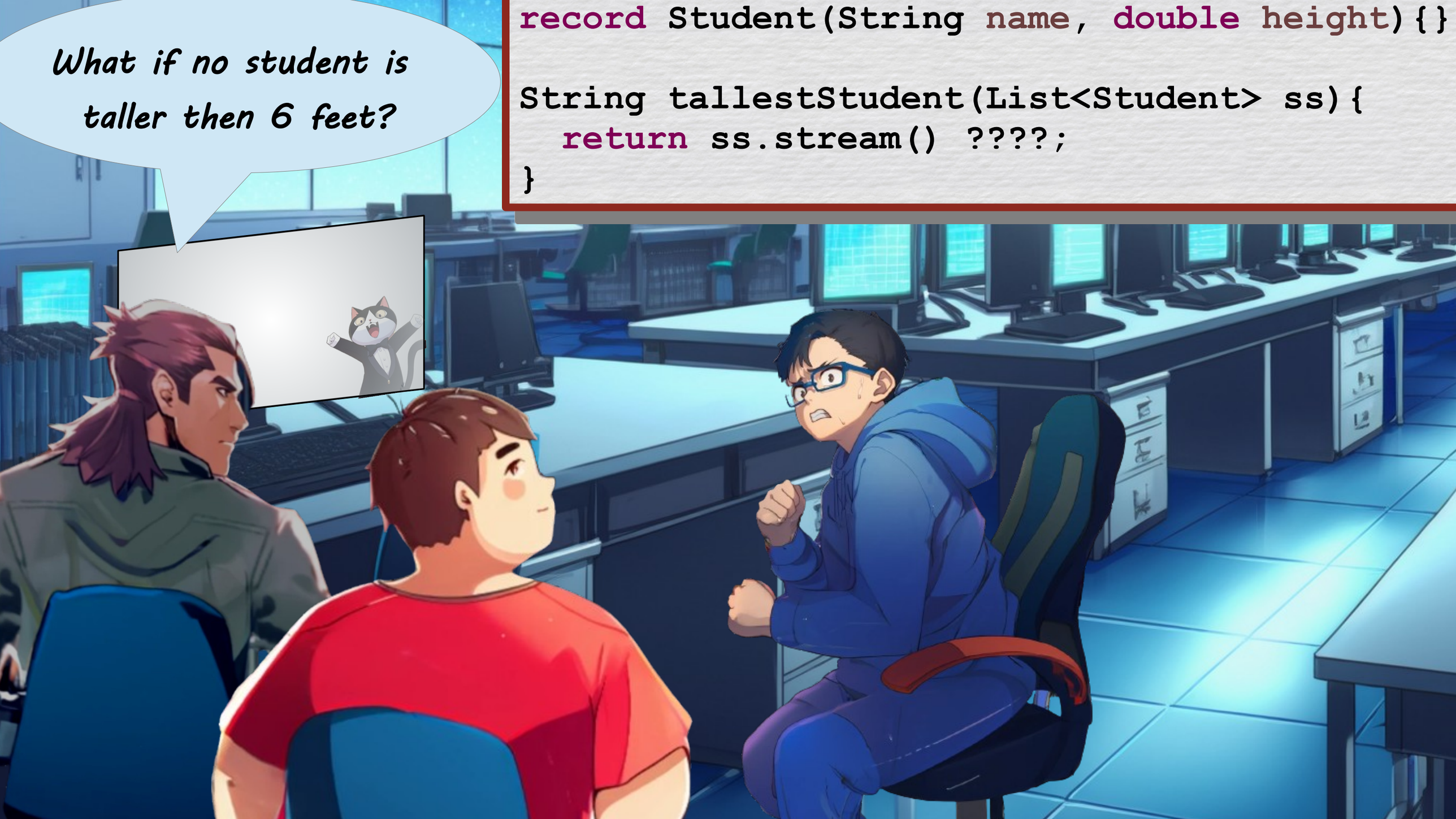
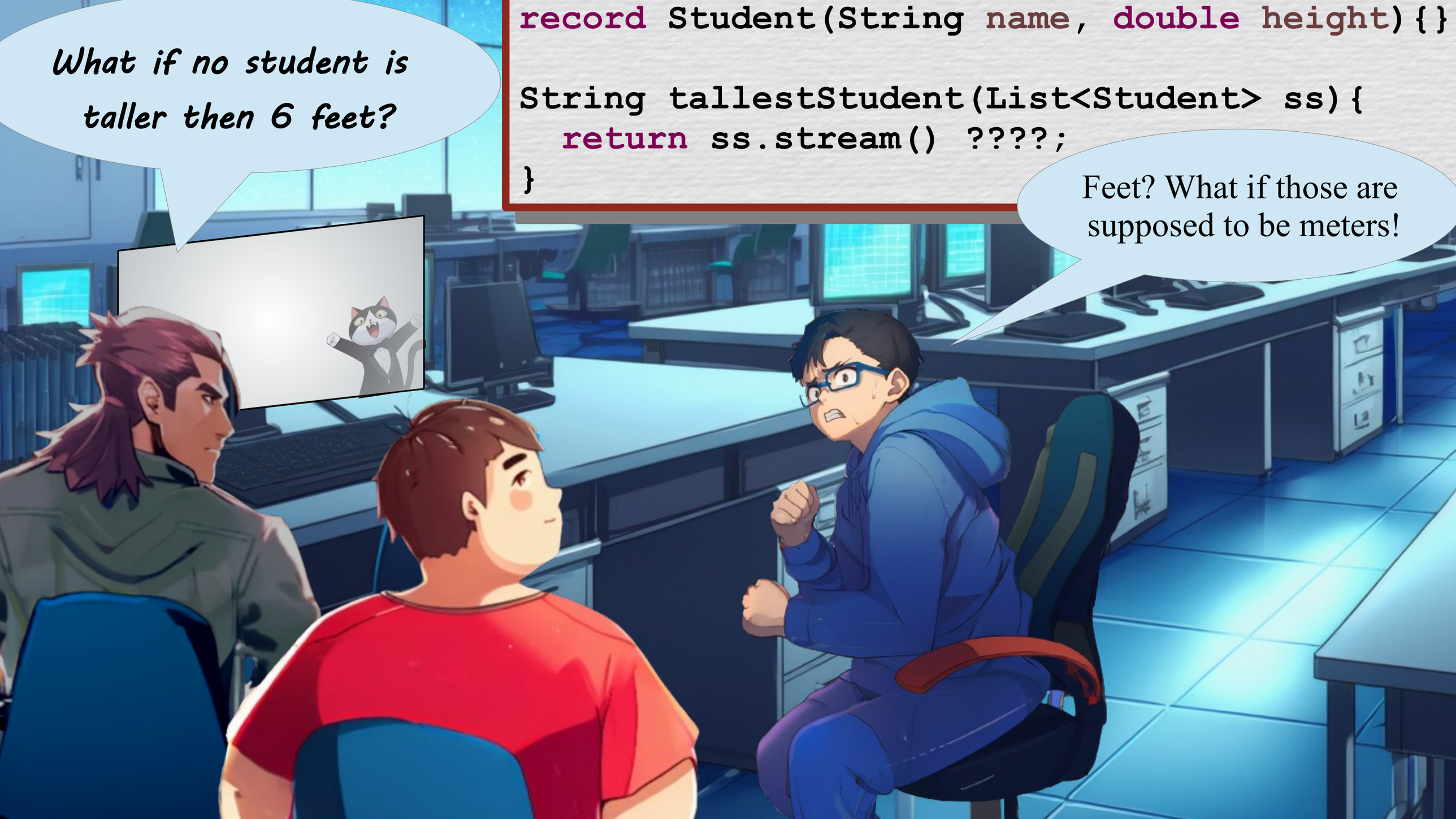
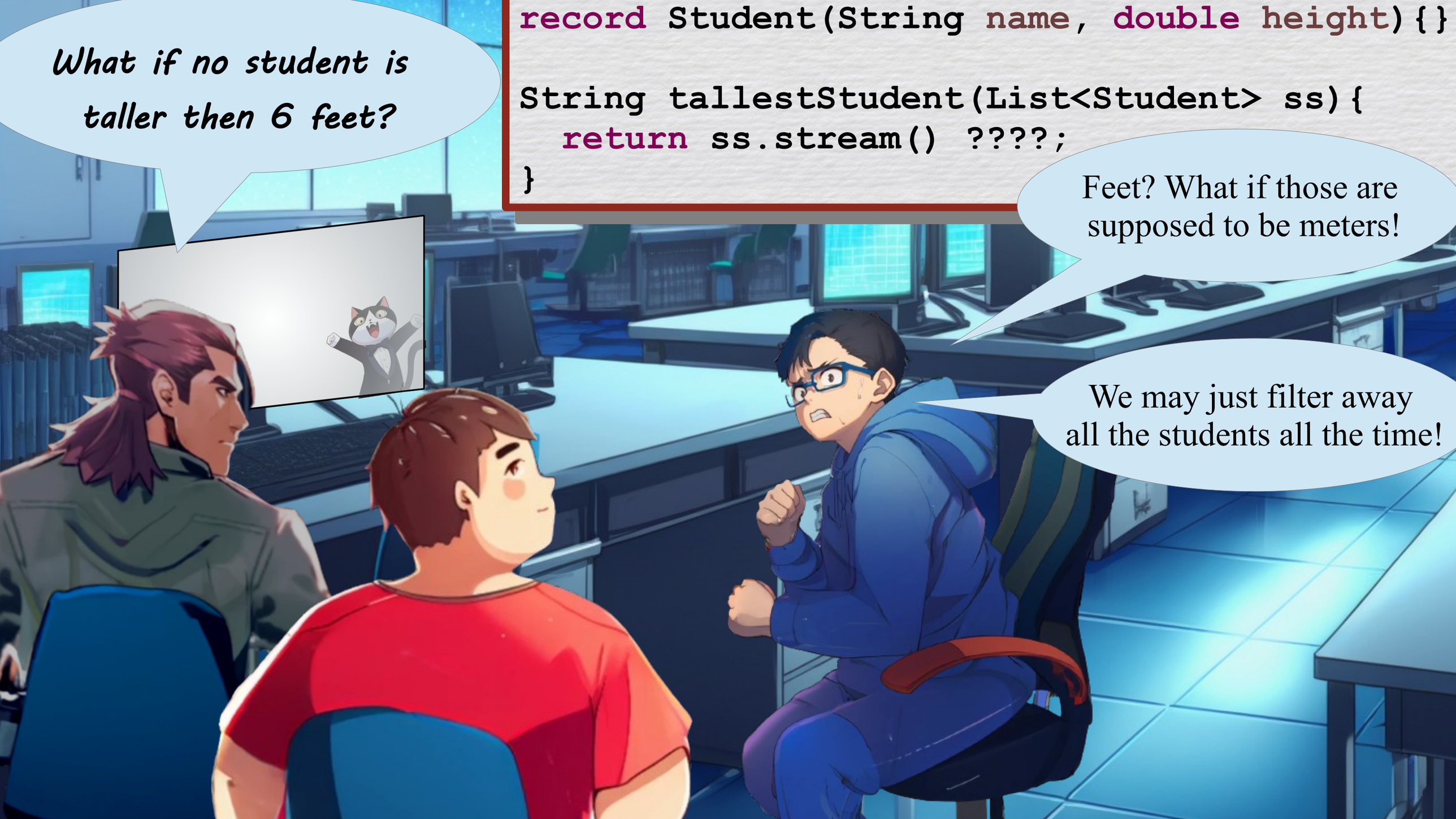
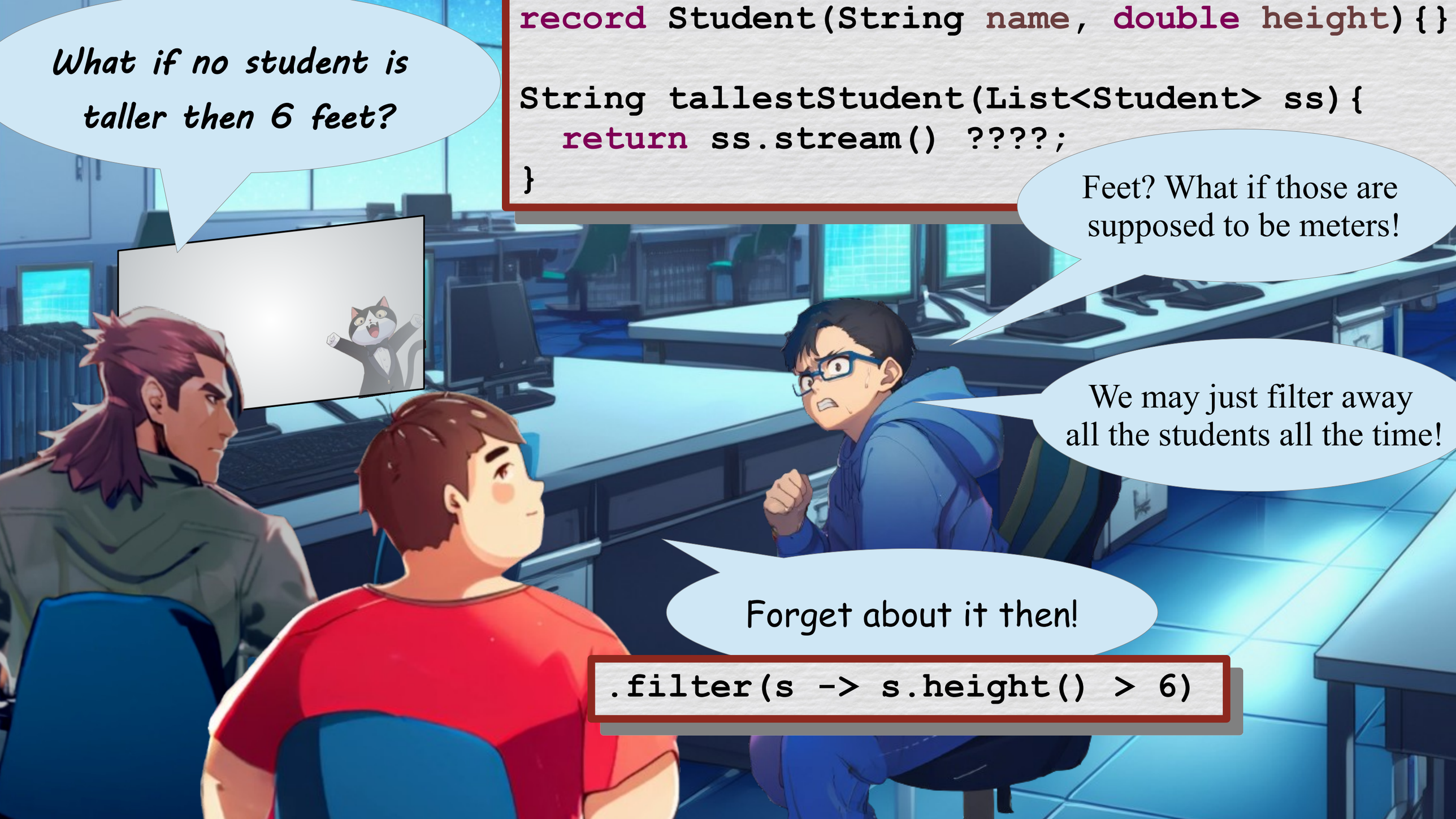
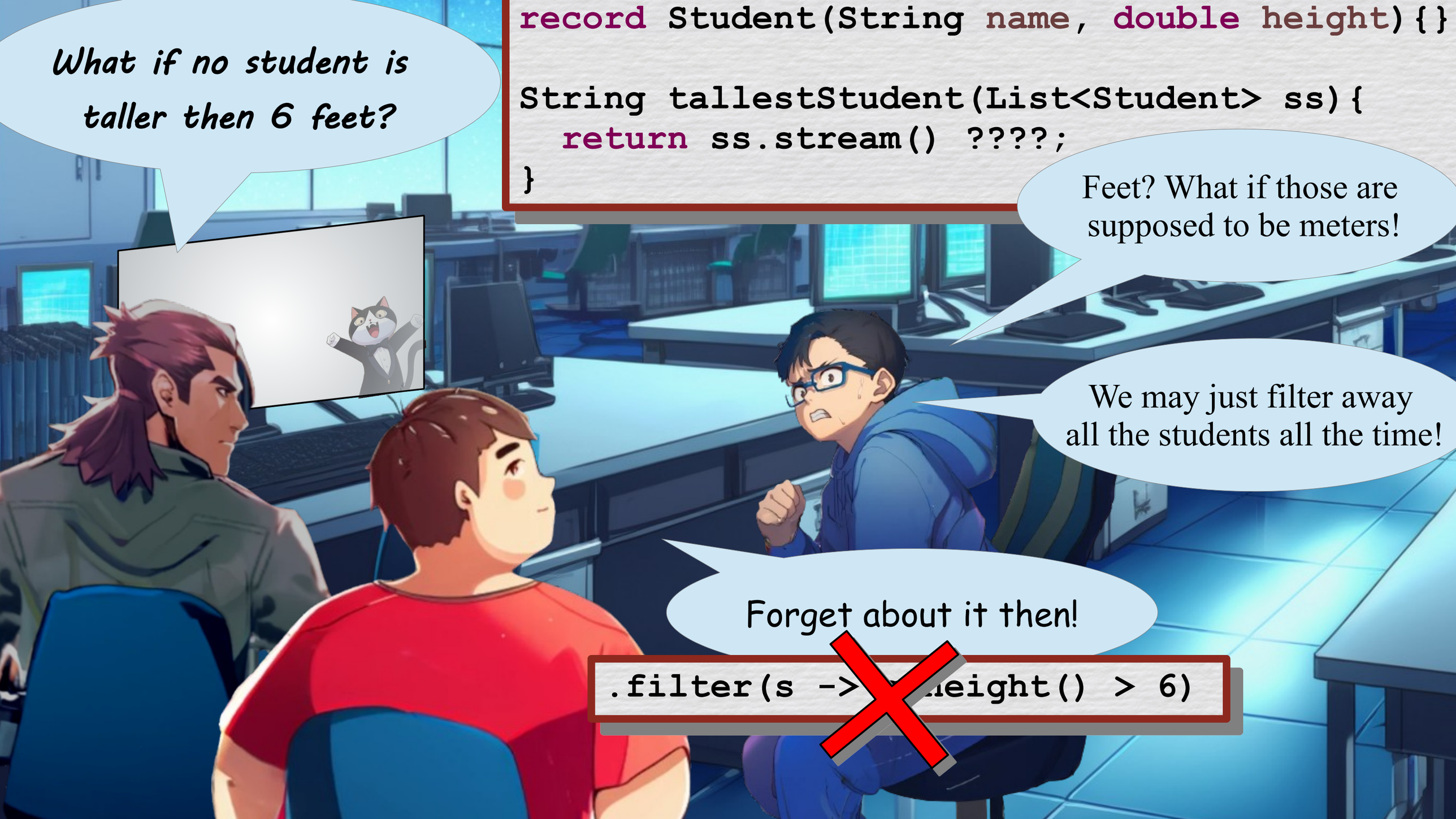

What if no student is taller then 6 feet?
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream() ????;
}


What if no student is taller then 6 feet?
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream() ????;
}
Feet? What if those are supposed to be meters!


What if no student is taller then 6 feet?
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream() ????;
}
Feet? What if those are supposed to be meters!
We may just filter away all the students all the time!


