Welcome back

Welcome back
Let’s learn Streams one dreadful example at a time!

01: Invalid Metaphors

01: Invalid Metaphors

02: The sky is infinite

02: The sky is infinite

03: Java
Screams!
UPU
Ultimate
Programming
University
02
01
University

UPU
Ultimate
Programming
University
03
02
01
03: Java
Screams!

UPU
Ultimate
Programming
University
03

Dany is lying in bed, looking weak and ill

Dany is lying in bed, looking weak and ill
Ugh, I'm sick!

Dany is lying in bed, looking weak and ill
Ugh, I'm sick!
I hope I will not miss too much in programming class

A few days later

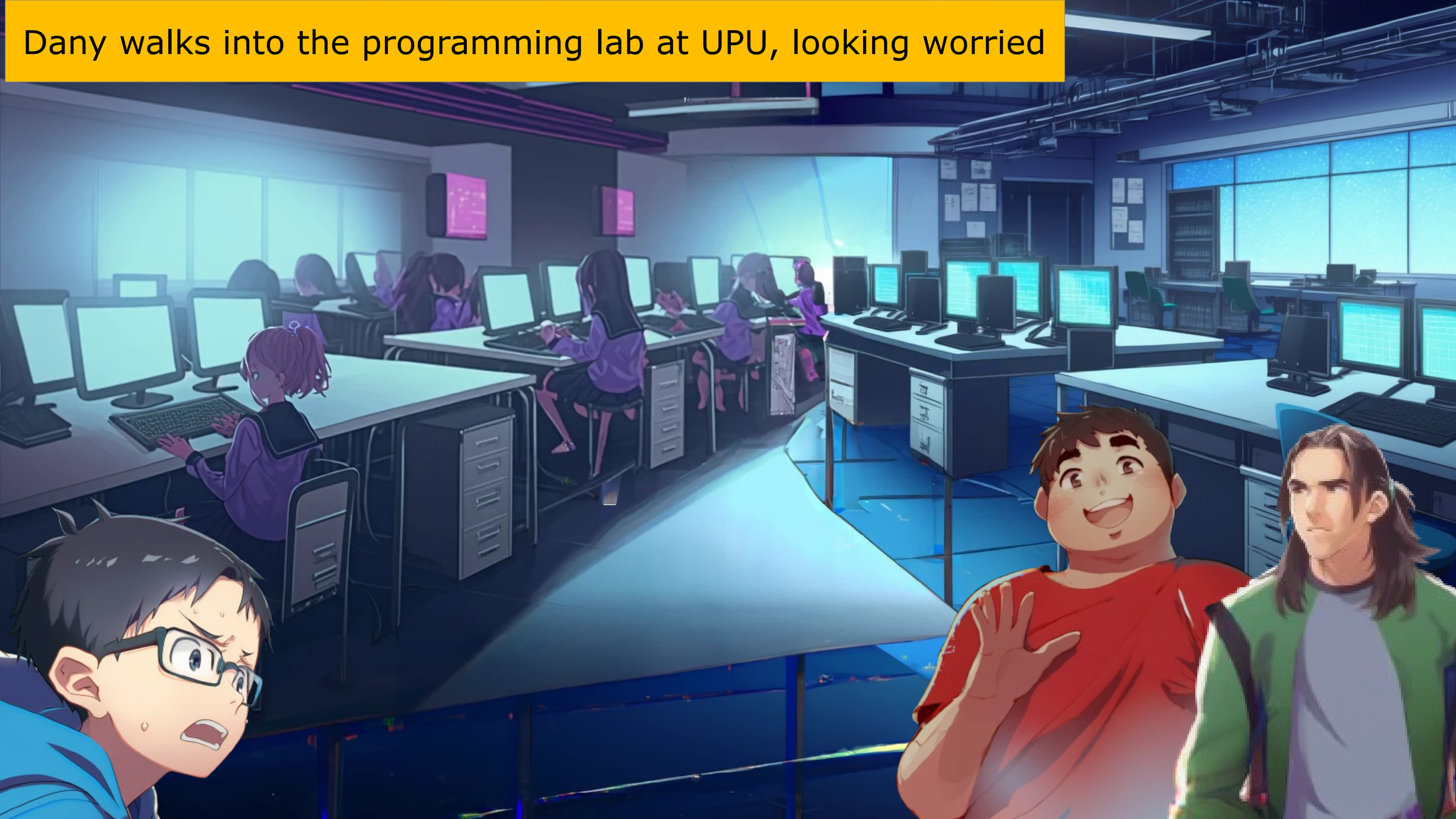
A few days later
I can't believe I missed two lectures

