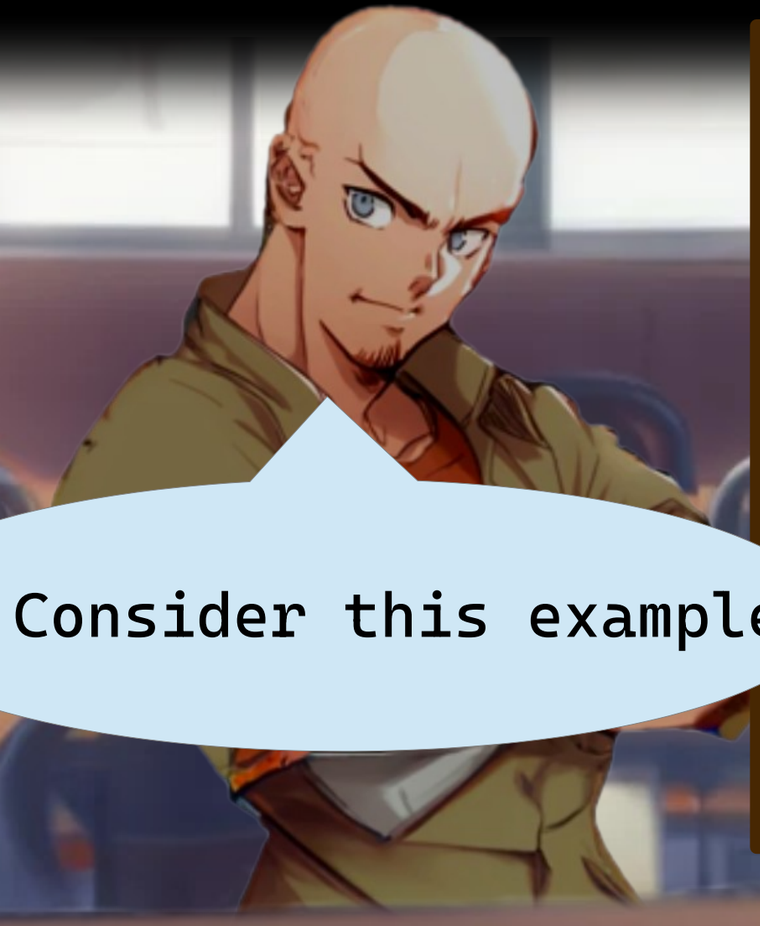

Now, where should we use a class?
Records expose all their state with getters
So, if you need to hide state, use a class
Classes are like a fancy suit that your grandpa gave you


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!

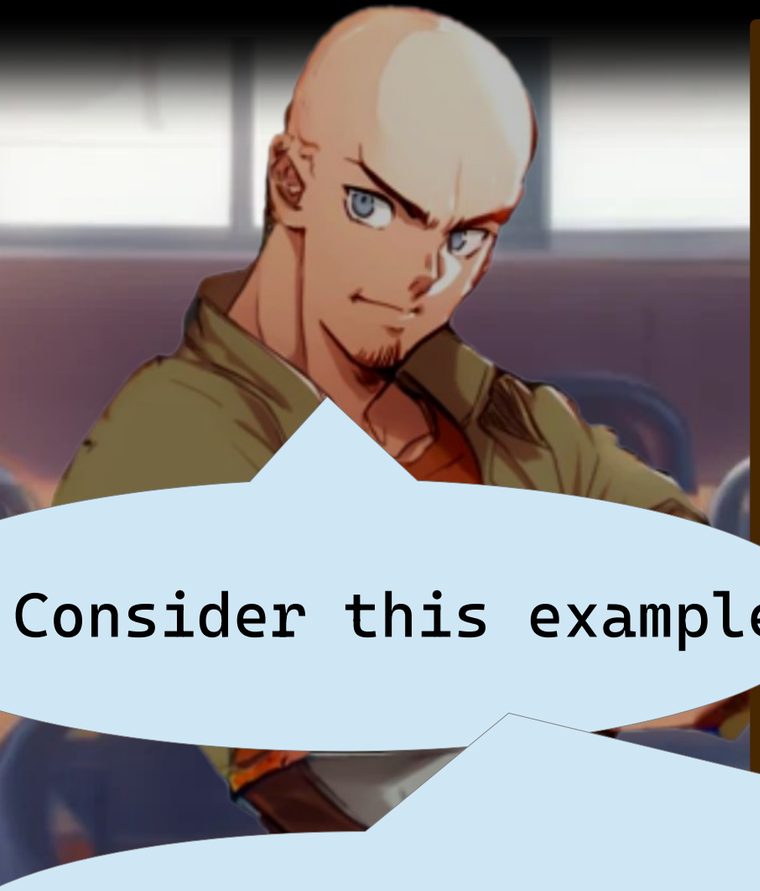

Also,
all record fields are final.
If you need field update,
use a class.


Also,
all record fields are final.
If you need field update,
use a class.
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

```java
//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; }
    });
  }
}
```




```java
//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; }
    });
  }
}
```



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