What if no student is taller then 6 feet?
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream() ????;
}
Feet? What if those are supposed to be meters!
We may just filter away all the students all the time!
Forget about it then!
.filter(s -> s.height() > 6)


What if no student is taller then 6 feet?
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream() ????;
}
Feet? What if those are supposed to be meters!
We may just filter away all the students all the time!
Forget about it then!
.filter(s -> eight() > 6)

```
.map(s -> s.name())
```

```
.get()
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
  return ss.stream() ????;
}
```

```
.max(Comparator.comparingDouble(s -> s.height()))
```

```
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
```

```
.findFirst()
```

There are two possible options here for finding the tallest;

```
.forEach(s -> System.out.println(s.name()))
```

```
.orElseThrow(()->new Error("Empty student list provide
```

```
.map(s -> s.name())
```

```
.get()
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
  return ss.stream() ????;
}
```

```
.max(Comparator.comparingDouble(s -> s.height()))
```

```
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
```

```
.findFirst()
```

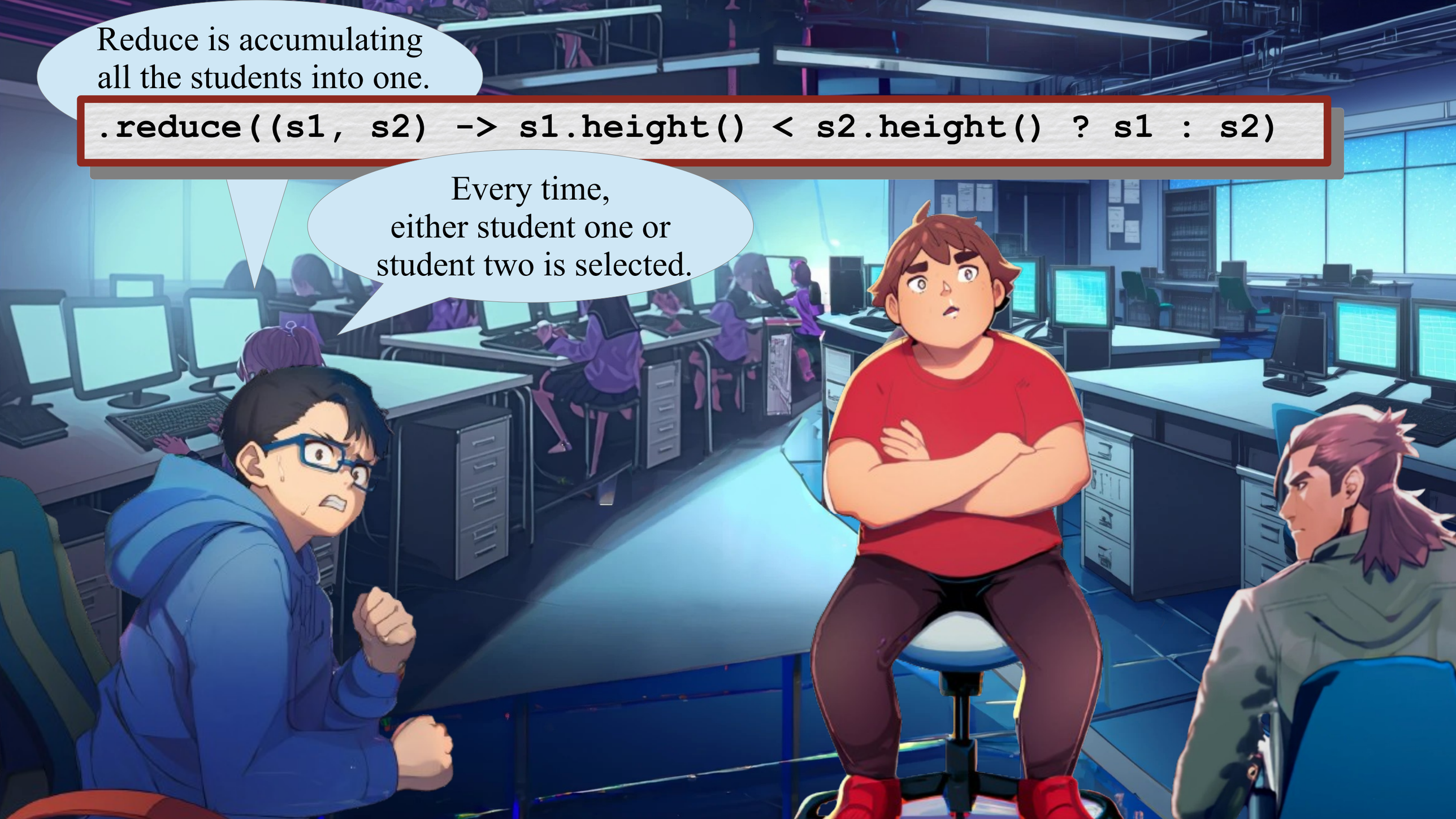
There are two possible options here for finding the tallest;

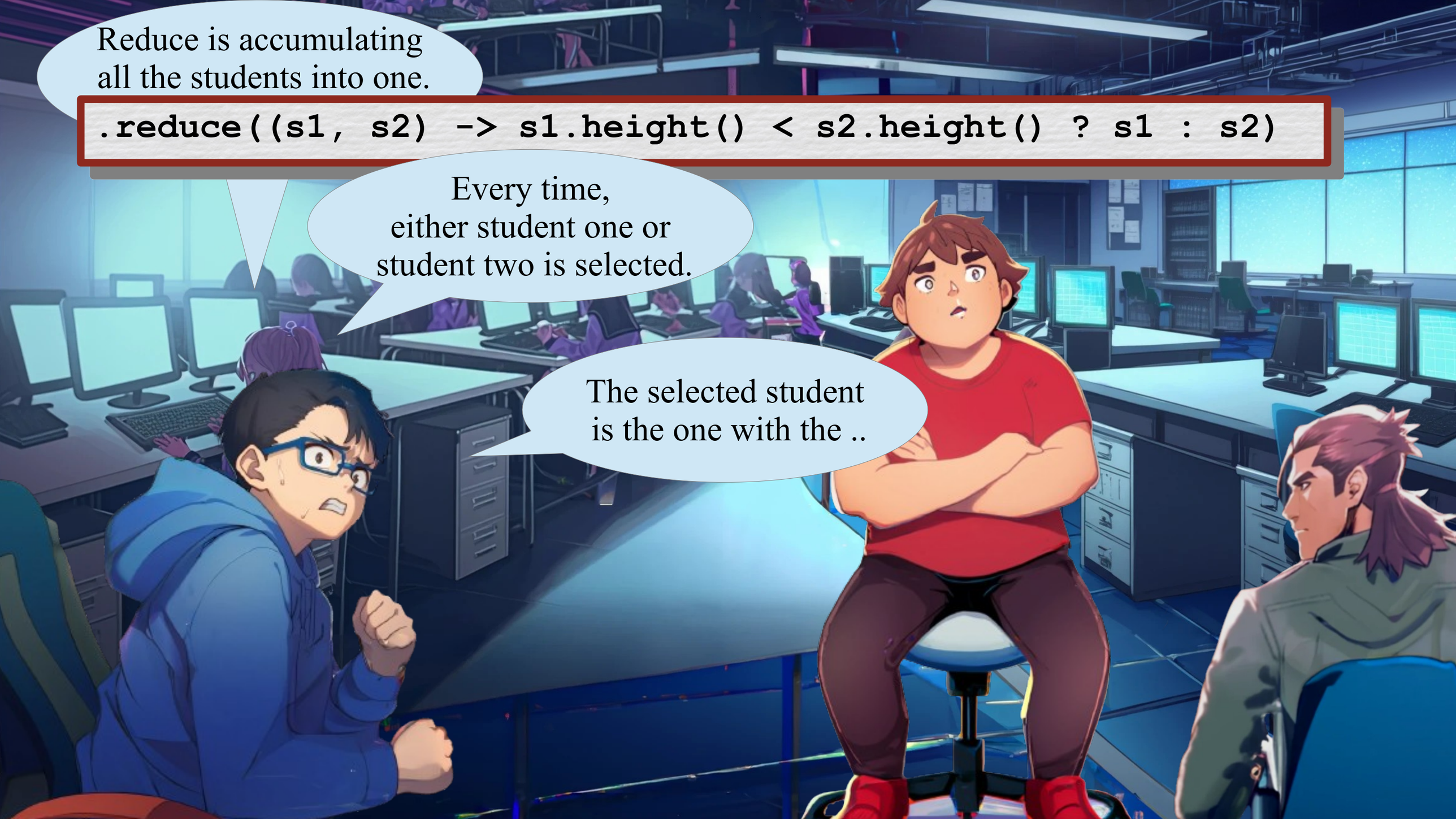
```
.forEach(s -> System.out.println(s.name()))
```

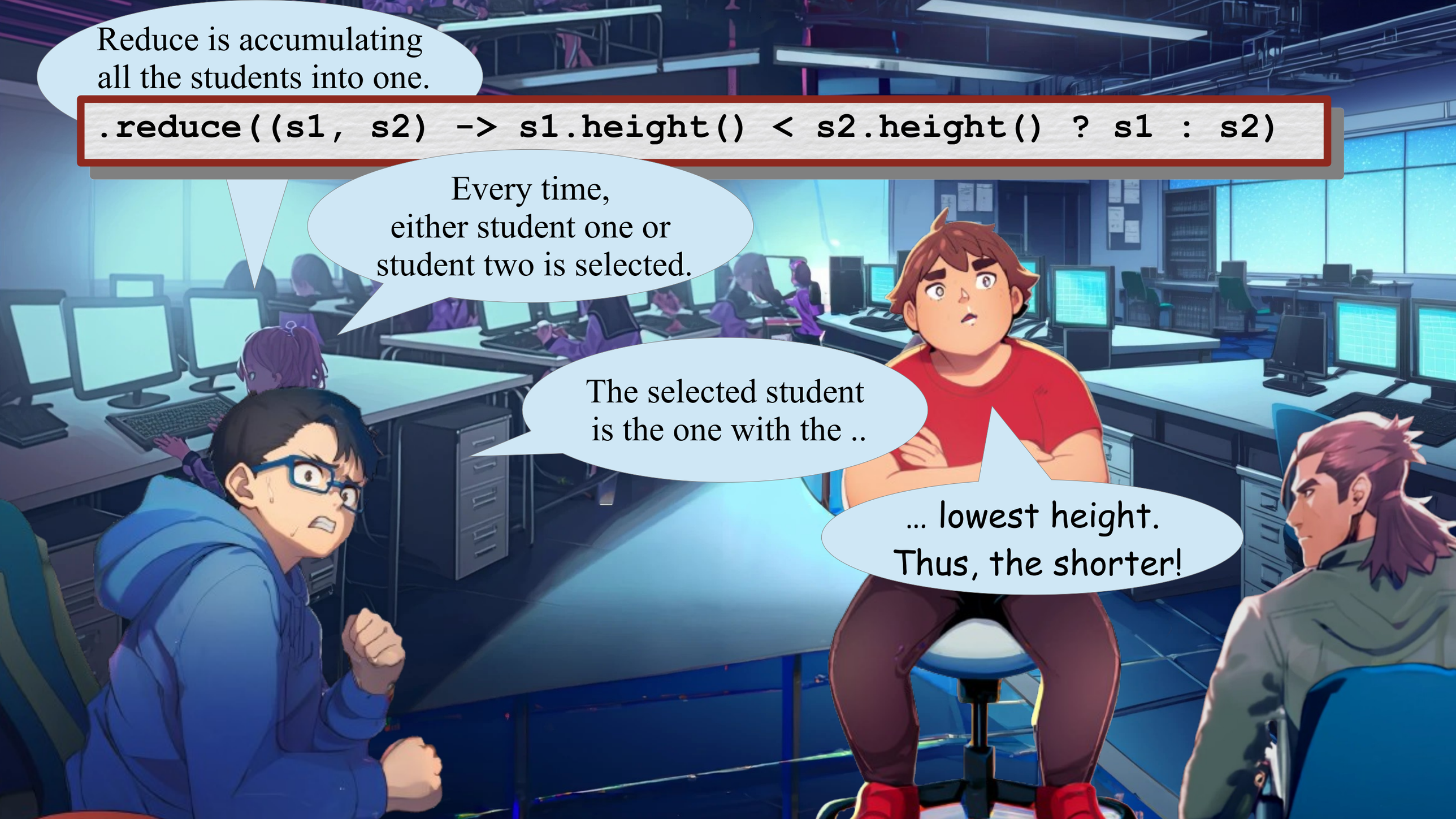
max or reduce

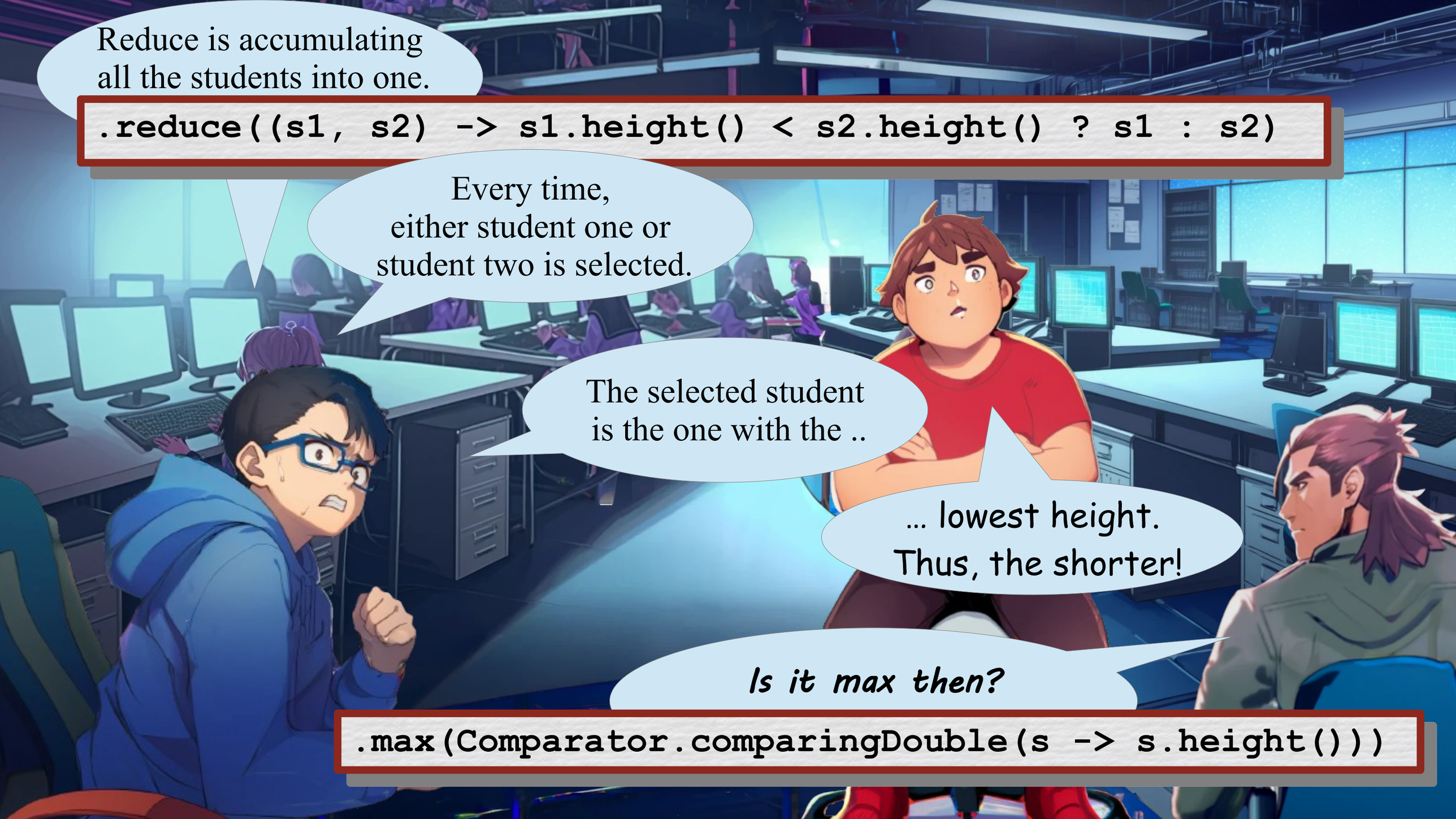
```
.orElseThrow(()->new Error( list provide
```

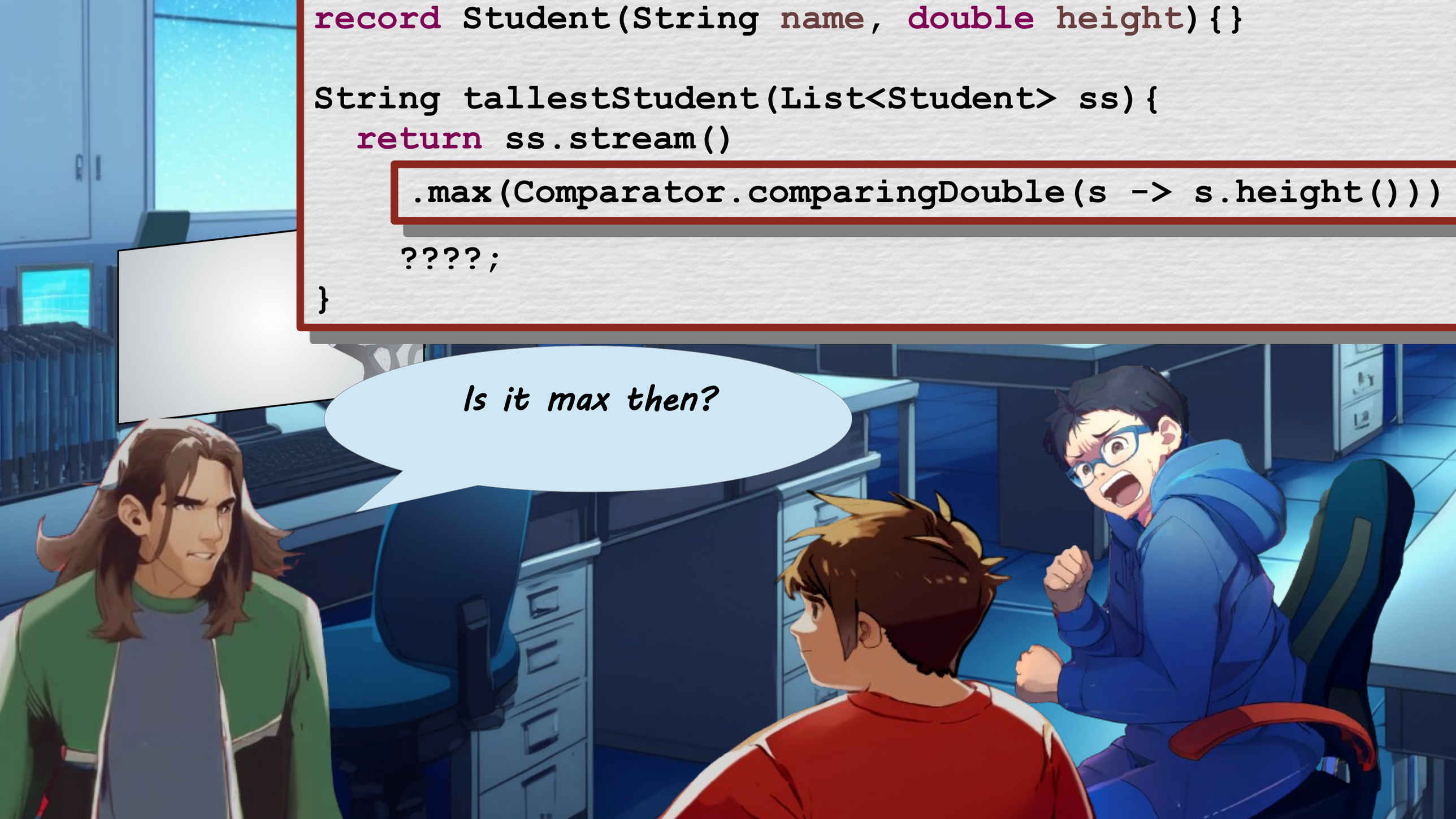

Reduce is accumulating all the students into one.
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)

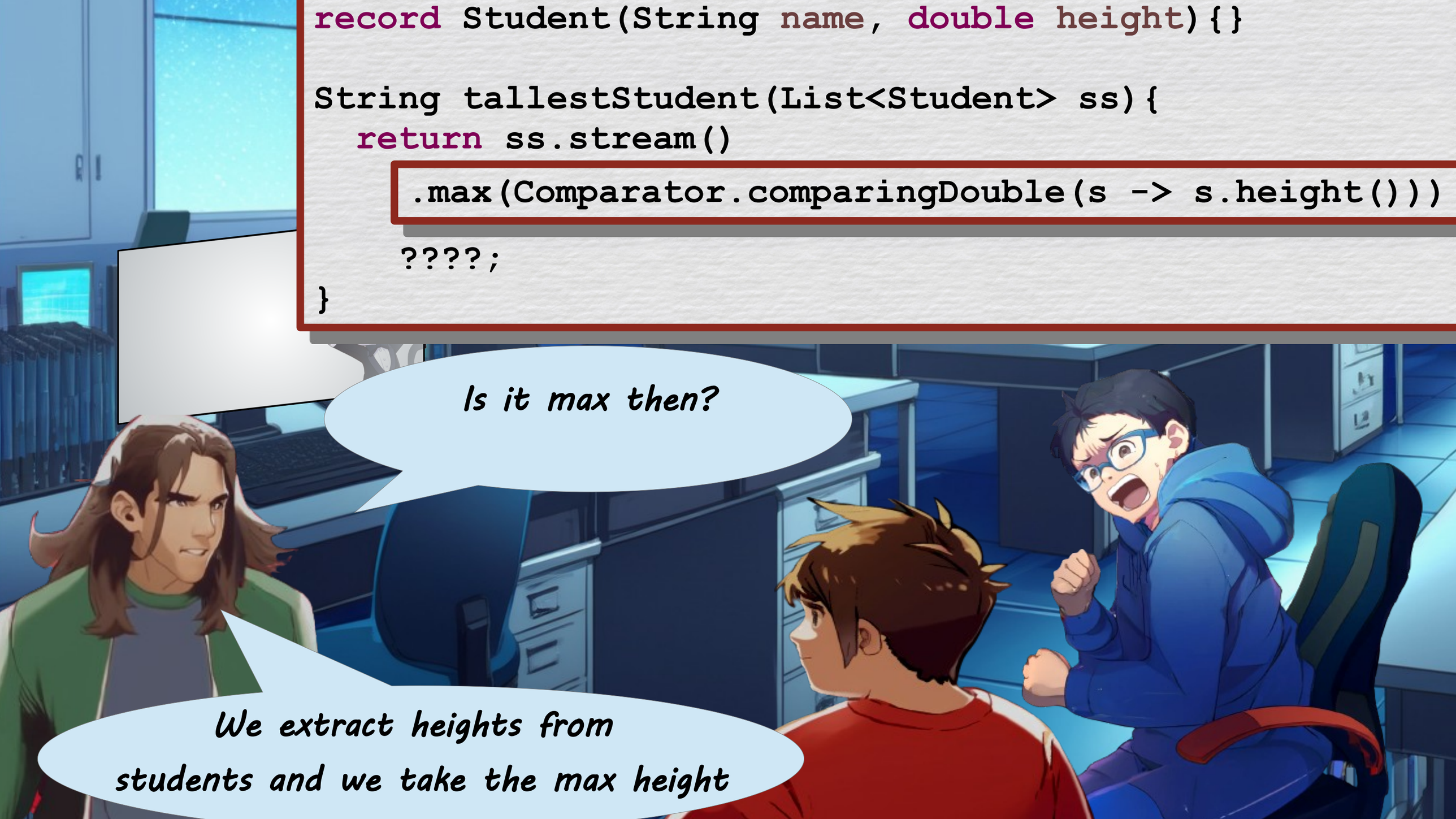

Reduce is accumulating all the students into one.
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
Every time, either student one or student two is selected.