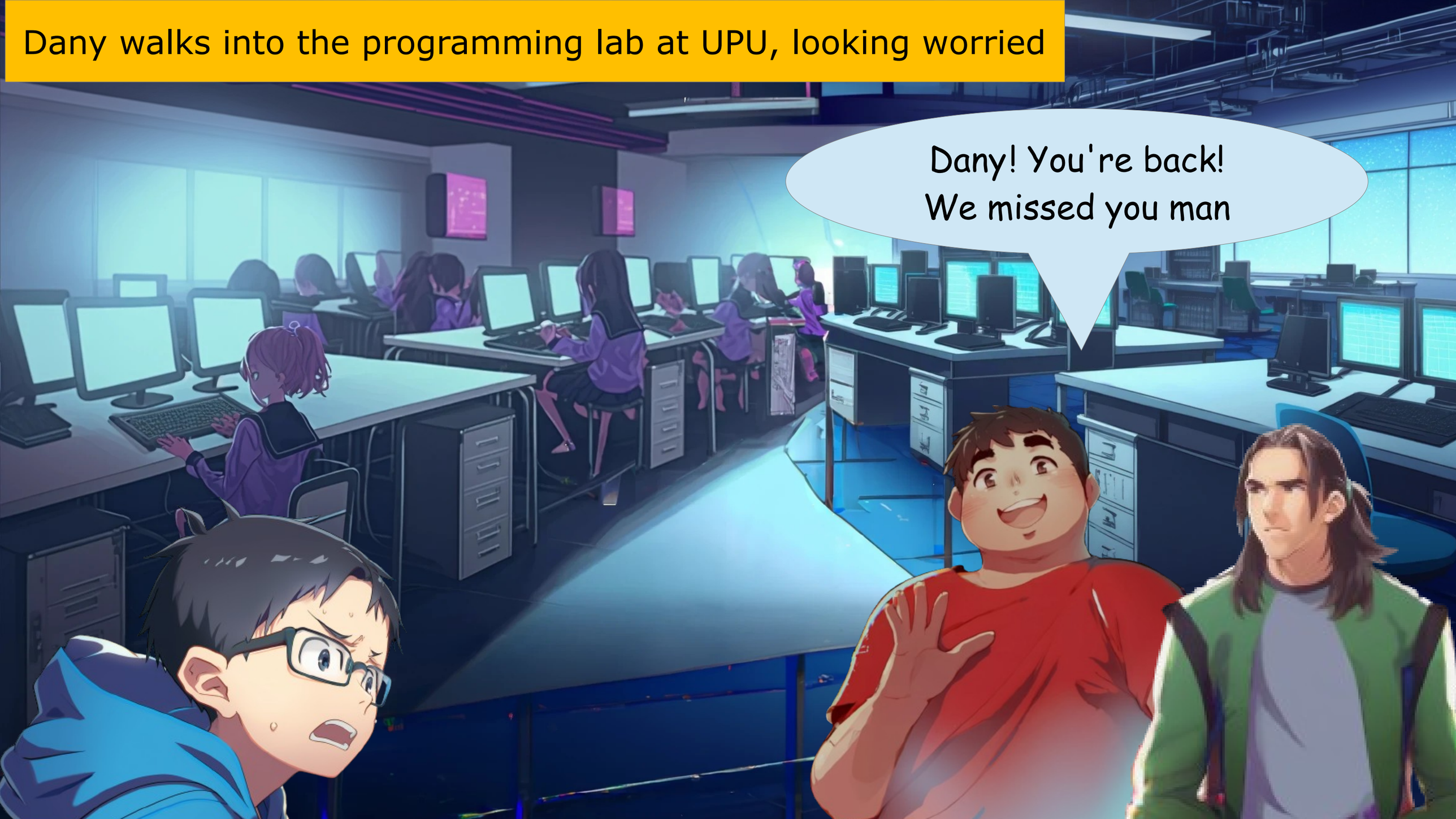
A few days later
I can't believe I missed two lectures
How am I going to catch up?

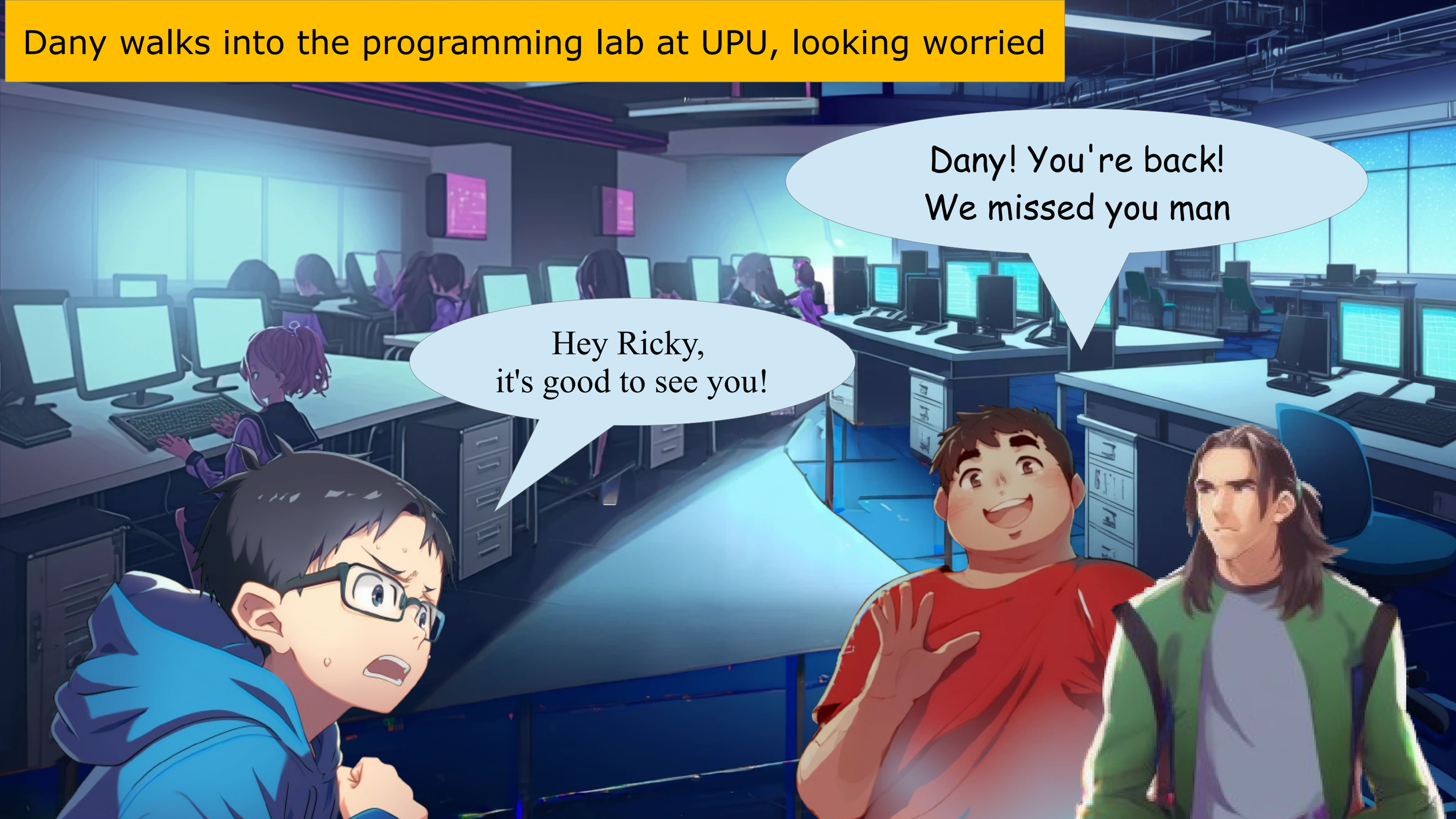
A few days later
I can't believe I missed two lectures
How am I going to catch up?
Where is everyone?