Reduce is accumulating all the students into one.
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
Every time, either student one or student two is selected.
The selected student is the one with the ..


Reduce is accumulating all the students into one.
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
Every time, either student one or student two is selected.
The selected student is the one with the ..
... lowest height. Thus, the shorter!


Reduce is accumulating all the students into one.
.reduce((s1, s2) -> s1.height() < s2.height() ? s1 : s2)
Every time, either student one or student two is selected.
The selected student is the one with the ..
... lowest height. Thus, the shorter!
Is it max then?
.max(Comparator.comparingDouble(s -> s.height()))


record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream()
.max(Comparator.comparingDouble(s -> s.height()))
????;
}
Is it max then?


record Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream()
.max(Comparator.comparingDouble(s -> s.height()))
????;
}
Is it max then?
We extract heights from
students and we take the max height

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
}
```

.get()

.forEach(s -> System.out.println(s.name()))

.map(s -> s.name())

.findFirst()

.orElseThrow(()->new Error("Empty student list provided")

And that is it.

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
}
```

.get()

And that is it.

We have filtered out all the other students.
The only one left is the tallest; just print it out!

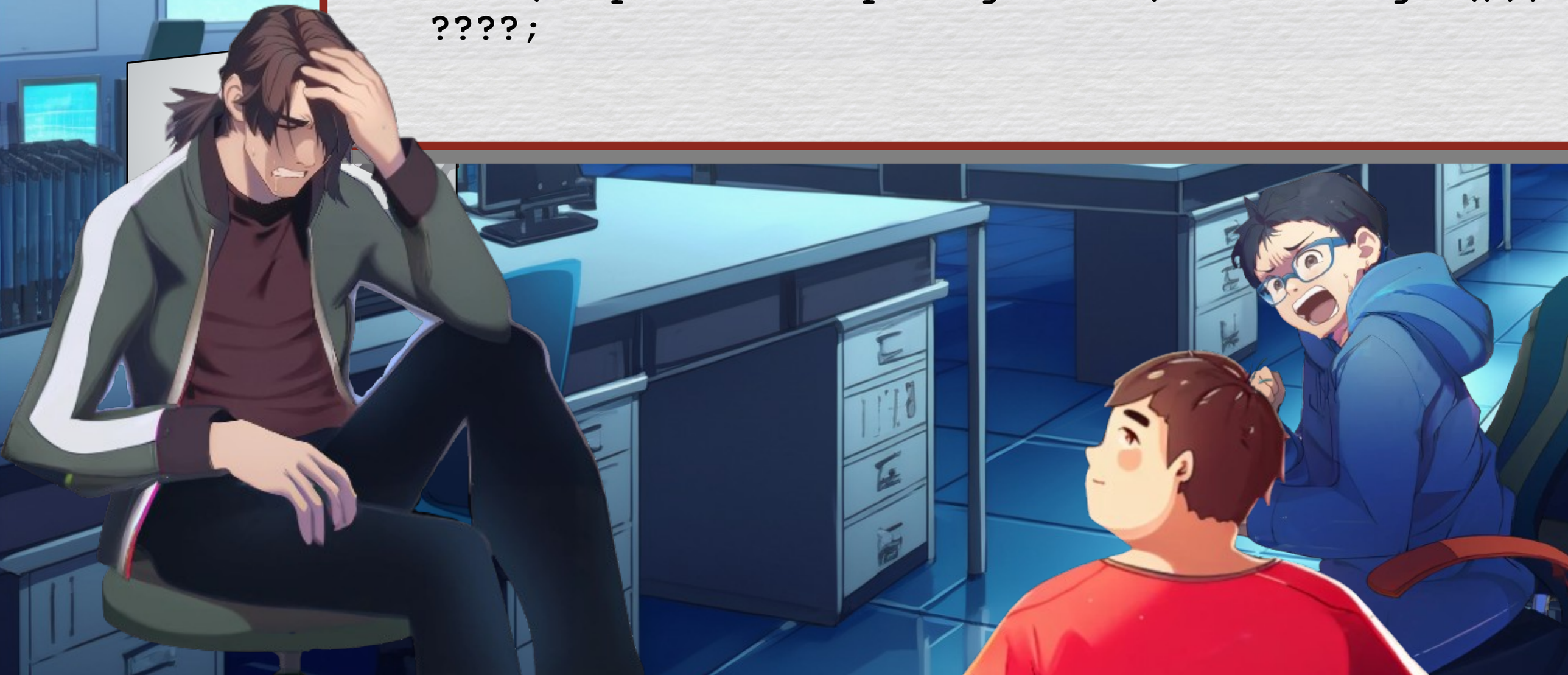
```
.forEach(s -> System.out.println(s.name()))
```

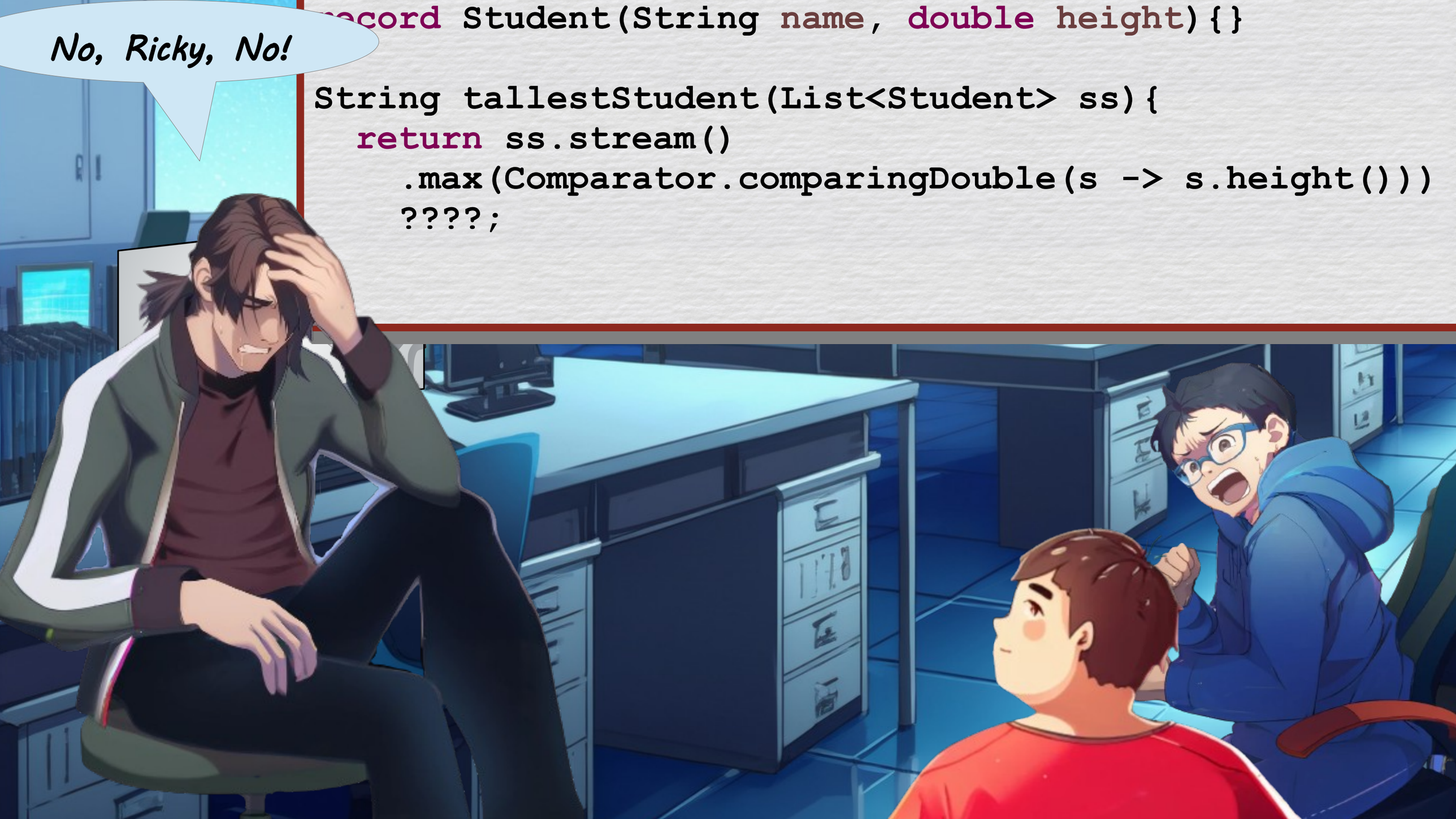
```
.map(s -> s.name())
```

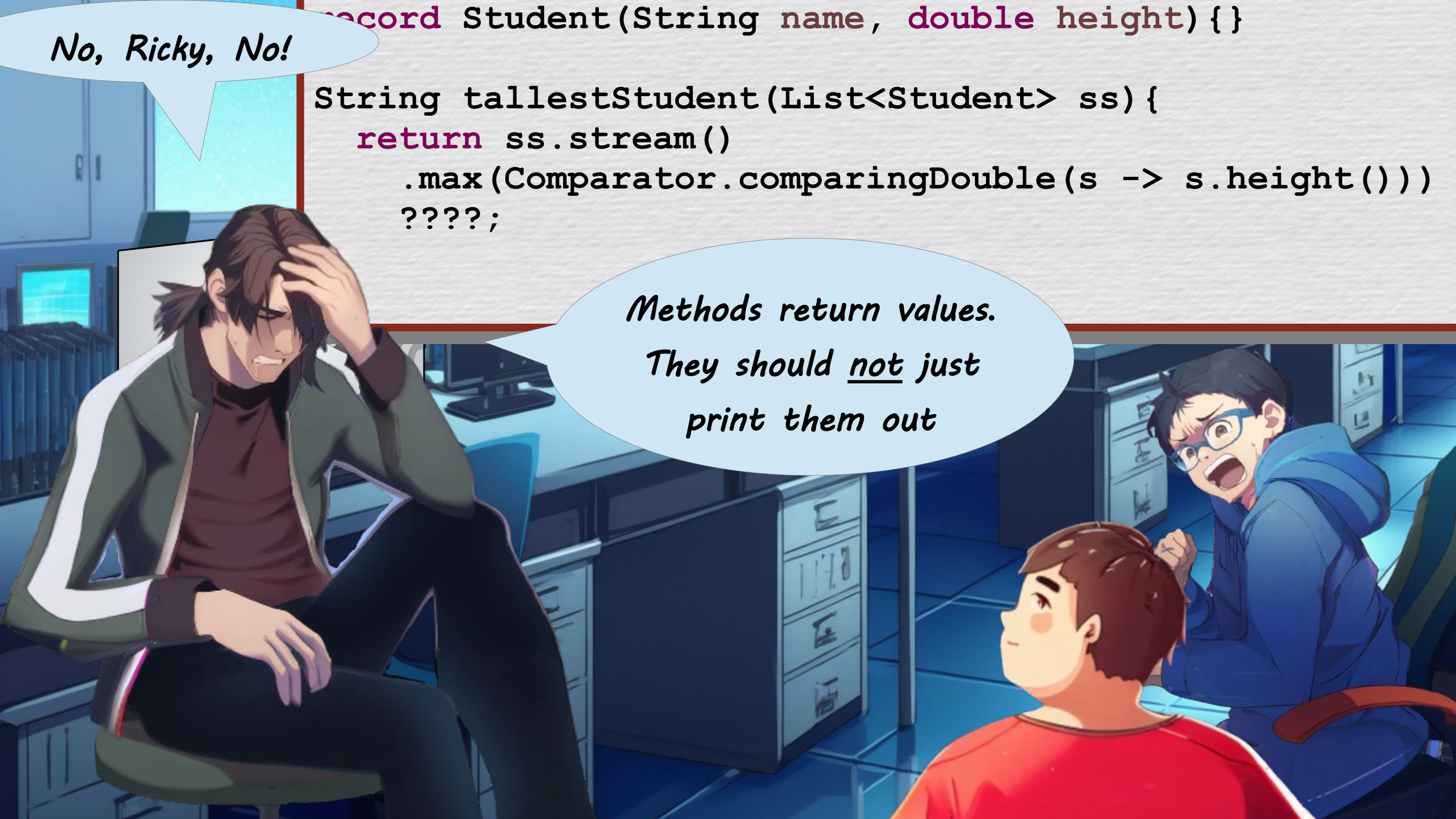
```
.findFirst()
```

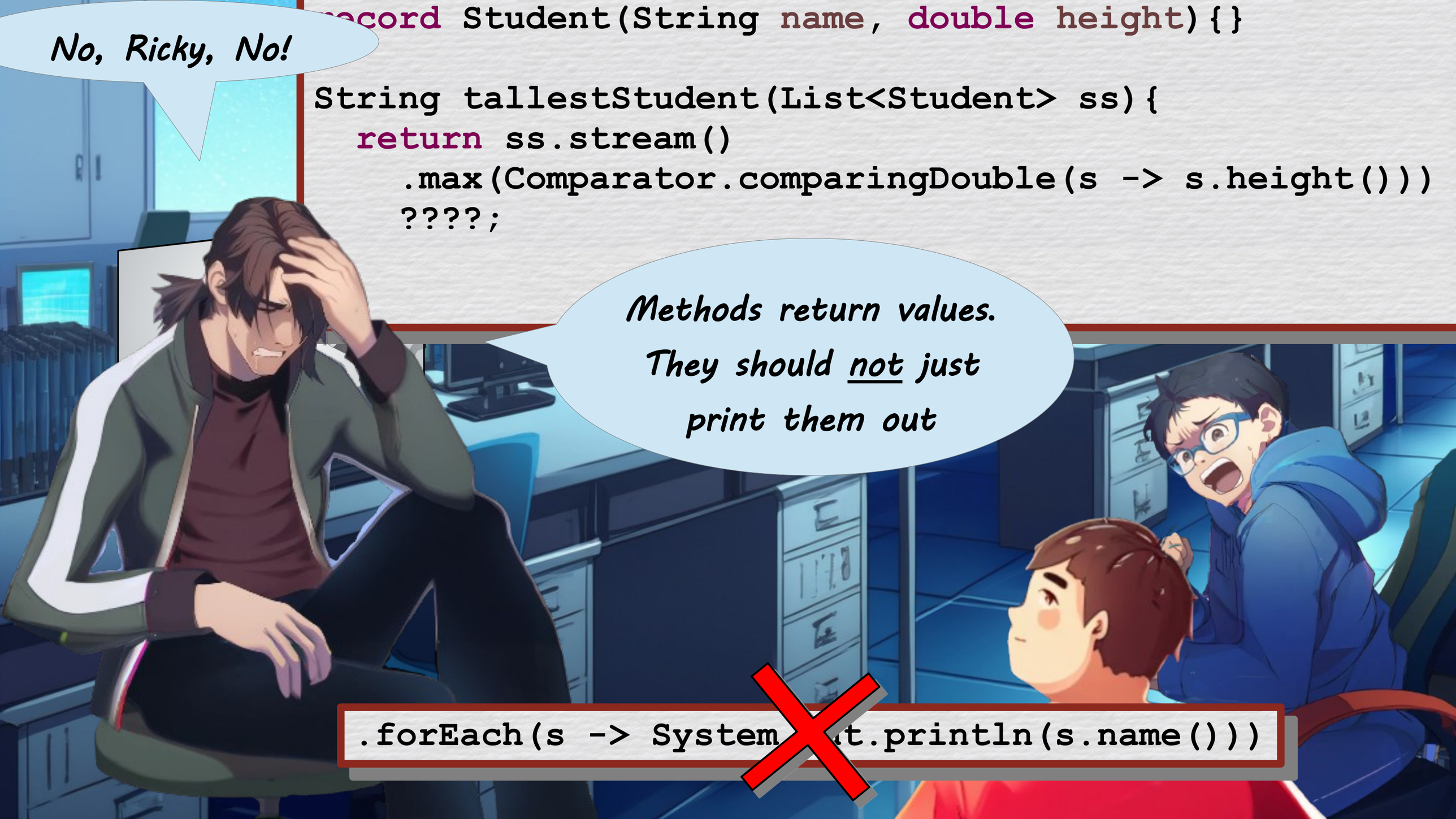
```
.orElseThrow(()->new Error("Empty student list provided");
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
```




No, Ricky, No!
ecord Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream()
.max(Comparator.comparingDouble(s -> s.height()))
????;


No, Ricky, No!
ecord Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream()
.max(Comparator.comparingDouble(s -> s.height()))
????;
Methods return values.
They should not just
print them out

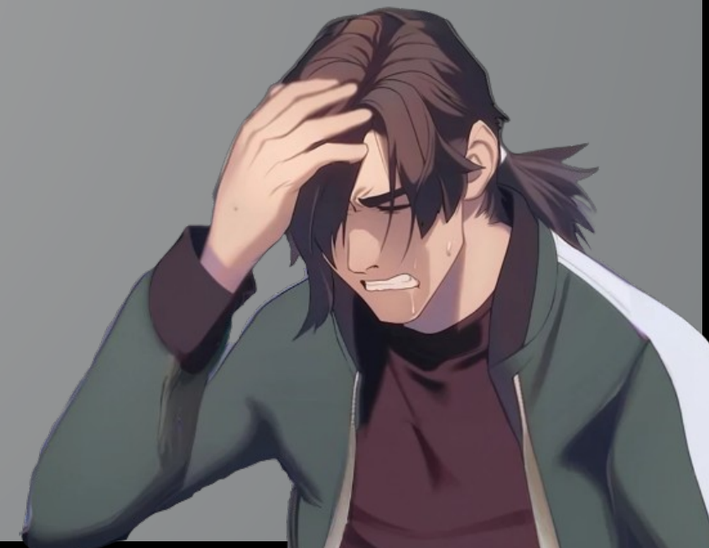

No, Ricky, No!
ecord Student(String name, double height){}
String tallestStudent(List<Student> ss){
return ss.stream()
.max(Comparator.comparingDouble(s -> s.height()))
????;
Methods return values.
They should not just
print them out
.forEach(s -> System.out.println(s.name()))

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
}
```

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
.findFirst()
```



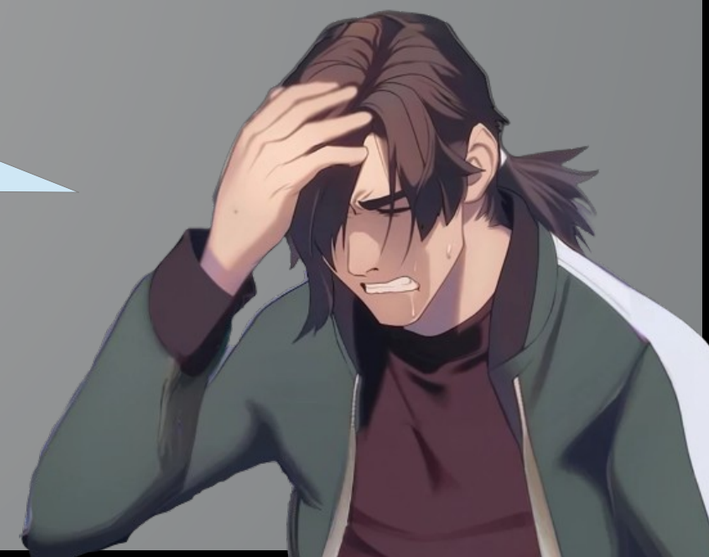
```java
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
}
```

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
.findFirst()
```

Since the result of the method is String, we have to return just the student name



```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      ????;
}
```

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
.findFirst()
```

Since the result of the method is String, we have to return just the student name

I think we can use map for that!

```
.map(s -> s.name())
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
```

As CAT said,
let the types guide you.

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
```

As CAT said,
let the types guide you.

```
.orElse   row(()->                    pty student list provided"))

.get()
```

By using map we go from
a Stream of Persons to
a Stream of Strings!

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
```

As CAT said,
let the types guide you.

```
.orElse    row(()->                    pty student list provided"))
```

```
.get()
```

By using map we go from
a Stream of Persons to
a Stream of Strings!

We can then use findFirst
to get the max element out!

```
.findFirst()
```

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
```

As CAT said,
let the types guide you.

```
.orElse    row(()->                    pty student list provided"))
```

```
.get()
```

By using map we go from
a Stream of Persons to
a Stream of Strings!

We can then use findFirst
to get the max element out!

```
.findFirst()
```

Resolved!


map, findFirst
and it is done!


map, findFirst
and it is done!
Wait a moment,
this doesn't
seem right to me.


map, findFirst
and it is done!
Wait a moment,
this doesn't
seem right to me.
We haven't examined
all the options

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
}
```

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
.findFirst()
```

I'm not sure how orElseThrow works, but it's hinting us to think

```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      ????;
}
```

```
.orElseThrow(()->new Error("Empty student list provided"))
```

```
.get()
```

```
.findFirst()
```

I'm not sure how orElseThrow works, but it's hinting us to think

What happens for the empty list?


Do you remember?
In the lecture on lists


Do you remember?
In the lecture on lists
Pupon made us chant ...


Do you remember?
In the lecture on lists
Pupon made us chant ...
THE EMPTY LIST
IS A LIST


Do you remember?
In the lecture on lists
Pupon made us chant ...
THE EMPTY LIST
IS A LIST
THE EMPTY LIST
IS A LIST


Do you remember?
In the lecture on lists
Pupon made us chant ...
THE EMPTY LIST
IS A LIST
THE EMPTY LIST
IS A LIST
THE EMPTY LIST
IS A LIST


Do you remember?
In the lecture on lists
Pupon made us chant ...
Have you really been chanting during lectures?
THE EMPTY LIST IS A LIST
THE EMPTY LIST IS A LIST
THE EMPTY LIST IS A LIST


So, findFirst can not possibly return just the string.


So, findFirst can not possibly return just the string.
So, what is it returning?


So, findFirst can not possibly return just the string.
So, what is it returning?
We should just check the documentation!


So... according to the documentation...
.max(..) : Optional<T>
.findFirst(): Optional<T>


So... according to the documentation...
.max(..) : Optional<T>
.findFirst(): Optional<T>
both findFirst and max return an Optional of T


So... according to the documentation...
.max(..) : Optional<T>
.findFirst(): Optional<T>
both findFirst and max return an Optional of T
And what is that?


So... according to the documentation...
.max(..) : Optional<T>
.findFirst(): Optional<T>
both findFirst and max return an Optional of T
And what is that?
Does it matters? Just look to what methods it has


Interesting!
.get() : T
.orElseThrow(..) : T
.map(Function<A,B> f): Optional<B>

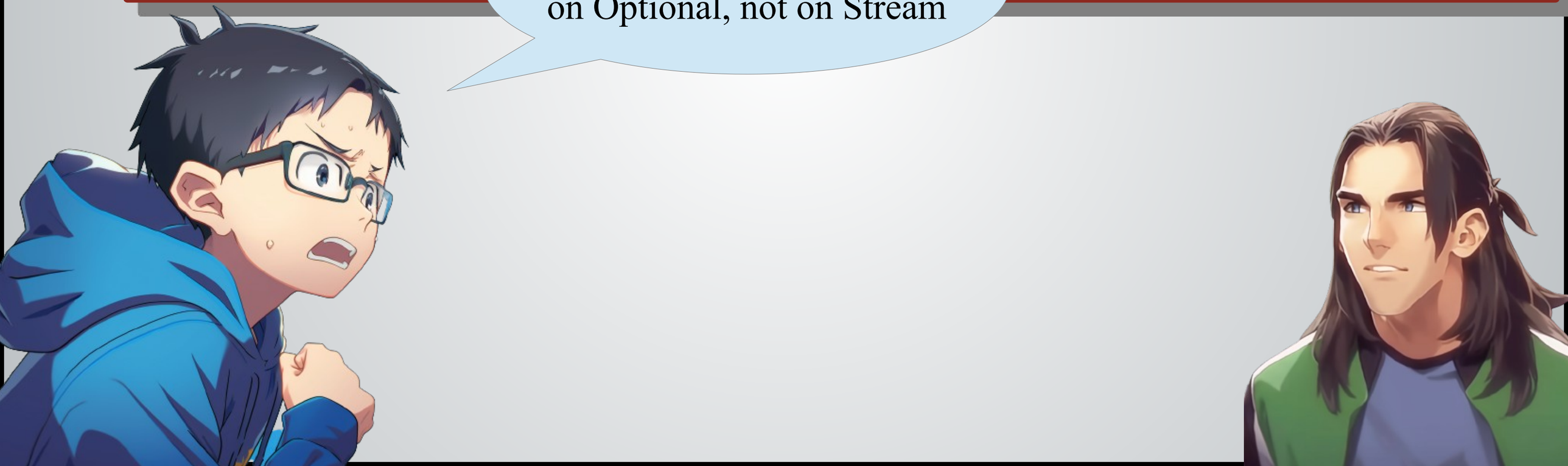

Interesting!
Optional has both get and orElseThrow
.get() : T
.orElseThrow(..) : T
.map(Function<A,B> f): Optional<B>


Interesting!
Optional has both get and orElseThrow
But, surprisingly, it also has map!
.get() : T
.orElseThrow(..) : T
.map(Function<A,B> f): Optional<B>


Interesting!
Optional has both get and orElseThrow
But, surprisingly, it also has map!
But, map was a stream thing!
.get() : T
.orElseThrow(..) : T
.map(Function<A,B> f): Optional<B>


Interesting!
Optional has both get and orElseThrow
But, surprisingly, it also has map!
.get() : T
.orElseThrow(..) : T
.map(Function<A,B> f): Optional<B>
But, map was a stream thing!
Nothing stops two classes from declaring methods with the same name.

```java
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
  return ss.stream()
     .max(Comparator.comparingDouble(s -> s.height()))
     .map(s -> s.name())
     .orElseThrow(()->new Error("Empty student list provided"));
}
```



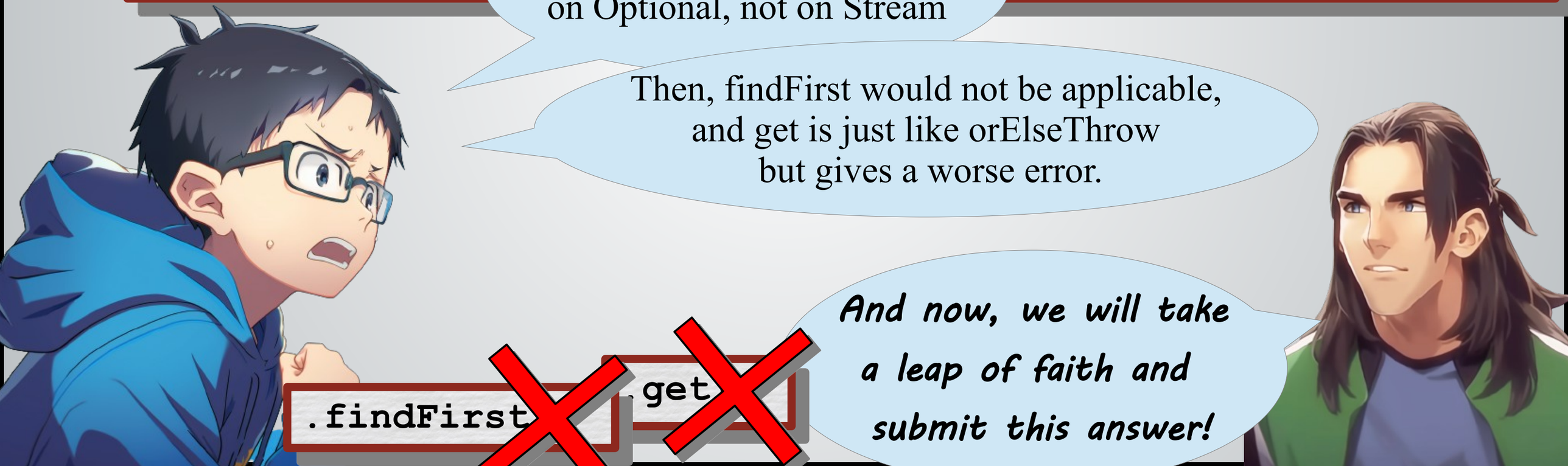
```
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
  return ss.stream()
     .max(Comparator.comparingDouble(s -> s.height()))
     .map(s -> s.name())
     .orElseThrow(()->new Error("Empty student list provided"));
}
```

So, this map call would be on Optional, not on Stream

Then, findFirst would not be applicable, and get is just like orElseThrow but gives a worse error.

~~.findFirst~~ ~~.get~~

```java
record Student(String name, double height){}

String tallestStudent(List<Student> ss){
  return ss.stream()
     .max(Comparator.comparingDouble(s -> s.height()))
     .map(s -> s.name())
     .orElseThrow(()->new Error("Empty student list provided"));
}
```



The trio submit their answer!

```
String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      .orElseThrow(()->new Error("Empty student list provided"));
}
```

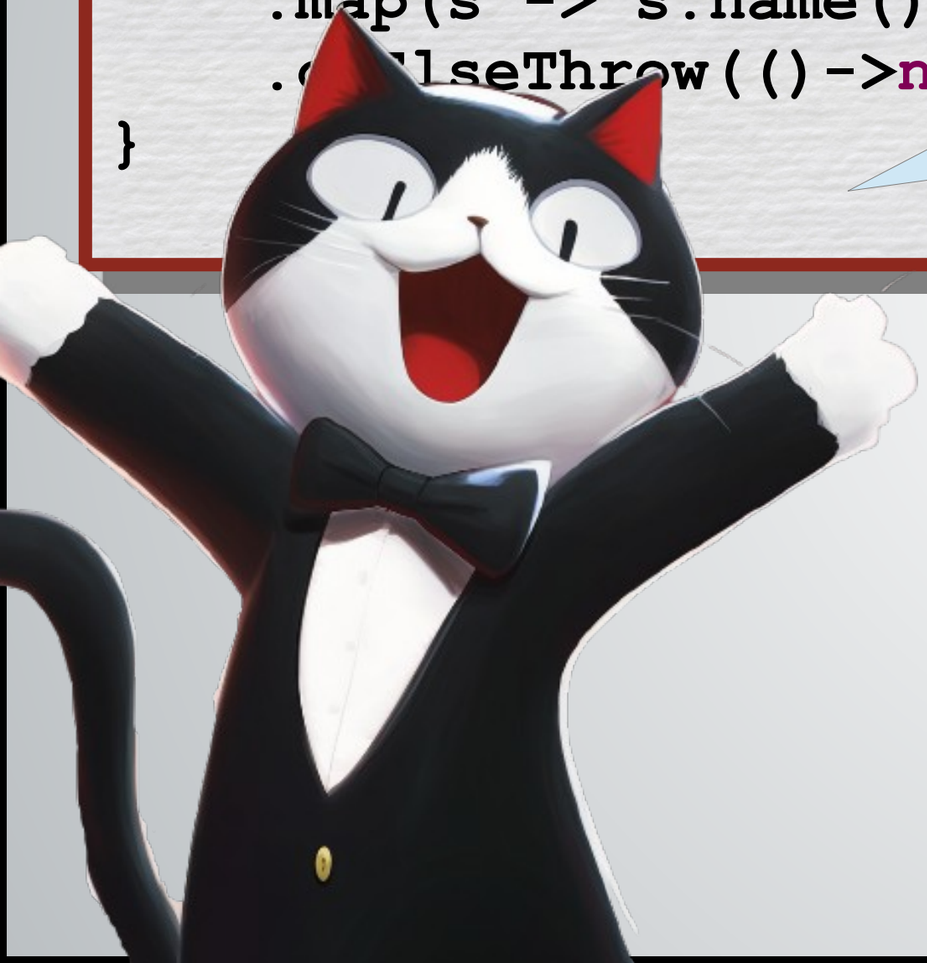
Processing answer...

Tik tik Trwoooowww Waaaakkkk

The trio submit their answer!

```
String tallestStudent(List<S
   return ss.stream()
      .max(Comparator.compa
      .map(s -> s.name())
      .o lseThrow(()->new E
}
```

Good news!
The answer is correct!



The trio submit their answer!

```
String tallestStudent(List<Student> ss){
   return ss.stream()
      .max(Comparator.comparingDouble(s -> s.height()))
      .map(s -> s.name())
      .orElseThrow(()->new Error("Empty student list provided"));
}
```

Bad news; you took way too long. There are 9 more exercises today!


TIK TIK TIK TAK TIK


TIK TIK TIK TAK TIK
Done !


TIK TIK TIK TIK TAK TIK
Done!
Ok, let's start the last exercise


TIK TIK TIK TAK TIK

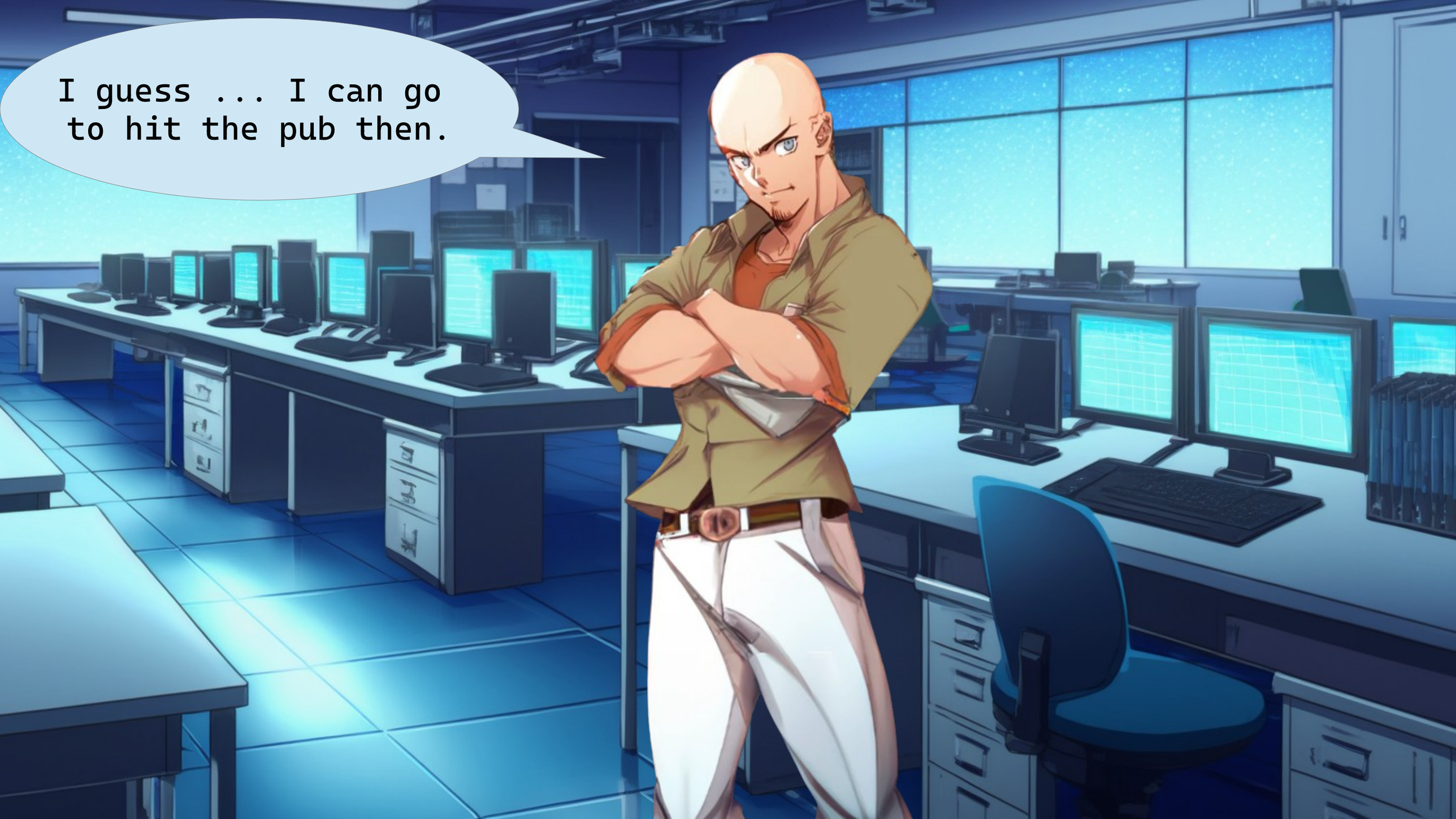

Done; that was easy, too.
TIK TIK TIK TAK TIK


Done; that was easy, too.
TIK TIK TIK TAK TIK
Great! ...
You completed it.


Done; that was easy, too.
Great! . . .
You completed it.
TIK TIK TIK TAK TIK
According to our data, you are the fastest ever recorded.


We just finished the first one and he finishes them all?


We just finished the first one and he finishes them all?
Yea, Hanton is just that kind of monster

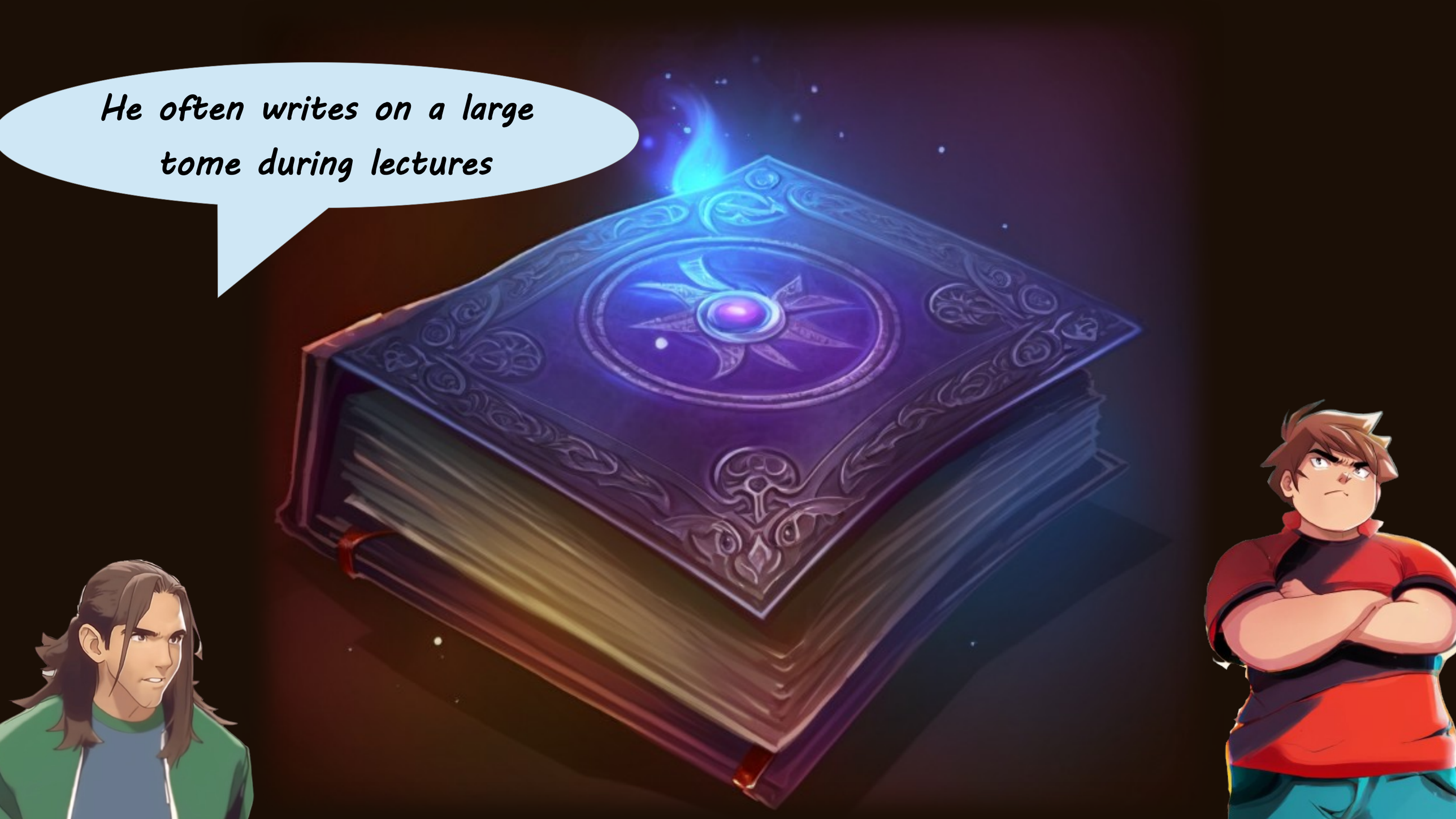

I guess ... I can go to hit the pub then.

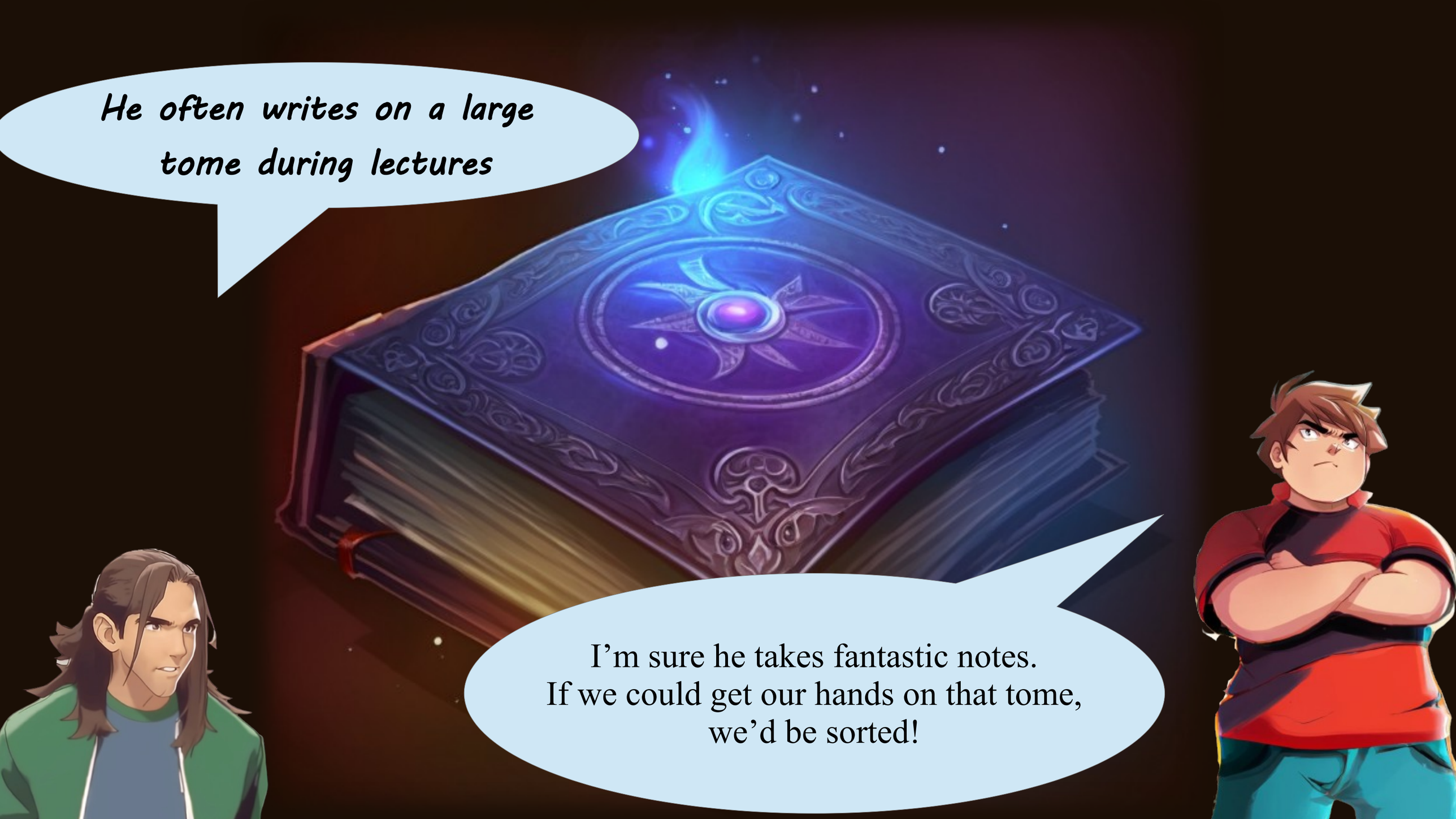

I guess ... I can go to hit the pub then.
See you later, humans!


Who is that guy?


Who is that guy?
He is Hanton.
He’s a brilliant buffoon


Who is that guy?
He is Hanton.
He’s a brilliant buffoon
He is going to be admitted in the second year; for sure!


He often writes on a large tome during lectures


He often writes on a large tome during lectures
I'm sure he takes fantastic notes.
If we could get our hands on that tome,
we'd be sorted!

CAT interrupts them

Get a move! You still have 9 exercises to go! Here is the next one!


Ok, let's keep going


Ok, let's keep going
If he just finished, it should be feasible for us too


Ok, let's keep going
If he just finished, it should be feasible for us too
Yea, it'll be over in no time!


Five hours later


Five hours later
Done!
This was the last one, right?


Five hours later
Done!
This was the last one, right?
That was just exercise 9;
now we can finally
start the last one

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

Drag and drop some of the following lines of code . . .

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

Drag and drop some of the following lines of code . . .

so that the method returns the list of the student names with grades of 80 or more

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

Drag and drop some of the following lines of code . . .

so that the method returns the list of the student names with grades of 80 or more

Their grade must also be good enough that it could place them in the best 10% of the class

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  ????
}
```

```
.toList();
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.map(s -> s.name())
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
.toList();
```

```
record Student(String name, int grade){}
```

```
.findFirst();
```

```
List<String> best(List<Student> ss){
```

```
return
```

```
.map(
```

```
.filt
```

```
int i
```

```
Optio
```

```
.filt
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

Pause the video. Can you solve this one?

Note: not all snippets are needed.

The method returns the list of the student names
with grades of 80 or more.
Their grade must also be good enough that it could place
them in the best 10% of the class.
The class is composed of all the students in the list.

This message will disappear shortly; you can pause after that.

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```


Ok, now we have 3 semicolons!


Ok, now we have 3 semicolons!
So, 3 statement?


Ok, now we have 3 semicolons!
So, 3 statement?
May be we do not have to use them all.

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.st
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

The local variable could be a good start

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   ????
}
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
int indexAt10pc = (ss.size()-1)/10;
```

```
Optional<Student> threshold = ss.st
```

```
.filter(s ->
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

The local variable could be a good start

It is the only place using 'size', so we need it for the 10% of students

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   int indexAt10pc = (ss.size()-1)/10;
   ????
}
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
.limit(indexAt10pc)
```

```
.skip(indexAt10pc)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=thresh
```

```
.sorted(Comparator.comparingIn
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
.limit(indexAt10pc)
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   int indexAt10pc = (ss.size()-1)/10;
   return ss.stream()
   ????
}
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade())
```