A few days later
I can't believe I missed two lectures
How am I going to catch up?
Where is everyone?
Oh, right, today we have the first lab session!

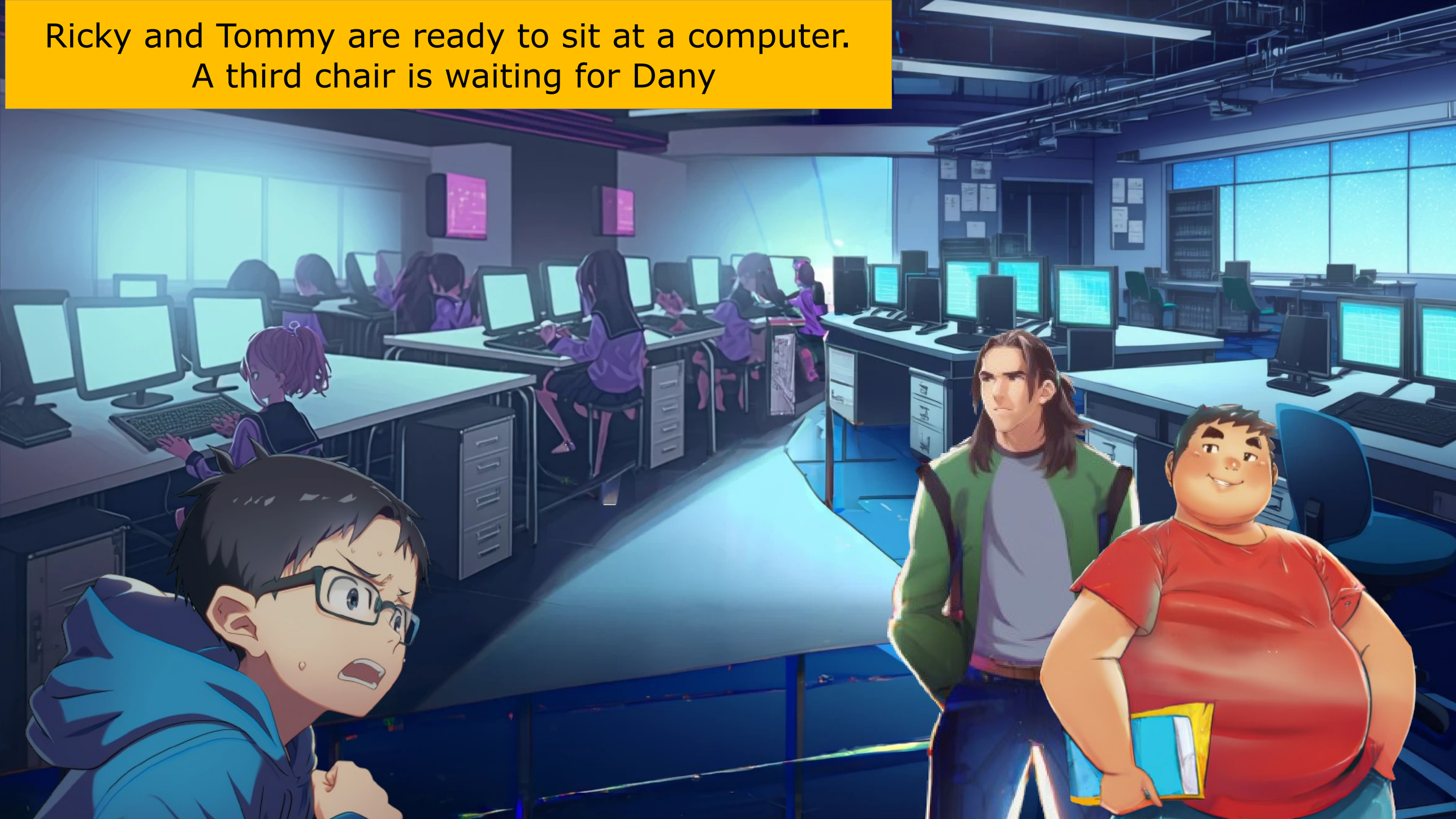
Dany walks into the programming lab at UPU, looking worried