Then we sort our stream!

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
.limit(indexAt10pc)
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

Then, we take just 10% of the students

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

Do not forget, we also need to ensure that ...

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

Do not forget, we also need to ensure that ...

their grade is at least 80!

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

And finally, map and toList!

```
.map(s -> s.name())
```

```
.toList();
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
<Student> threshold = ss.stream()
```

```
s.grade()>=threshold.get().grade())
```

Done! I think we can submit it!

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
<Student> thres
s.gr
```

Done! I think we can submit it!

Incredible, the last one was pretty easy!

That is why Hanton finished it in a blast!


..And you get an A+!


..And you get an A+!
Oh, yea, the good old high school times!


..And you get an A+ too!
And you!


..And you get an A+ too!
And you!
Easy A+s all over the place


And you! And you!

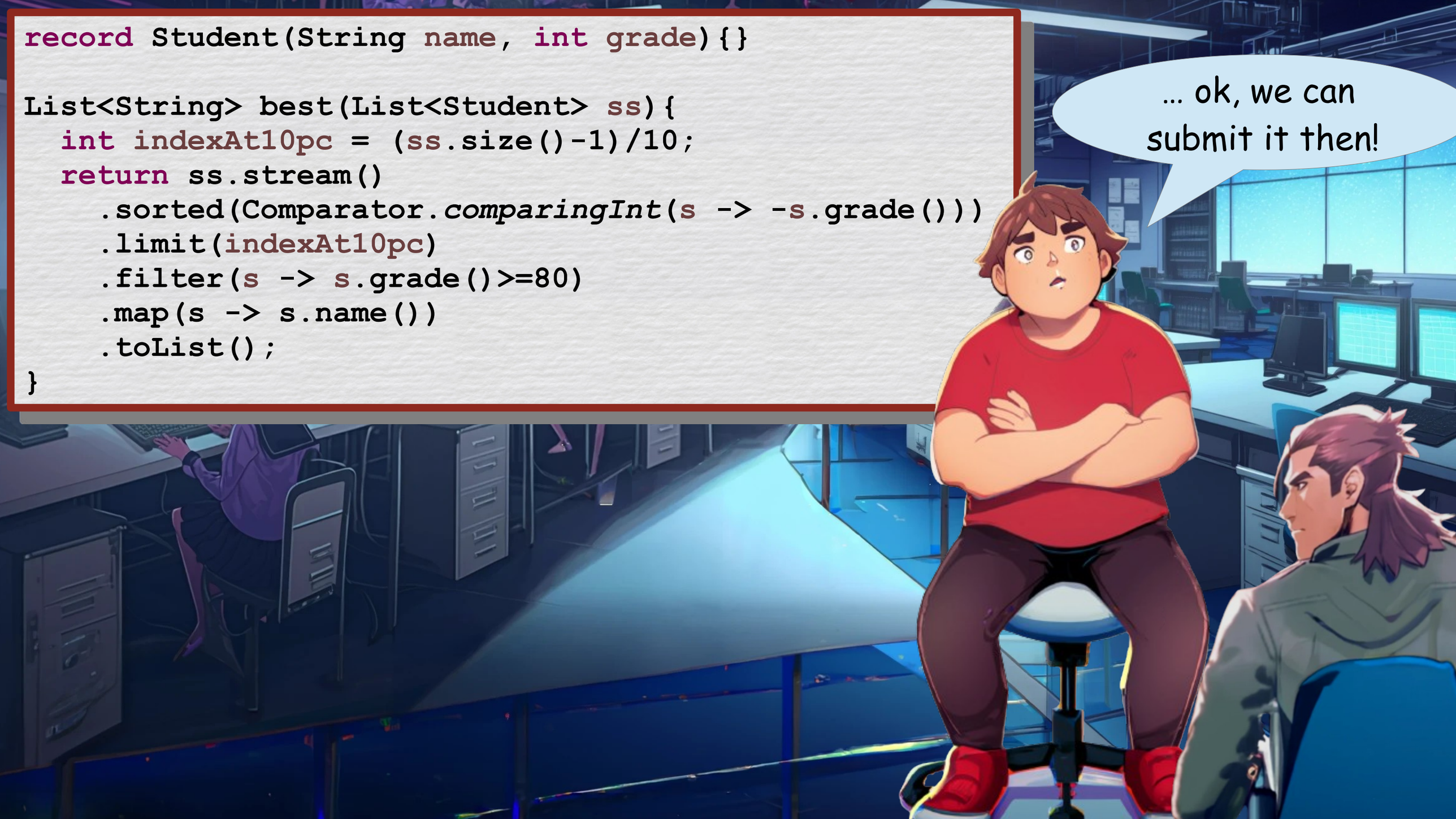

And you! And you!
Everyone gets an A+!

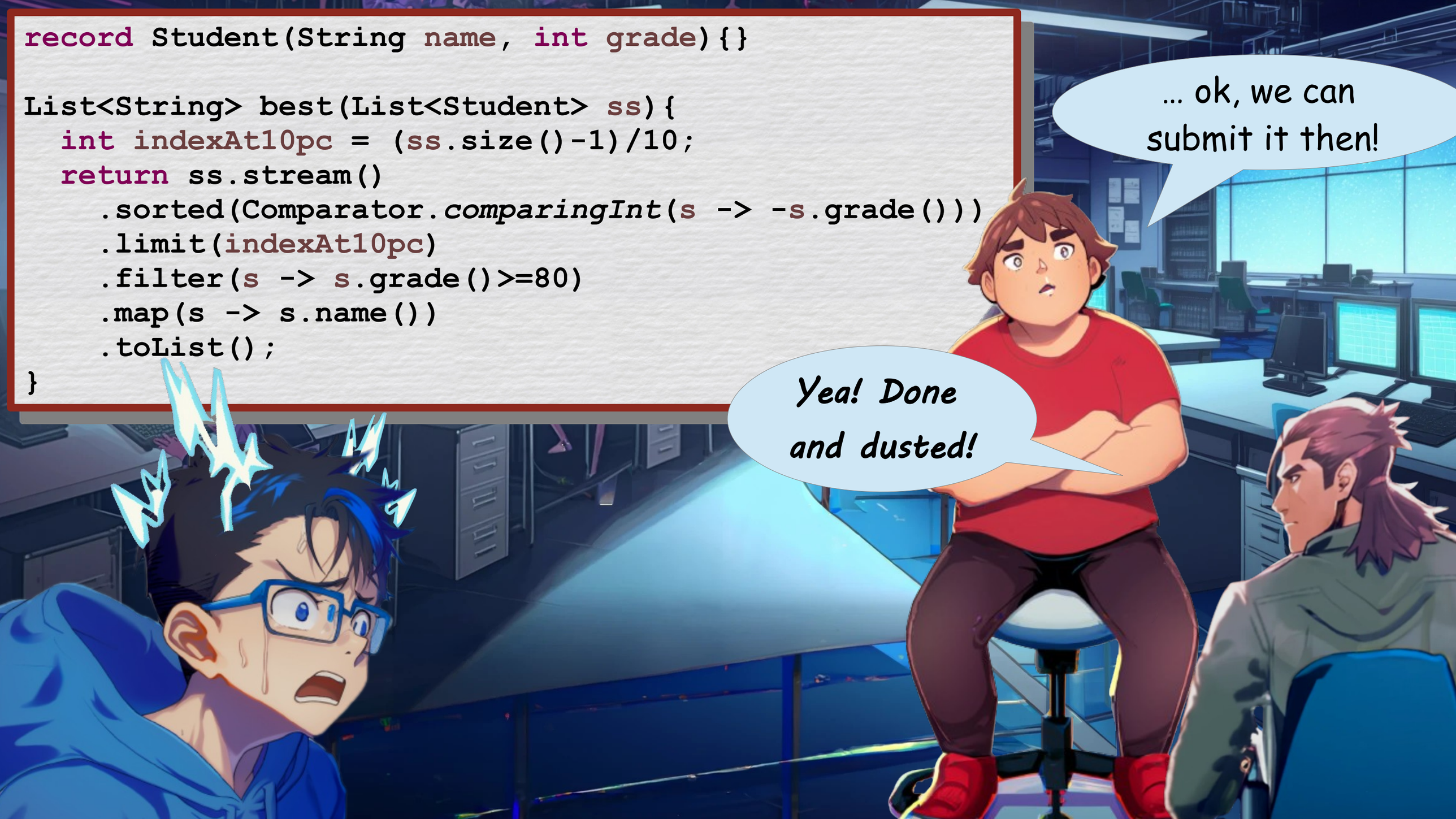

And you! And you!
Everyone gets an A+!
Wait a moment!

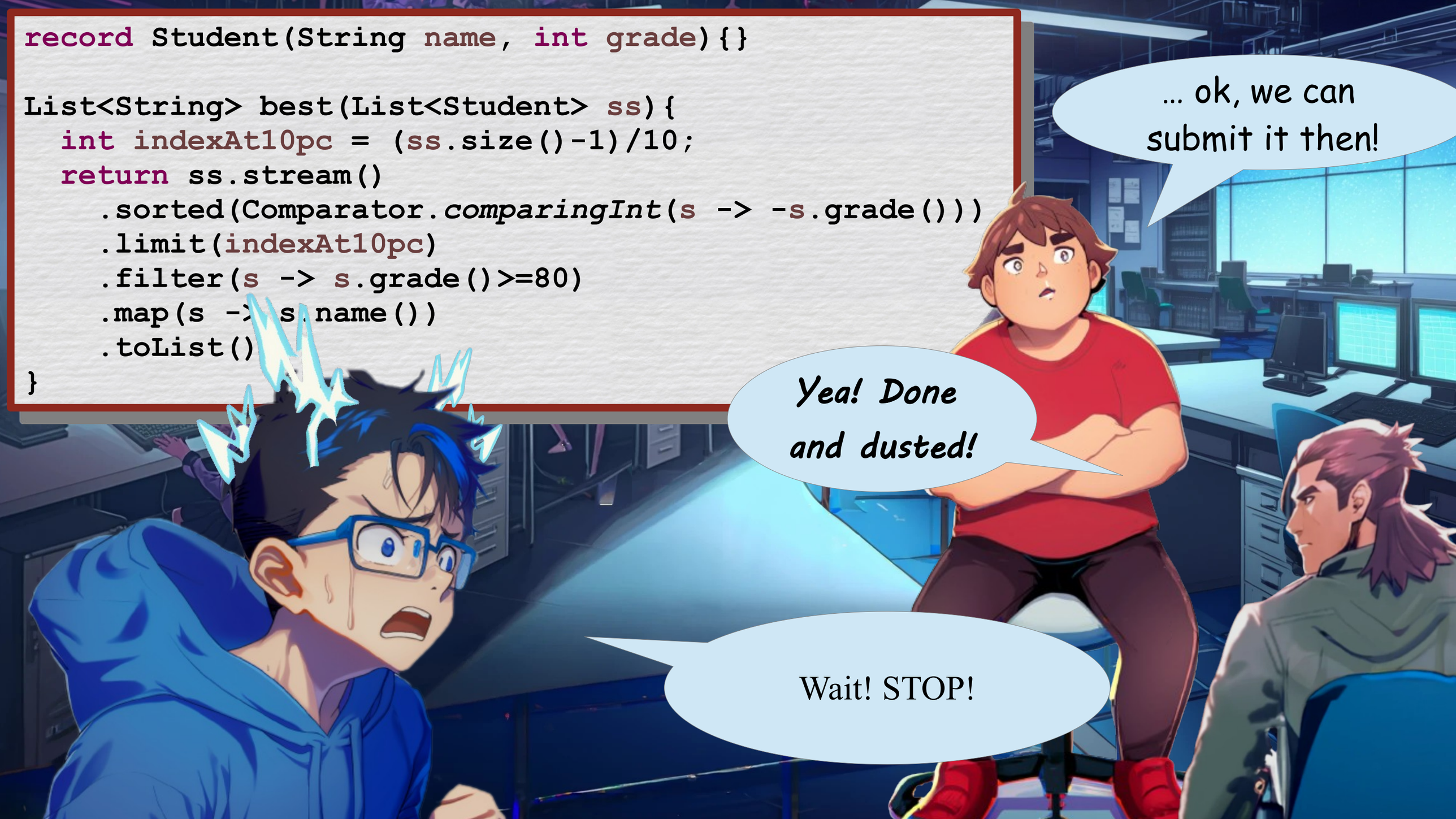

And you! And you!
Everyone gets an A+!
Wait a moment!
What if everyone
just got max marks?

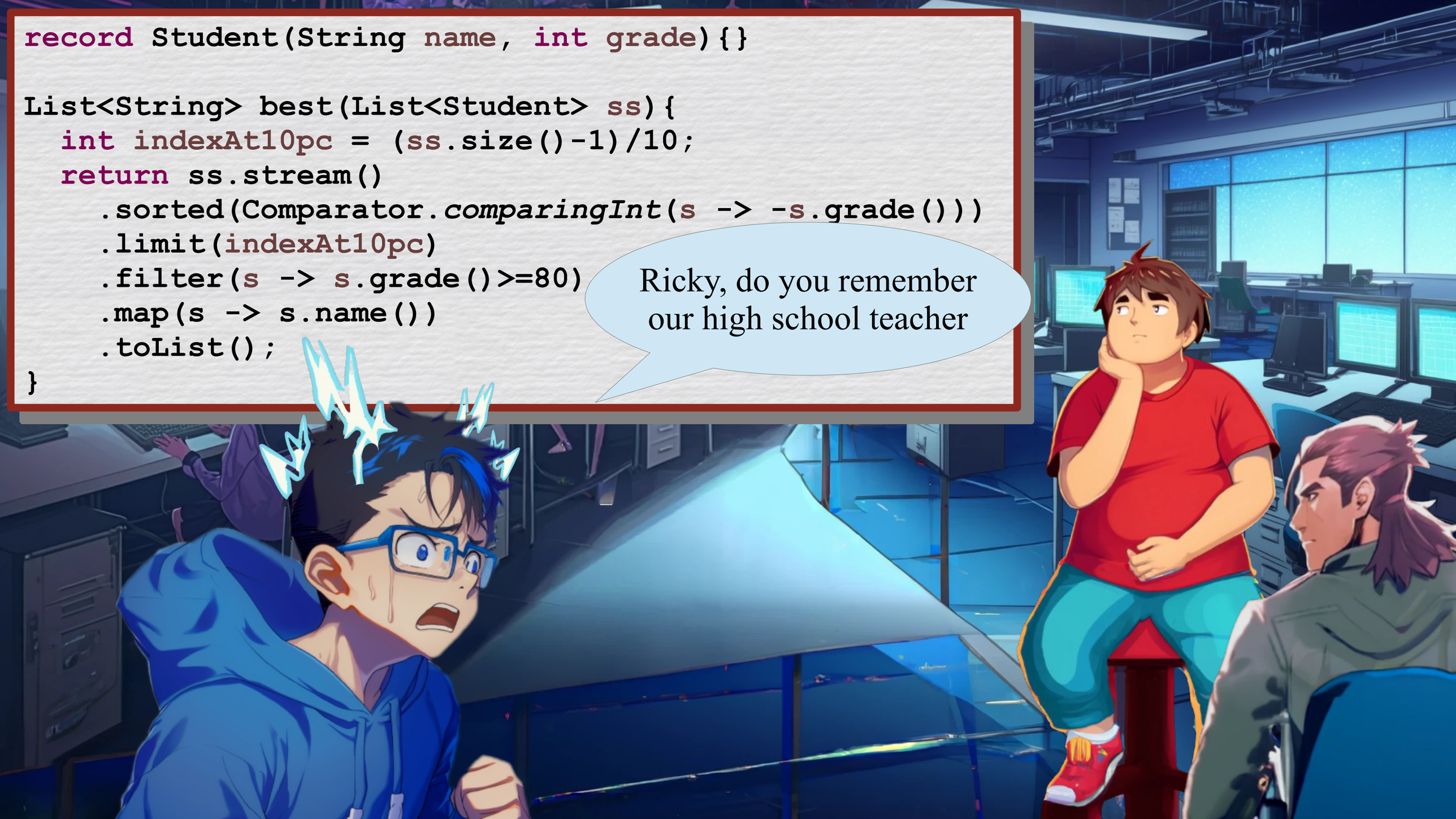

And you! And you!
Everyone gets an A+!
Wait a moment!
What if everyone just got max marks?
Their grade must also be good enough that it could place them in the best 10% of the class.

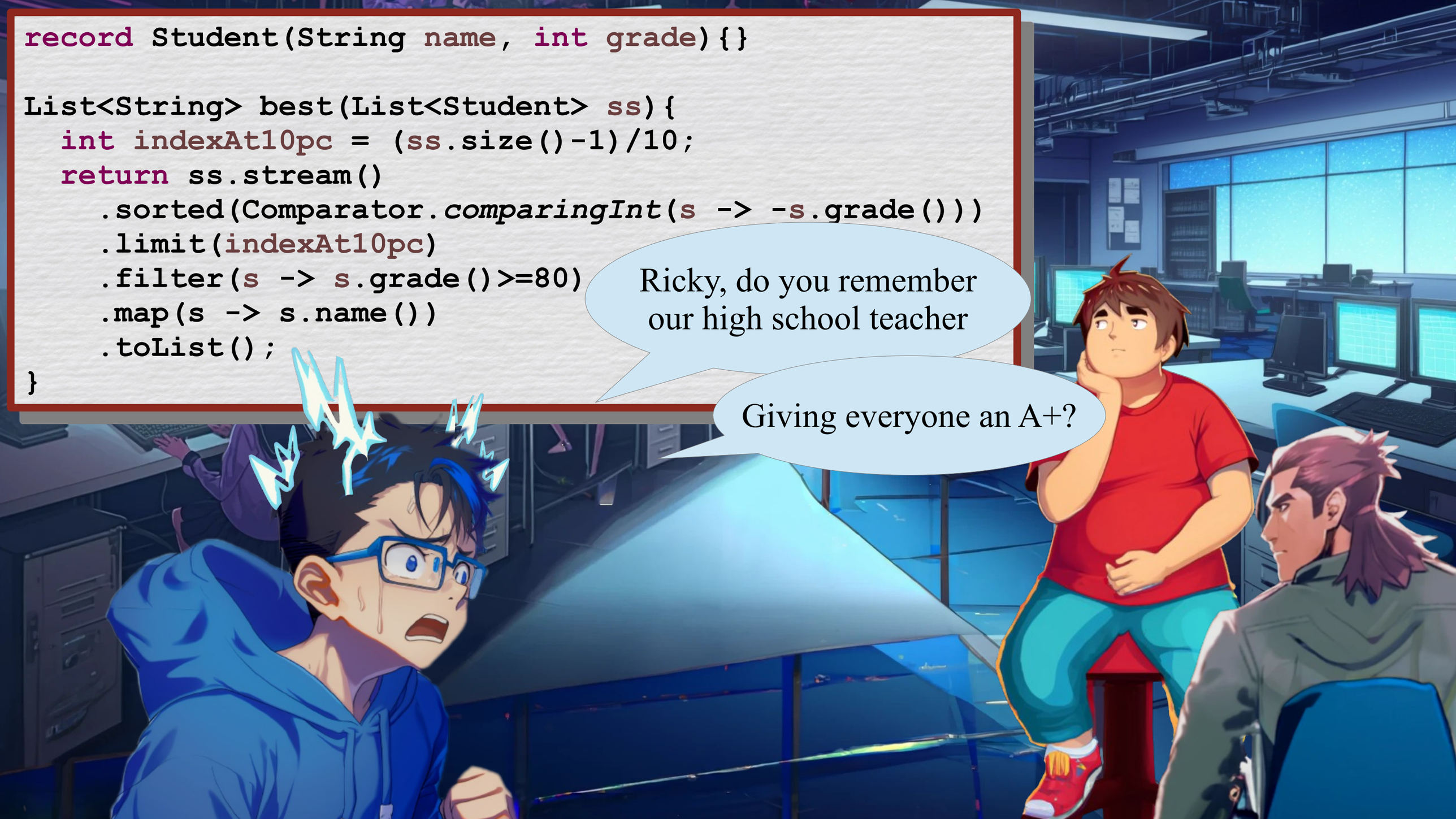
```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```

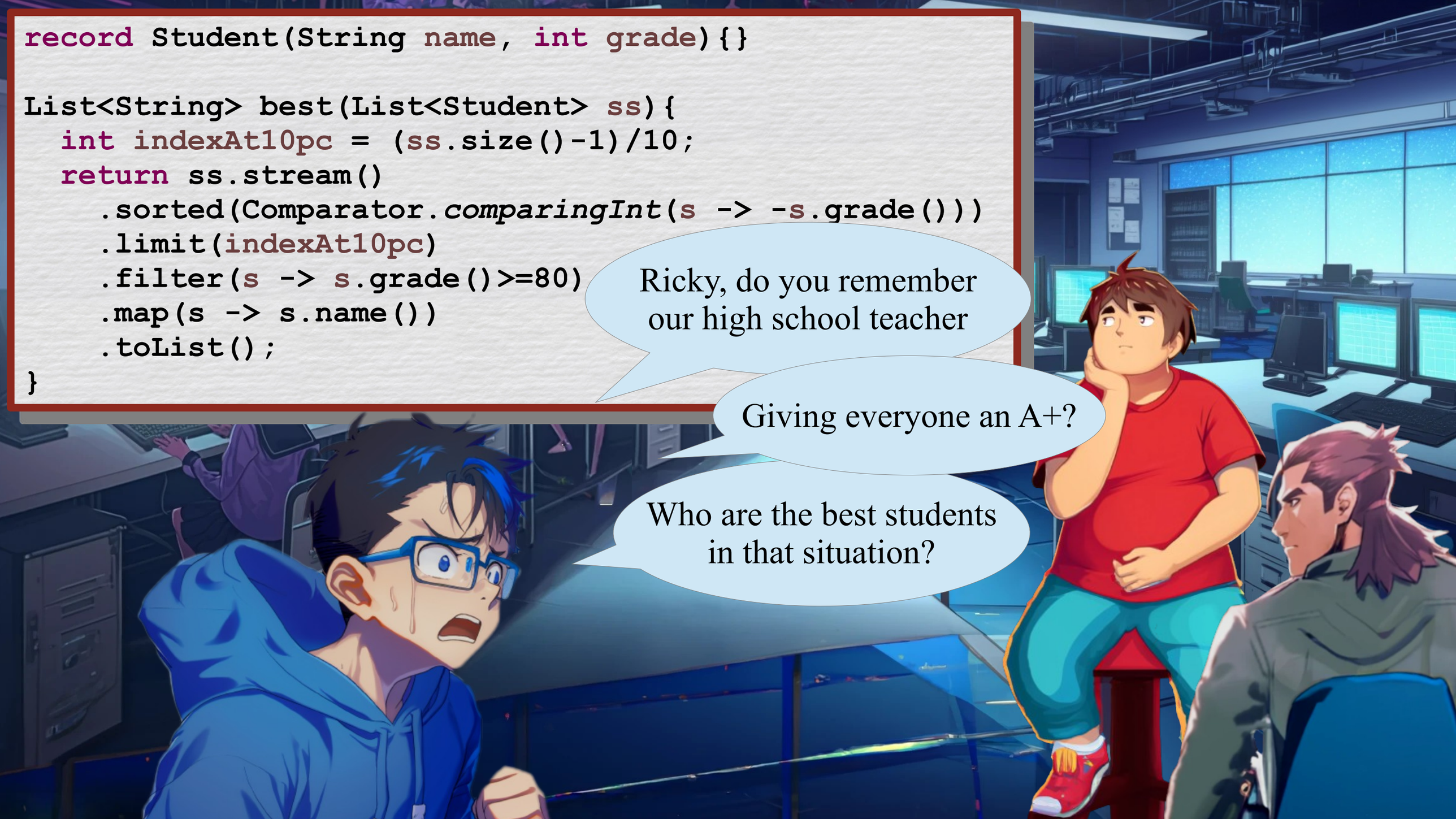


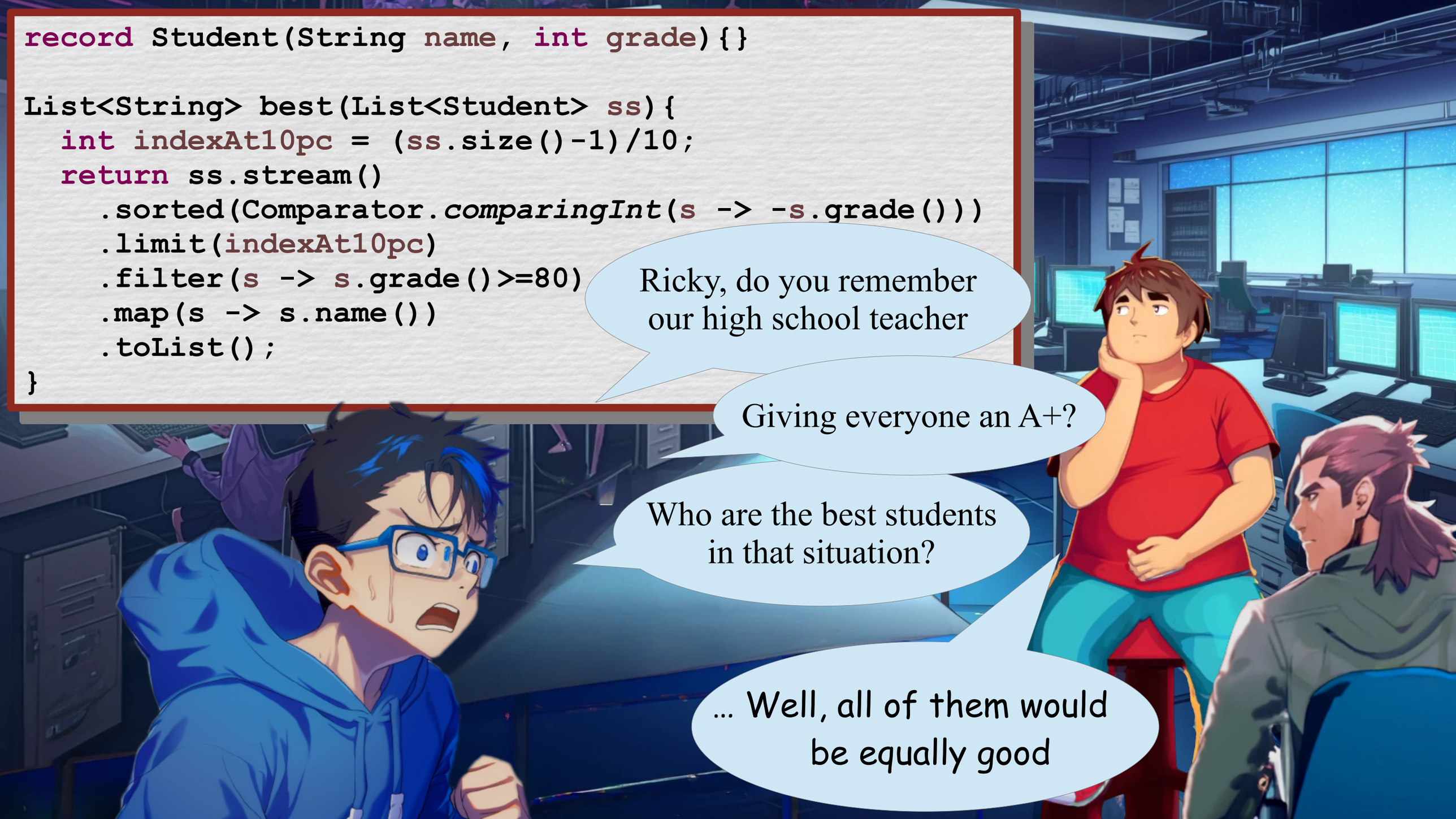

record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
… ok, we can submit it then!
Yea! Done and dusted!


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList()
}
... ok, we can submit it then!
Yea! Done and dusted!
Wait! STOP!


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Ricky, do you remember our high school teacher


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Ricky, do you remember our high school teacher
Giving everyone an A+?


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Ricky, do you remember our high school teacher
Giving everyone an A+?
Who are the best students in that situation?


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Ricky, do you remember our high school teacher
Giving everyone an A+?
Who are the best students in that situation?
... Well, all of them would be equally good


Wait... So we may have to return more then 10% of the students then!

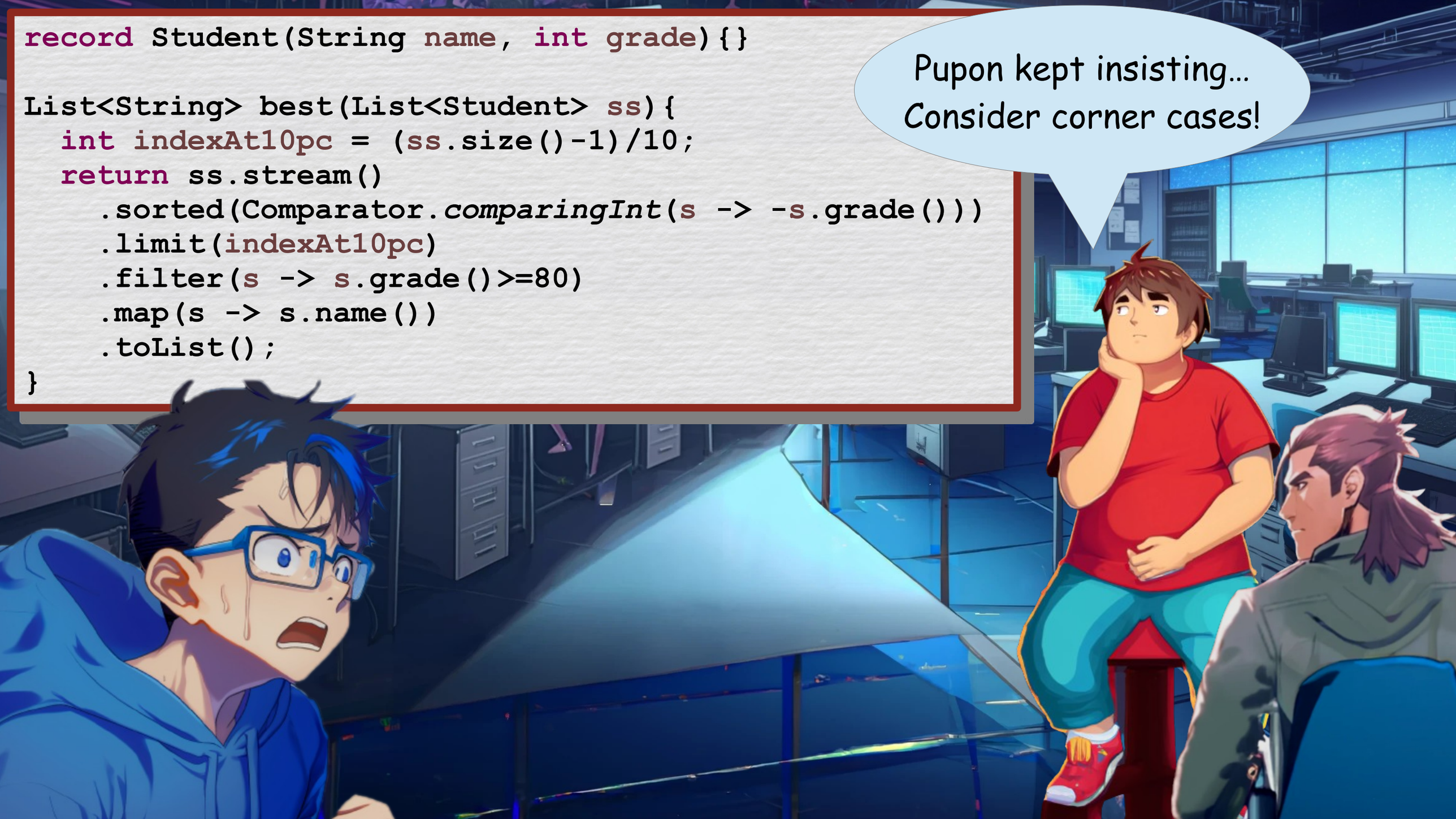

Wait... So we may have to return more then 10% of the students then!
Yes; so limit should not be used!
.limit(indexAt10pc


Wait... So we may have to return more then 10% of the students then!
Yes; so limit should not be used!
.limit(indexAt10pc
So the question actually means …

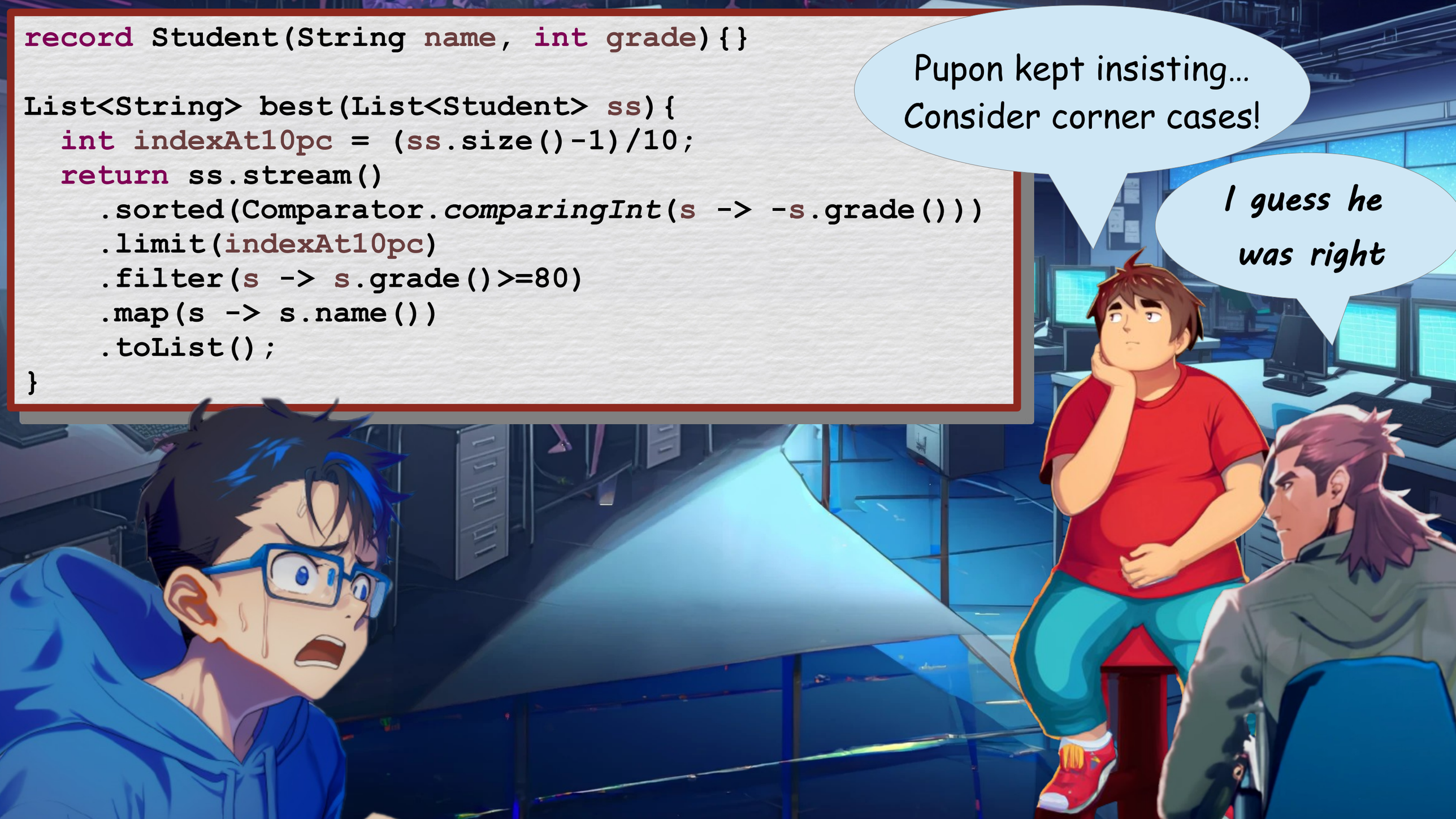

Wait... So we may have to return more then 10% of the students then!
Yes; so limit should not be used!
.limit(indexAt10pc
So the question actually means …
“find all students with more than 80 grade and their grade is no less than the 90th percentile”

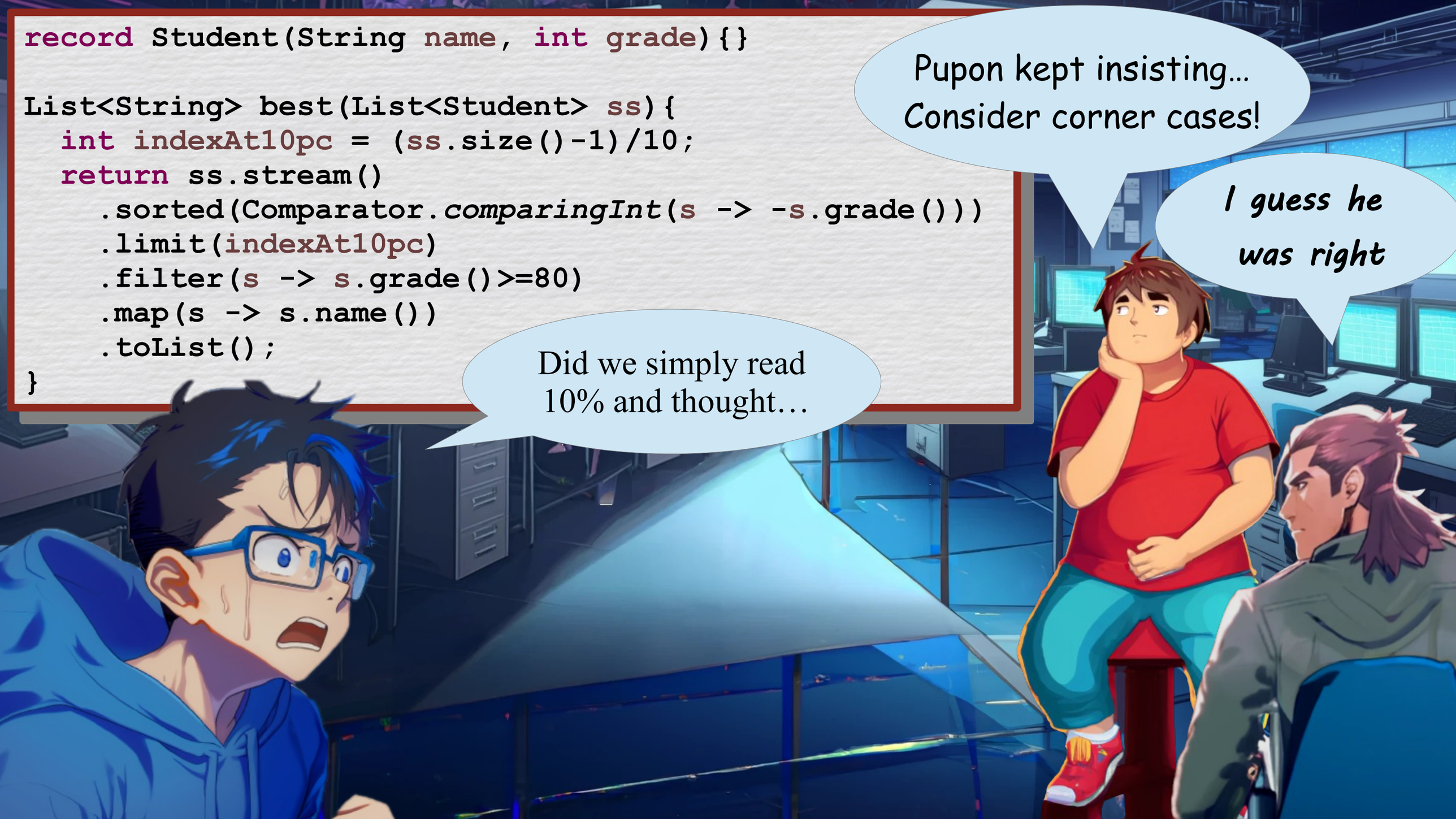

Wait... So we may have to return more then 10% of the students then!
Yes; so limit should not be used!
.limit(indexAt10pc
So the question actually means …
“find all students with more than 80 grade and their grade is no less than the 90th percentile”
Yes, you just turned the question into a solving algorithm!