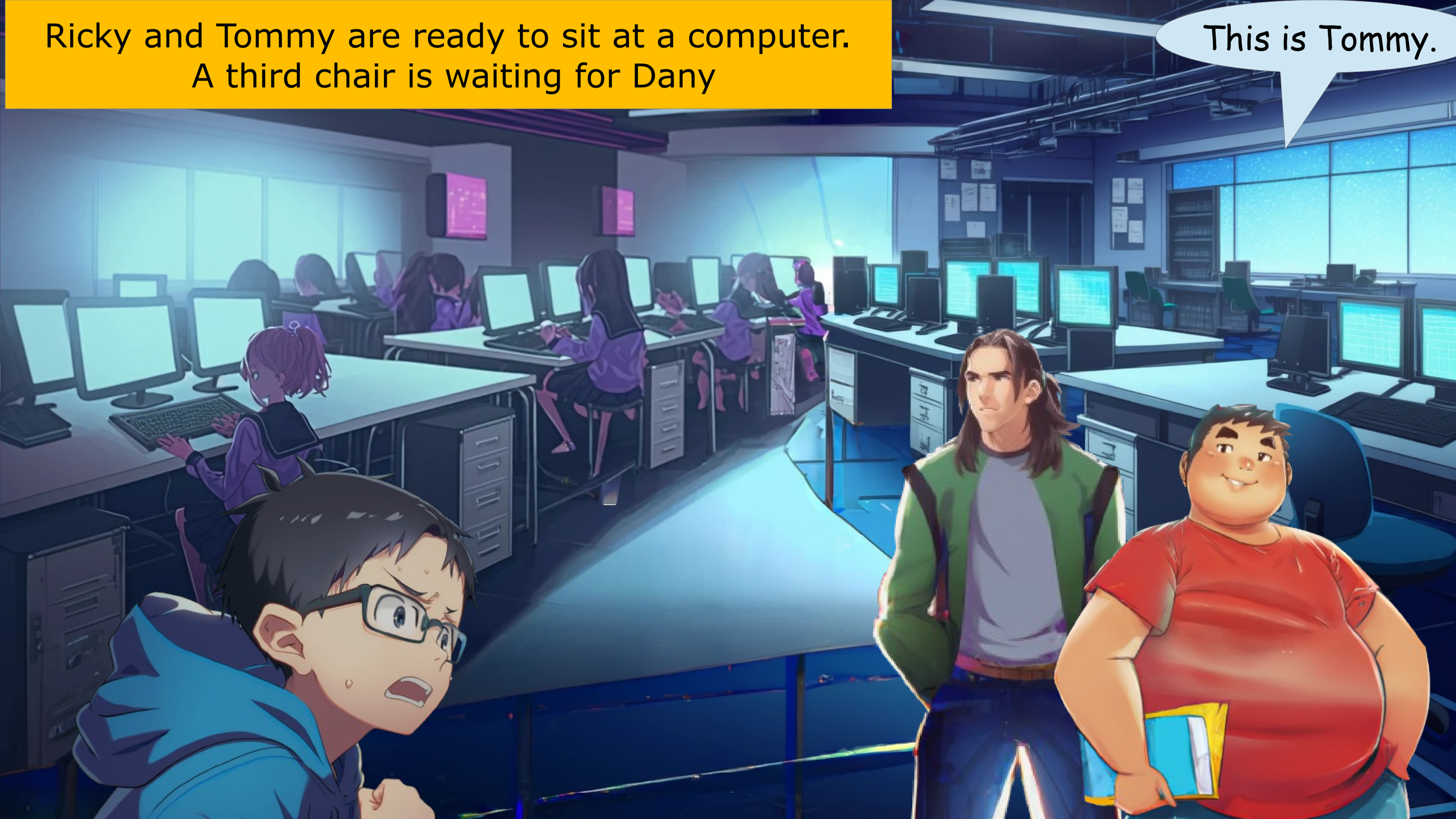



Dany walks into the programming lab at UPU, looking worried
Dany! You're back!
We missed you man