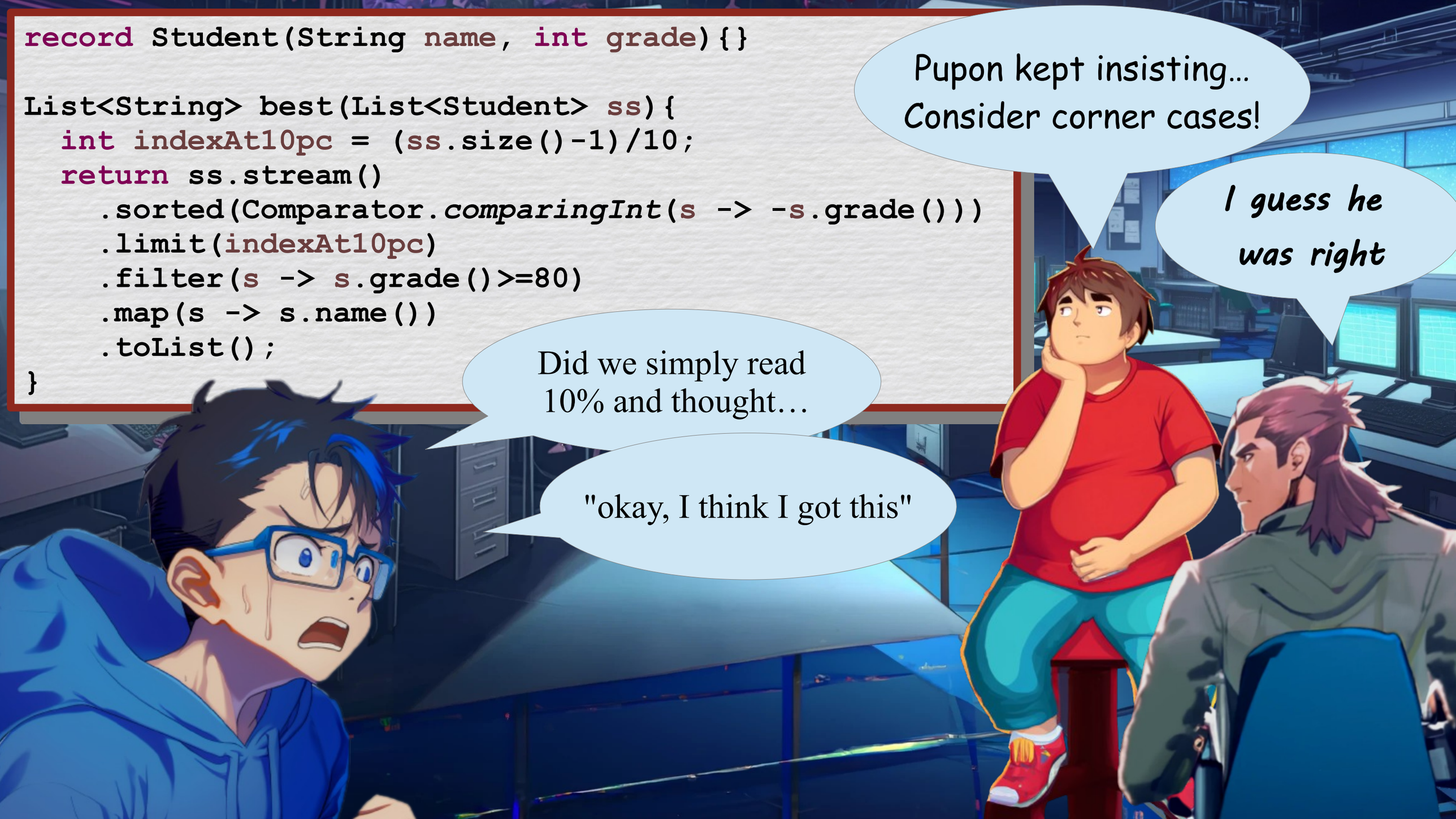
```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```

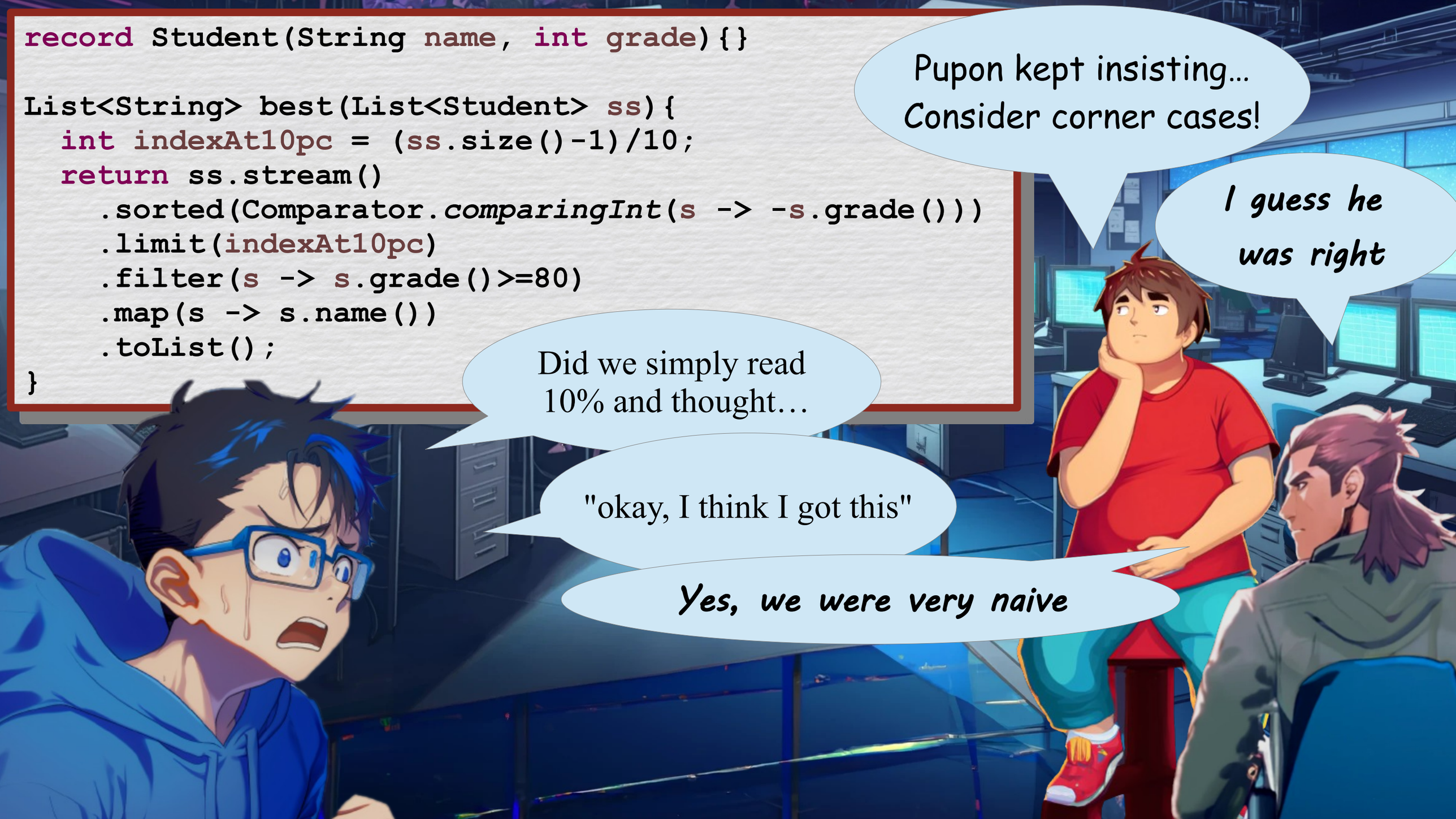


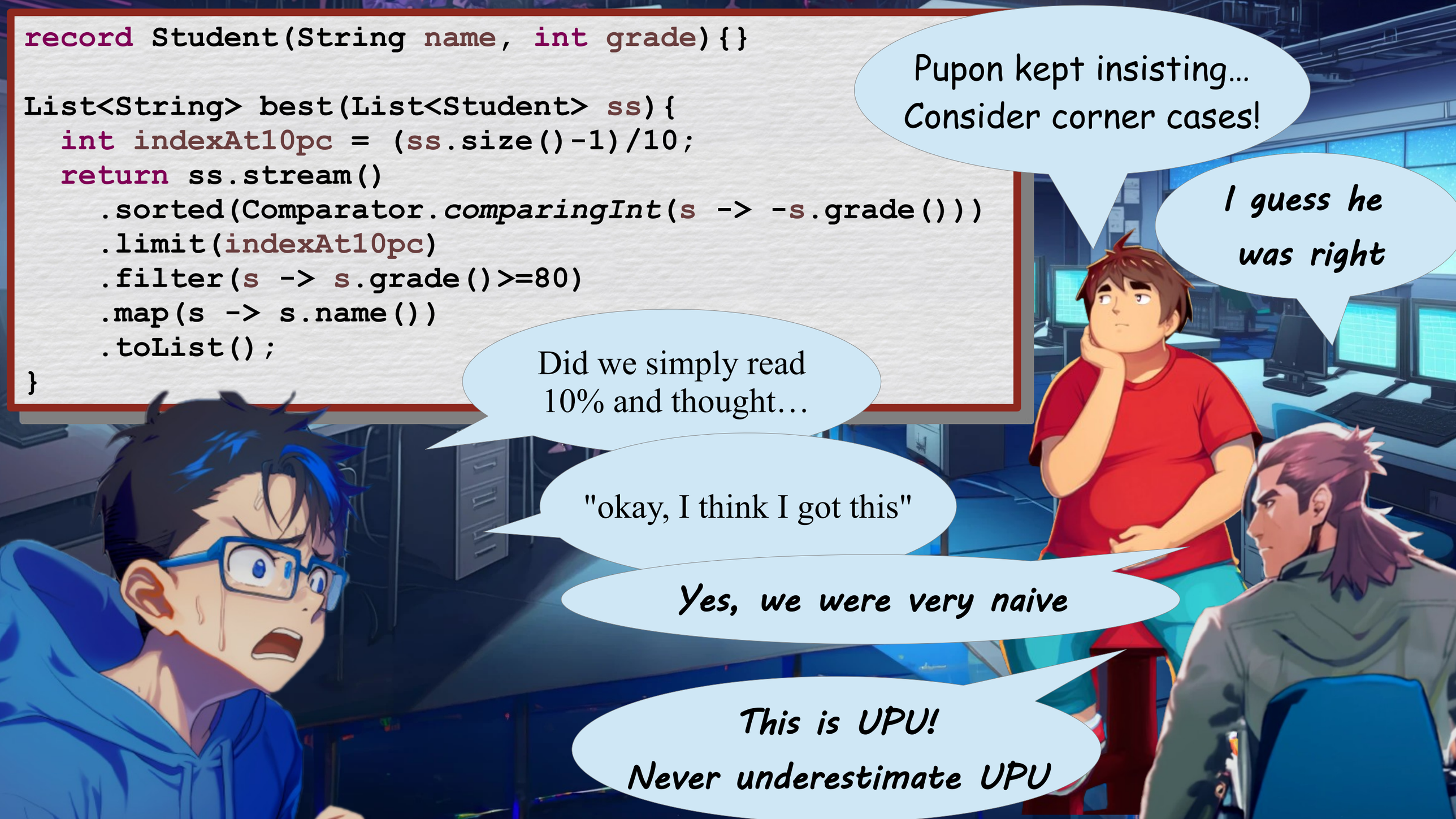
```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  return ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .limit(indexAt10pc)
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```




record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Pupon kept insisting…
Consider corner cases!
I guess he was right
Did we simply read 10% and thought…


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Pupon kept insisting…
Consider corner cases!
I guess he was right
Did we simply read 10% and thought…
"okay, I think I got this"


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Pupon kept insisting…
Consider corner cases!
I guess he was right
Did we simply read 10% and thought…
"okay, I think I got this"
Yes, we were very naive


record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
return ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.limit(indexAt10pc)
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
Pupon kept insisting...
Consider corner cases!
I guess he was right
Did we simply read 10% and thought…
"okay, I think I got this"
Yes, we were very naive
This is UPU!
Never underestimate UPU

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
.skip(indexAt10pc)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   int indexAt10pc = (ss.size()-1)/10;
   ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.filter(s -> s.grade()>=80)
```

```
.skip(indexAt10pc)
```

```
Optional<Student> threshold = ss.stream()
```

```
.filter(s -> s.grade()>=t
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   int indexAt10pc = (ss.size()-1)/10;
   ????
}
```

Ok, lets start again.
We need to explore
the list twice!

We will use the return later.
We can start with the optional!

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.skip(indexAt10pc)
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

Since there is no semicolon at the end, we have to chain more calls after it!

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
   int indexAt10pc = (ss.size()-1)/10;
   Optional<Student> threshold = ss.stream()
   ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.skip(indexAt10pc)
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

Then we sort on the grade

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
return ss.stream()
```

```
.map(s -> s.name())
```

```
.skip(indexAt10pc)
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()
```

```
.sorted(Comparator.comparingInt(s -> -s.grade()))
```

Then we sort on the grade

We use '-grade' to get the max first!

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
.map(s -> s.name())
```

```
return ss.stream()
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

Now we need to actually select the threshold

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
.map(s -> s.name())
```

```
return ss.stream()
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

Now we need to actually select the threshold

I do not think I could figure it out without help, but looking at the options...

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
  ????
}
```

```java
.toList();
```

```java
.findFirst();
```

```java
.skip(indexAt10pc)
```

```java
.map(s -> s.name())
```

```java
return ss.stream()
```

```java
.filter(s -> s.grade()>=80)
```

```java
.filter(s -> s.grade()>=threshold.get().grade())
```

Now we need to actually select the threshold

I do not think I could figure it out without help, but looking at the options...

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
  ????
}
```

```
.toList();
```

```
.findFirst();
```

```
.skip(indexAt10pc)
```

```
.map(s -> s.name())
```

```
return ss.stream()
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grade())
```

What if we skip up to the right point, and then we select the first element?

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  ????
}
```

Yes, skip(x) plus findFirst()
is basically like a List.get(x)!

```
.toList();
```

```
.map(s -> s.name())
```

```
return ss.stream()
```

```
.filter(s -> s.grade()>=80)
```

```
.filter(s -> s.grade()>=threshold.get().grad
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  ????
}
```

Yes, skip(x) plus findFirst()
is basically like a List.get(x)!

```
.toList();

.map(s -> s.name())

return ss.stream()

.filter(s -> s.grade()>=80)

.filter(s -> s.grade()>=threshold.get().grad
```

Now it is finally
time for the return!

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    ????
}
```

Now is the good time to apply the two filters

```java
.toList();
```

```java
.map(s -> s.name())
```

```java
.filter(s -> s.grade()>=80)
```

```java
.filter(s -> s.grade()>=threshold.get().grade())
```

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```java
.toList();
```

```java
.map(s -> s.name())
```

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.toList();
```

```
.map(s -> s.name())
```

Wait a moment!

Is that get safe?

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.toList();
```

```
.map(s -> s.name())
```

Wait a moment!

Is that get safe?

What if the list is empty?

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.toList();
```

```
.map(s -> s.name())
```

Yes, it's safe!

```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.toList();
```

```
.map(s -> s.name())
```

Yes, it's safe!

threshold is empty only
if the list is empty.
If the list is empty, then the
filter is never executed.

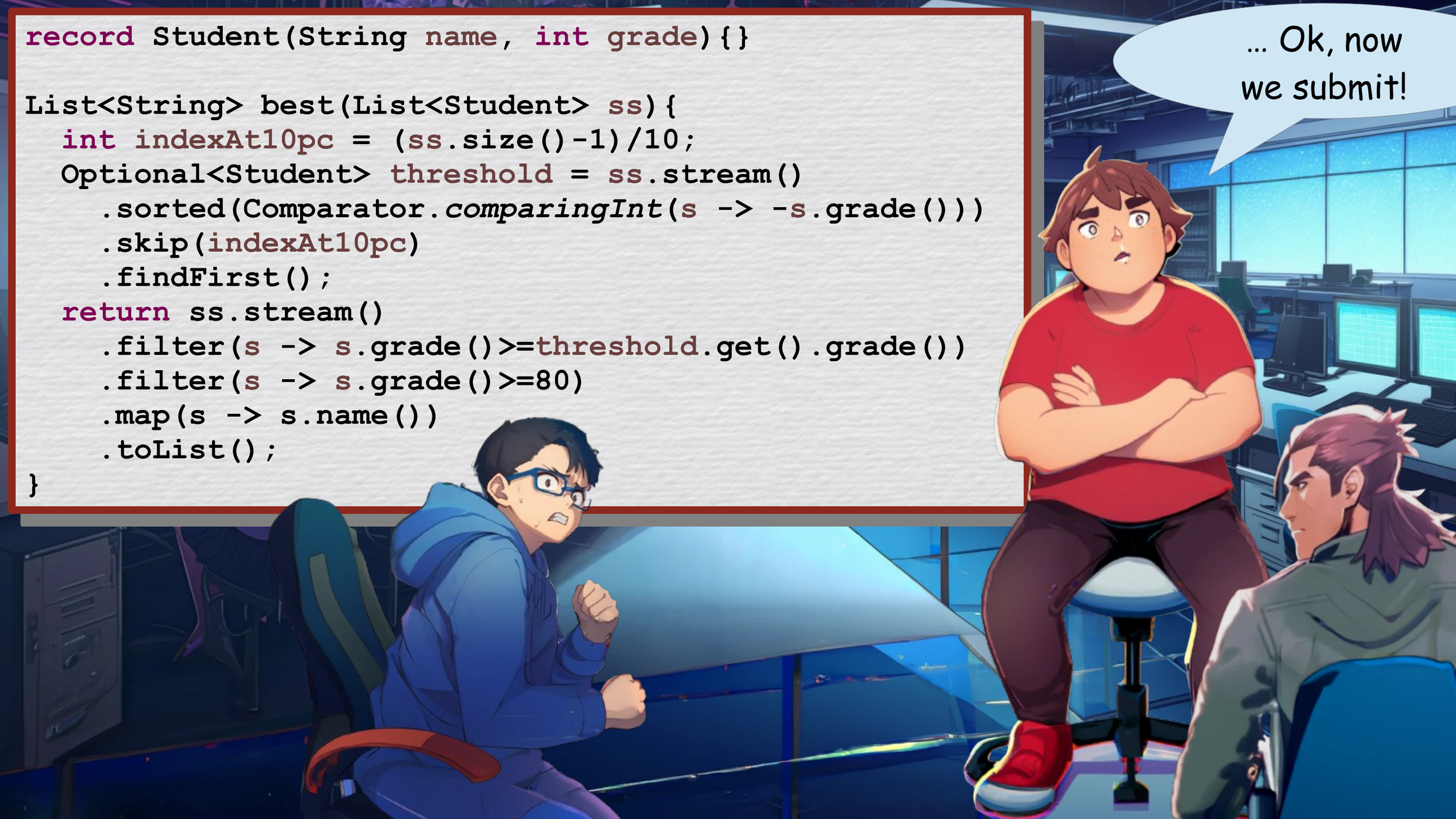
```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    ????
}
```

```
.map(s -> s.name())
```