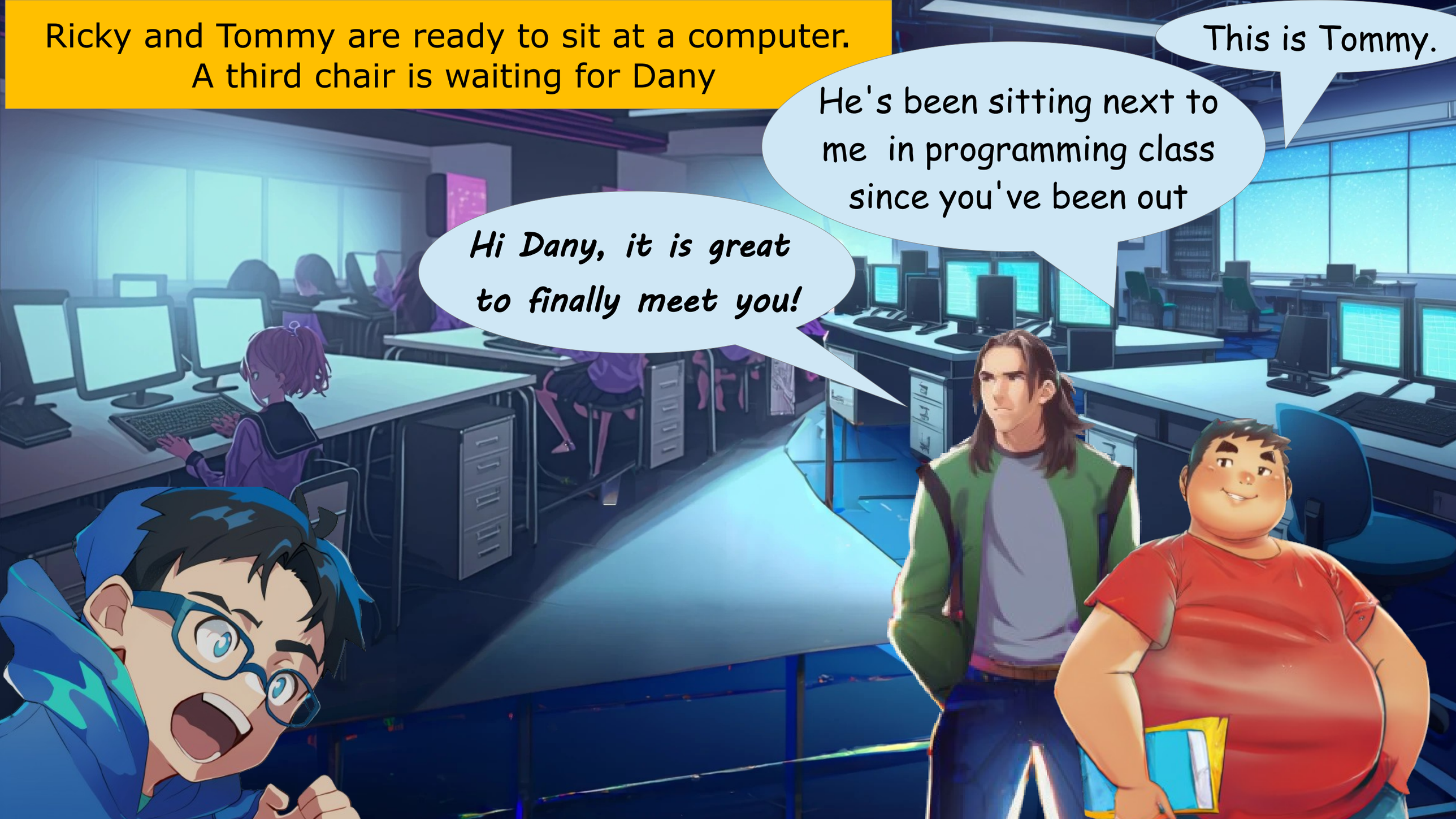

Dany walks into the programming lab at UPU, looking worried
Dany! You're back!
We missed you man
Hey Ricky,
it's good to see you!

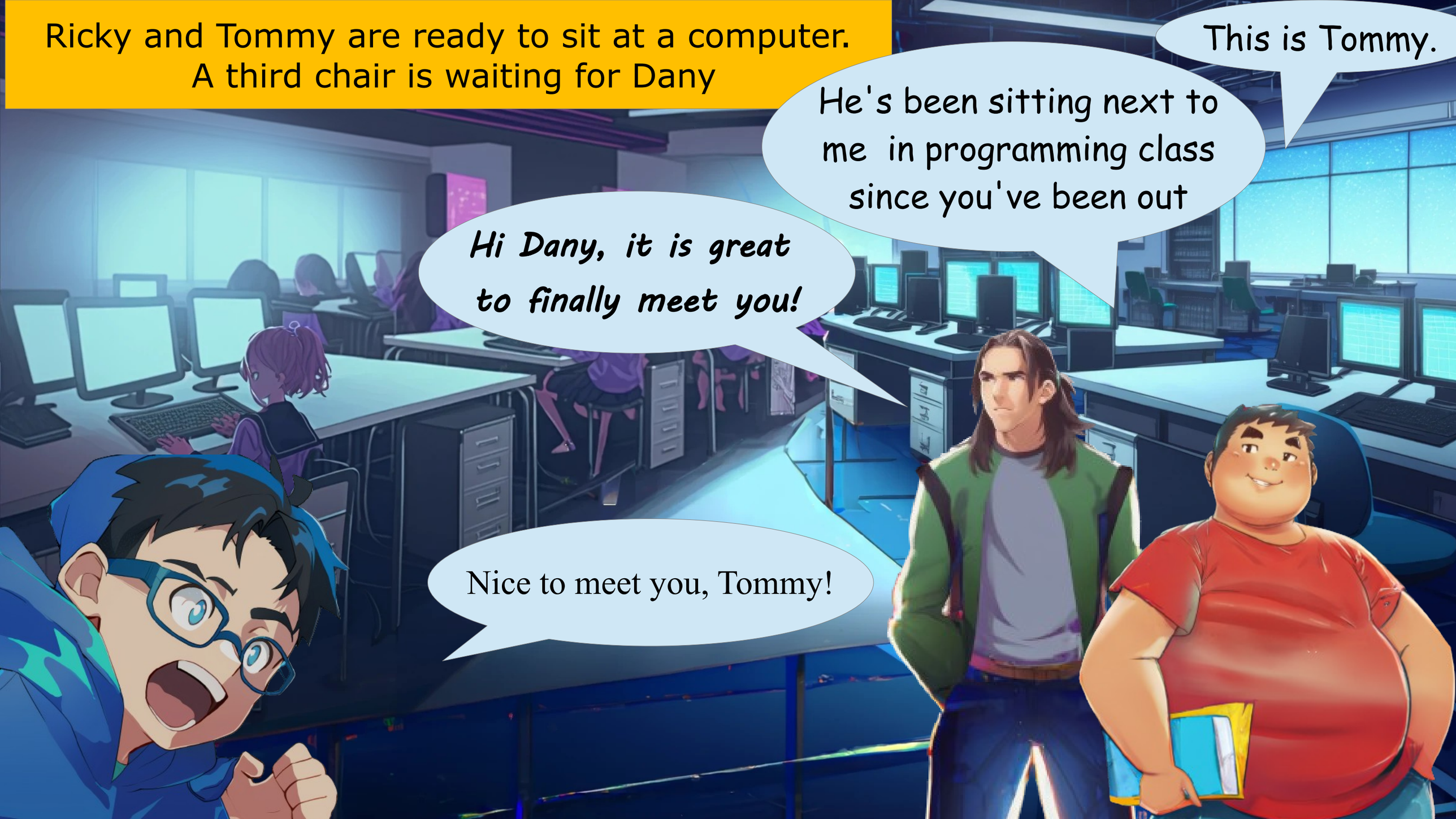
Ricky and Tommy are ready to sit at a computer.
A third chair is waiting for Dany




Ricky and Tommy are ready to sit at a computer.
A third chair is waiting for Dany
This is Tommy.

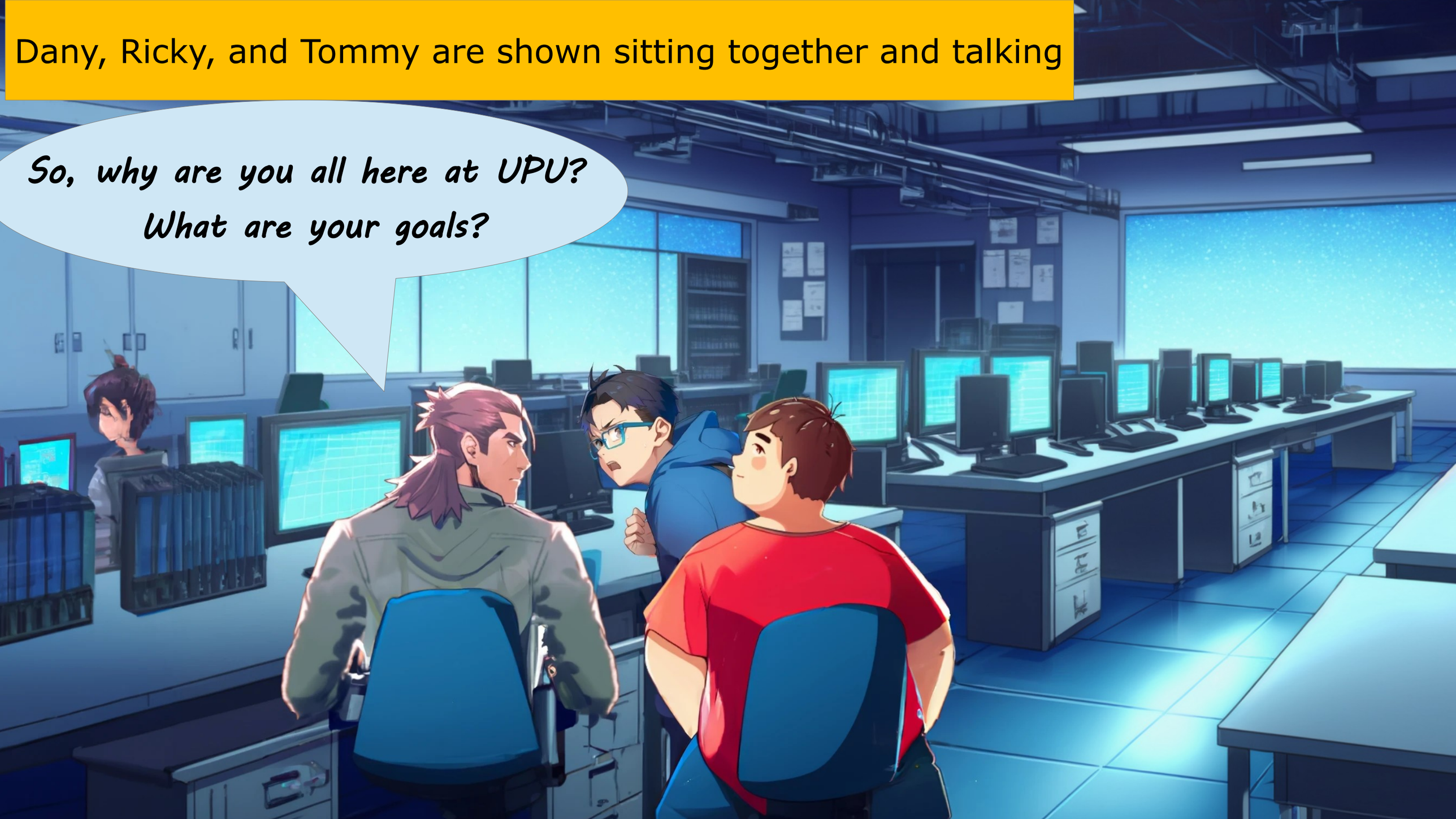
Ricky and Tommy are ready to sit at a computer.
A third chair is waiting for Dany
This is Tommy.
He's been sitting next to me in programming class since you've been out


Ricky and Tommy are ready to sit at a computer.
A third chair is waiting for Dany
This is Tommy.
He's been sitting next to me in programming class since you've been out
Hi Dany, it is great to finally meet you!