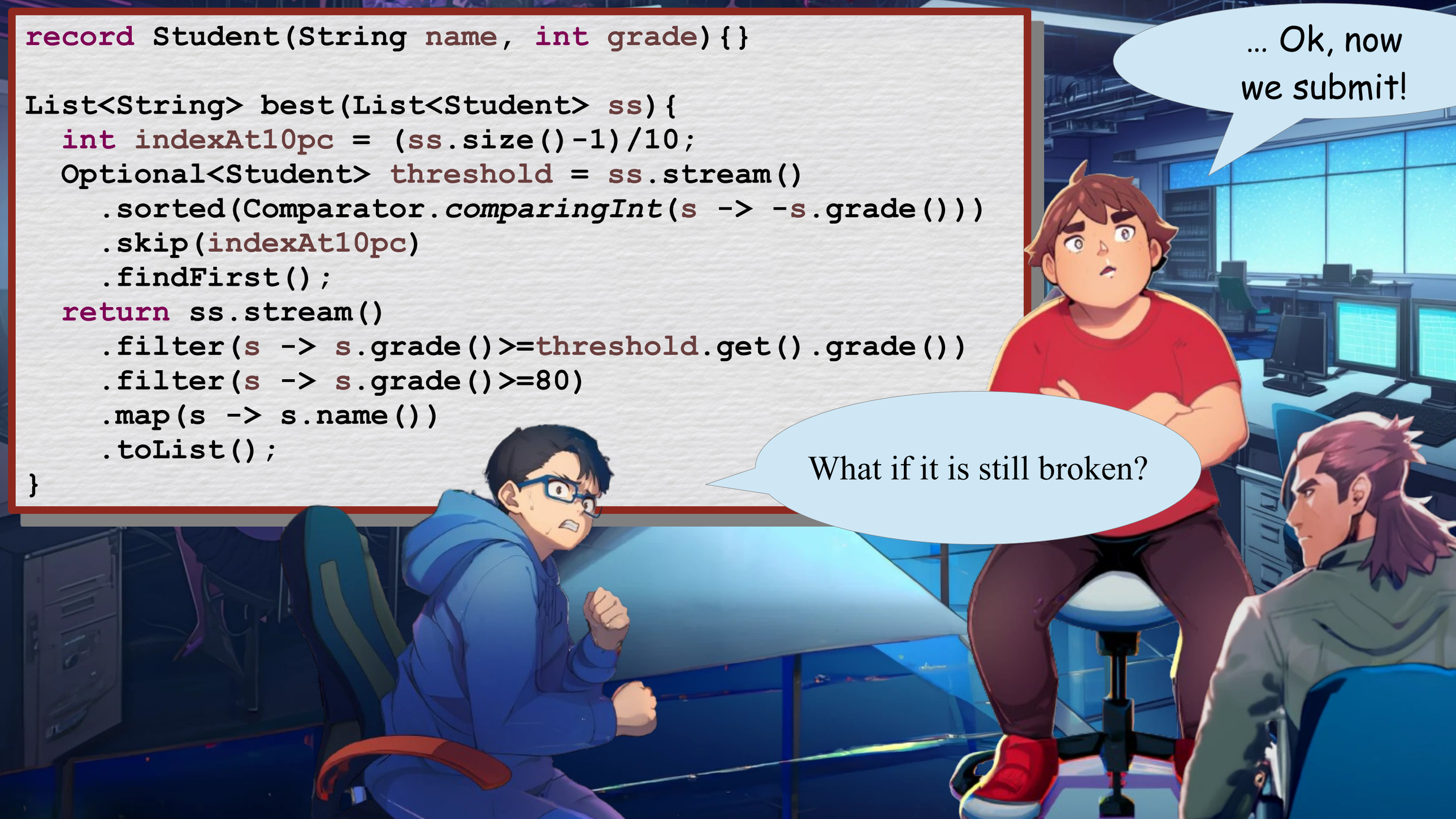
```
.toList();
```

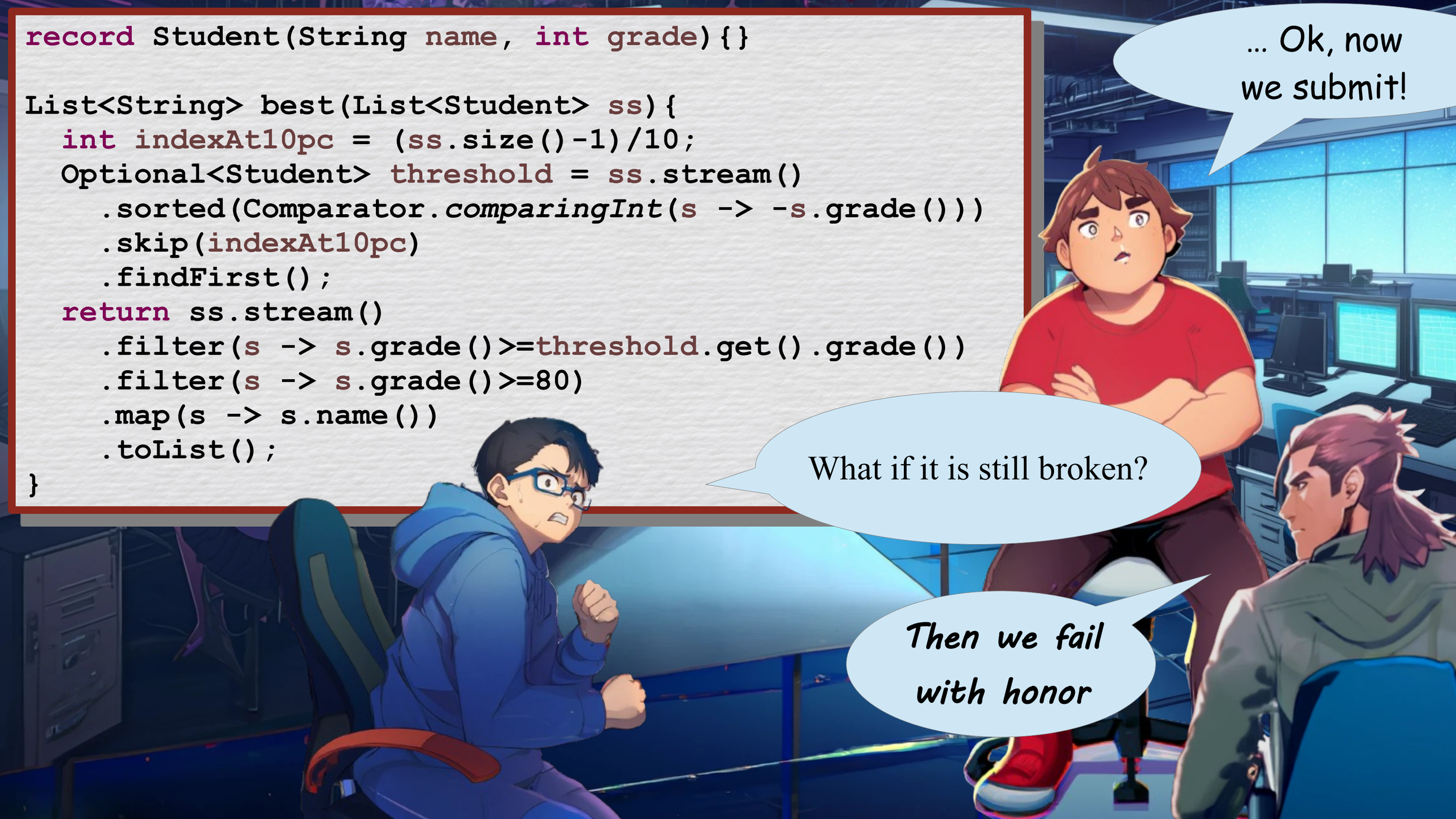
Finally, we map to the names, and we toList it!

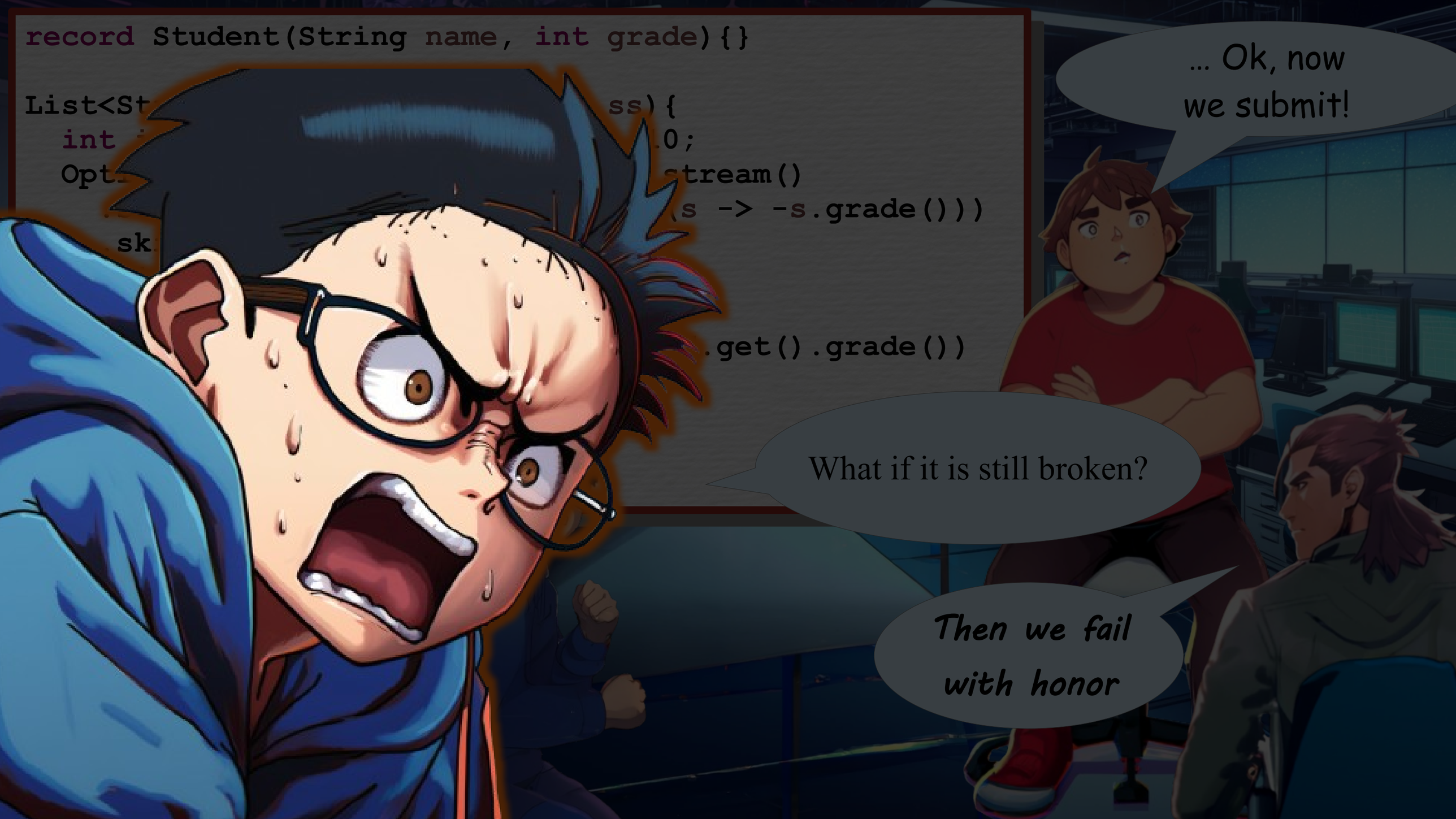
```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```



```java
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```




record Student(String name, int grade){}
List<String> best(List<Student> ss){
int indexAt10pc = (ss.size()-1)/10;
Optional<Student> threshold = ss.stream()
.sorted(Comparator.comparingInt(s -> -s.grade()))
.skip(indexAt10pc)
.findFirst();
return ss.stream()
.filter(s -> s.grade()>=threshold.get().grade())
.filter(s -> s.grade()>=80)
.map(s -> s.name())
.toList();
}
... Ok, now we submit!
What if it is still broken?
Then we fail with honor

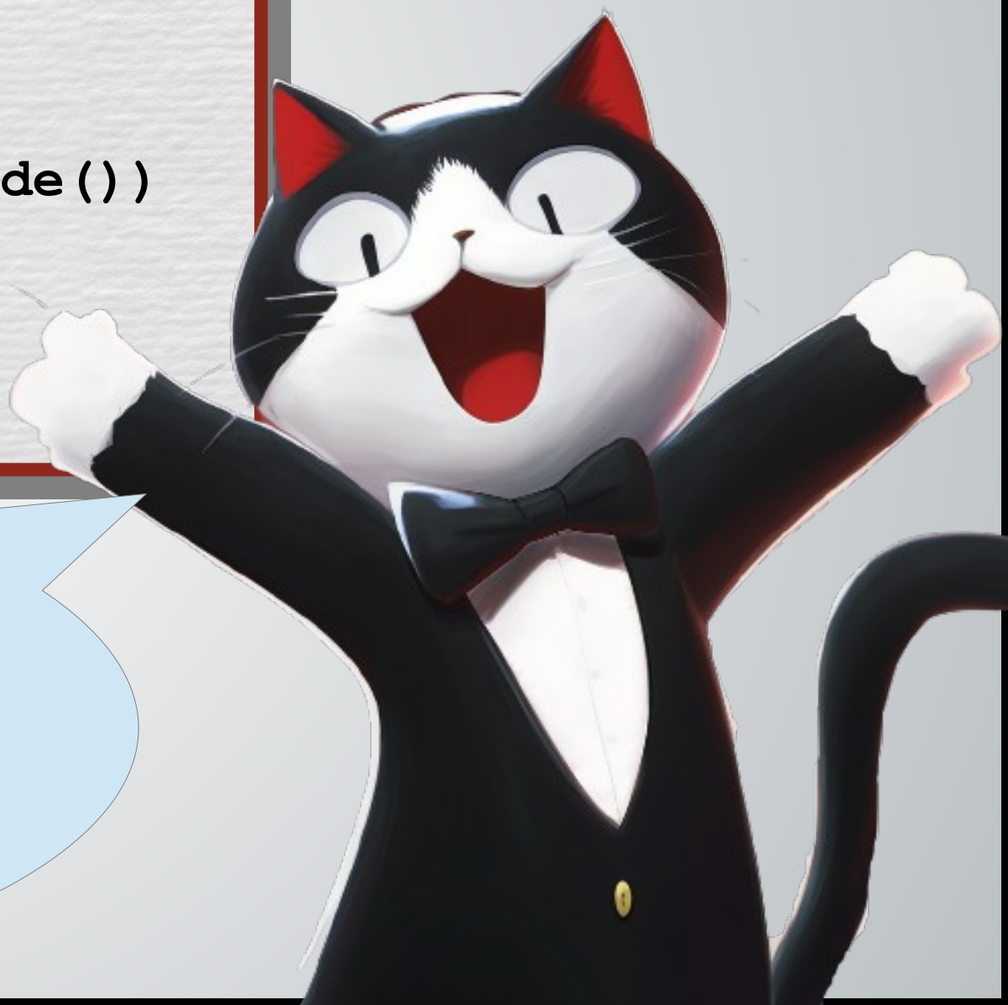

record Student(String name, int grade){}
List<St
ss){
int
0;
Opt
stream()
s -> -s.grade()))
sk
.get().grade())
... Ok, now we submit!
What if it is still broken?
Then we fail with honor

The trio submit their answer!

```
List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```

The trio submit their answer!

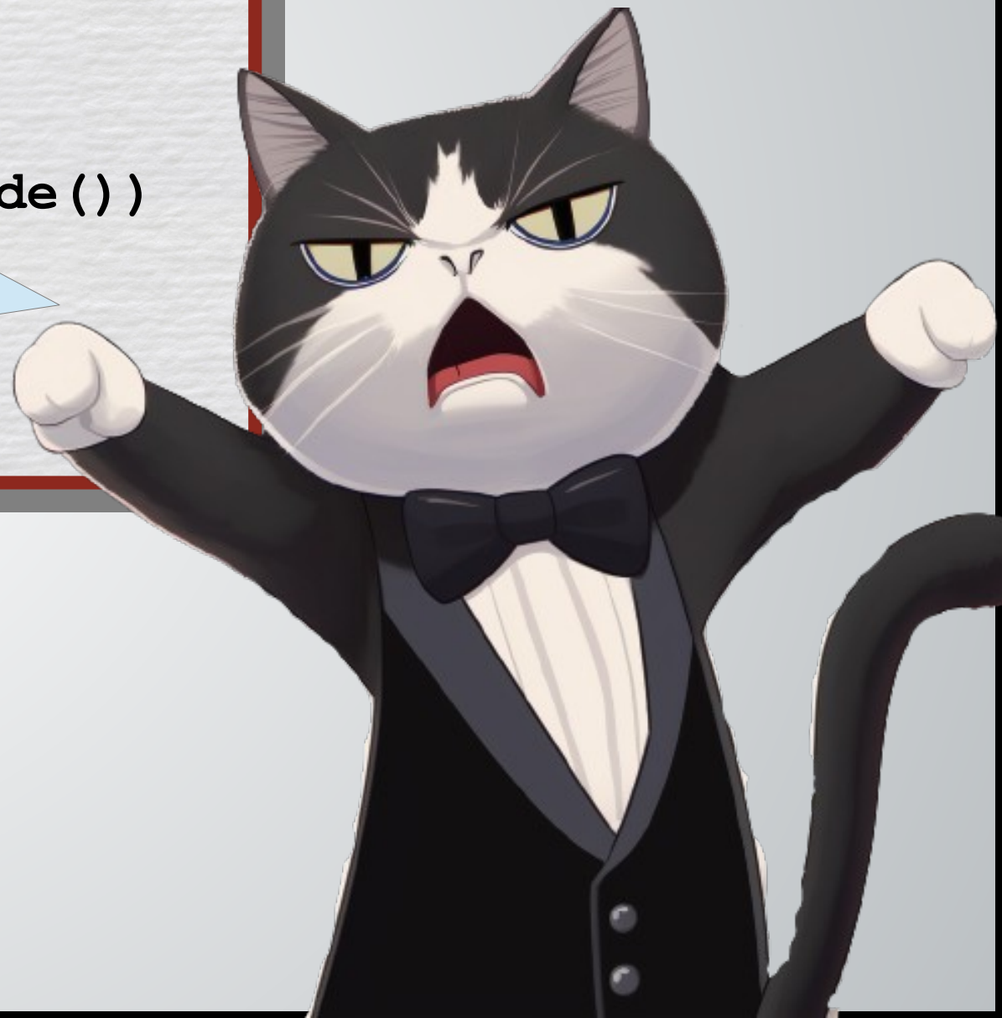
```
List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst();
  return ss.stream()
    .filter(s -> s.grade()>=threshold.get().grade())
    .filter(s -> s.grade()>=80)
    .map(s -> s.name())
    .toList();
}
```



Good news!
The answer is correct!

```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst()
  return
                                              ade())
    .m
    .toLis
}
```

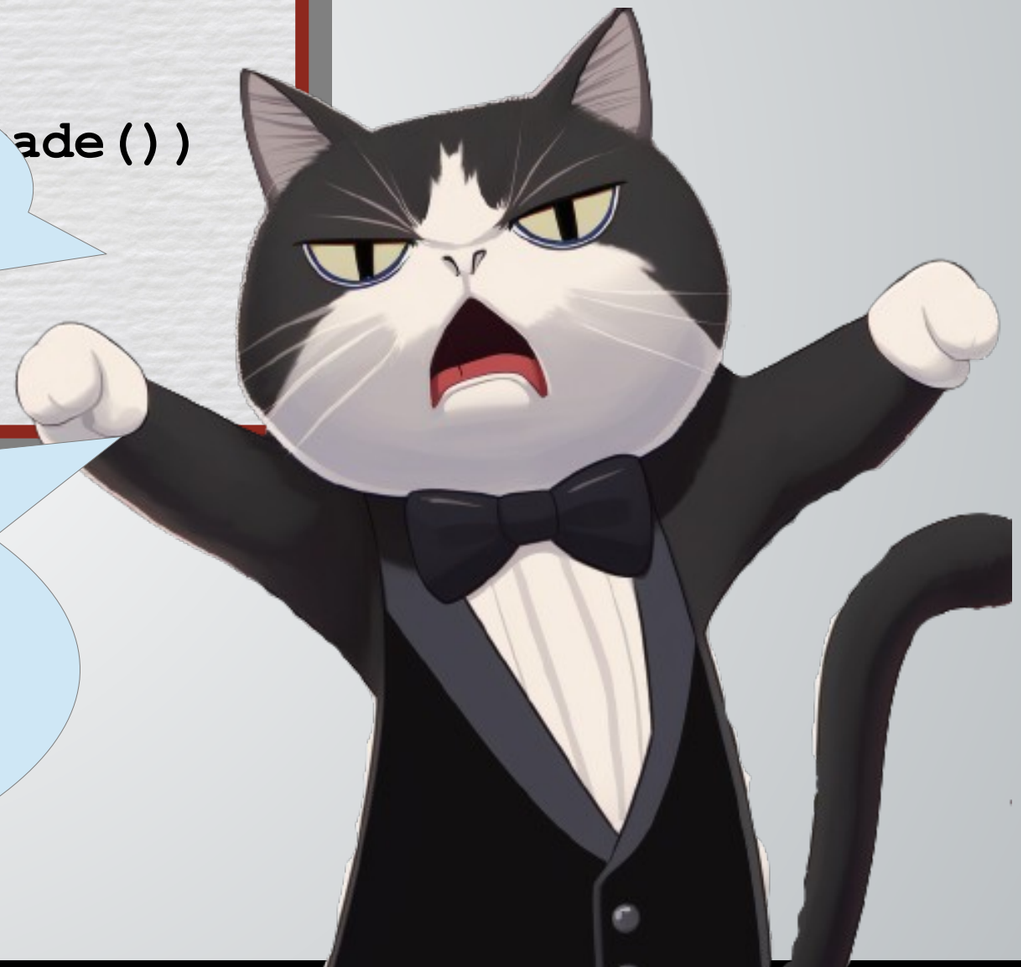
You are the most persistent students I've ever met at first year



```
record Student(String name, int grade){}

List<String> best(List<Student> ss){
  int indexAt10pc = (ss.size()-1)/10;
  Optional<Student> threshold = ss.stream()
    .sorted(Comparator.comparingInt(s -> -s.grade()))
    .skip(indexAt10pc)
    .findFirst()
  return
                                          ade())
    .in
    .toLis
}
```

You are the most persistent students I've ever met at first year

Your answers are all correct, but you took the longest of anyone ever recorded.




Finally!
Is it over?


Finally!
Is it over?
Yes, it is done!
YAAWN


Finally!
Is it over?
Yes, it is done!
YAAWN
Now we can go to sleep;
I'm exhausted


Finally!
Is it over?
Yes, it is done!
YAAWN
Now we can go to sleep;
I'm exhausted
Good night! I'm going
to sleep like a rock too.
No dinner, just sleep!