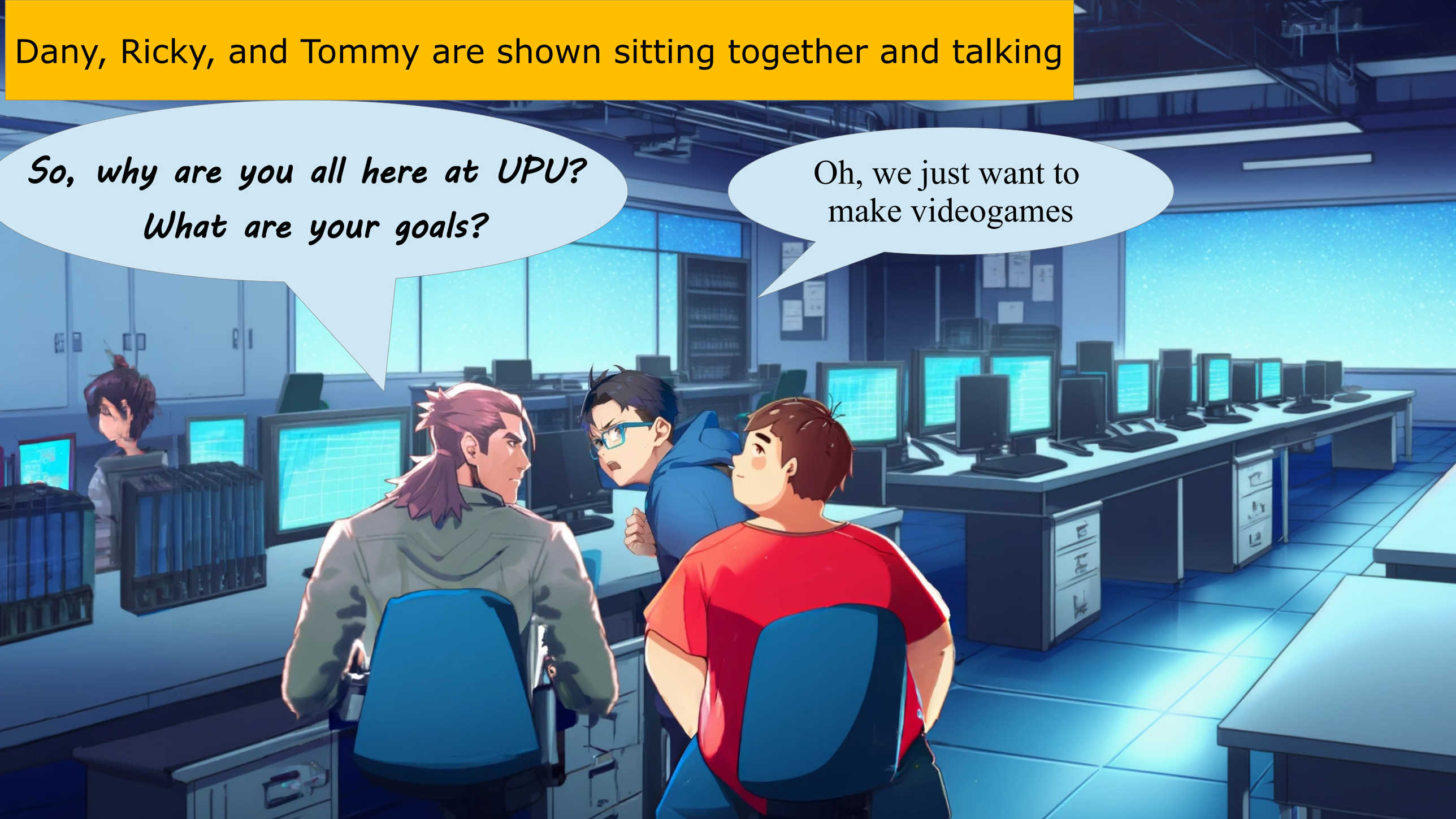

Ricky and Tommy are ready to sit at a computer.
A third chair is waiting for Dany
This is Tommy.
He's been sitting next to me in programming class since you've been out
Hi Dany, it is great to finally meet you!
Nice to meet you, Tommy!

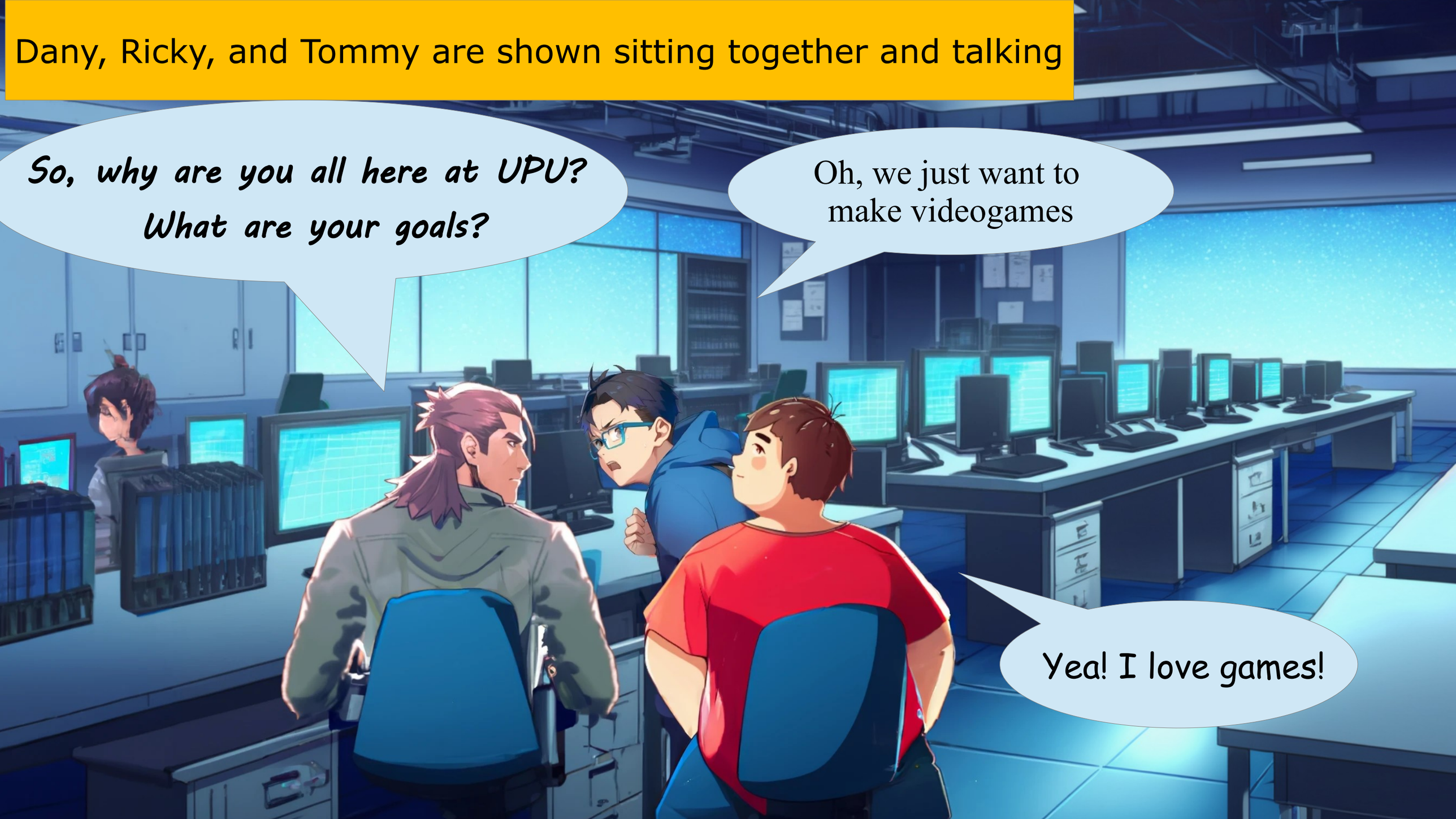
Dany, Ricky, and Tommy are shown sitting together and talking

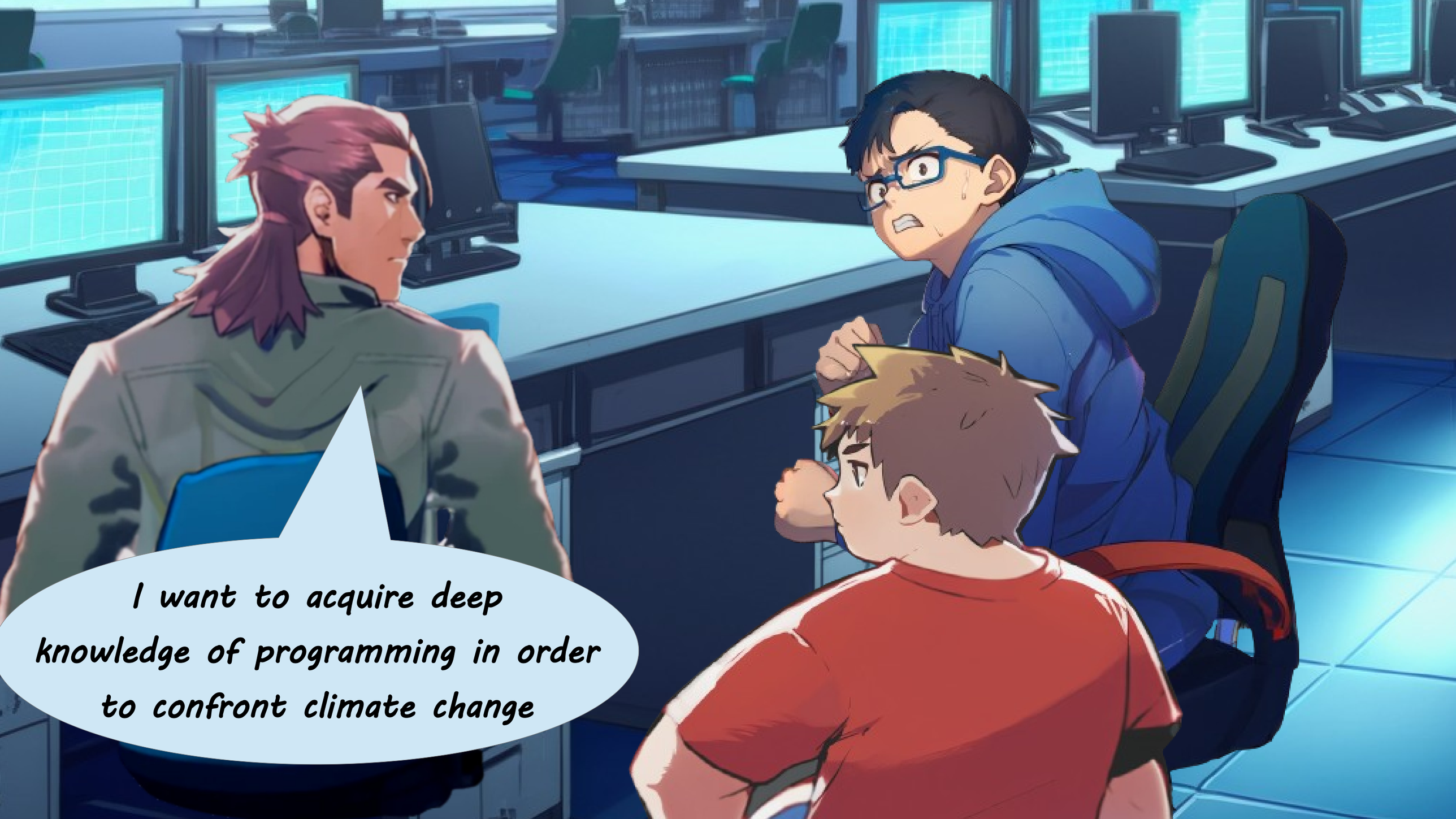
Dany, Ricky, and Tommy are shown sitting together and talking



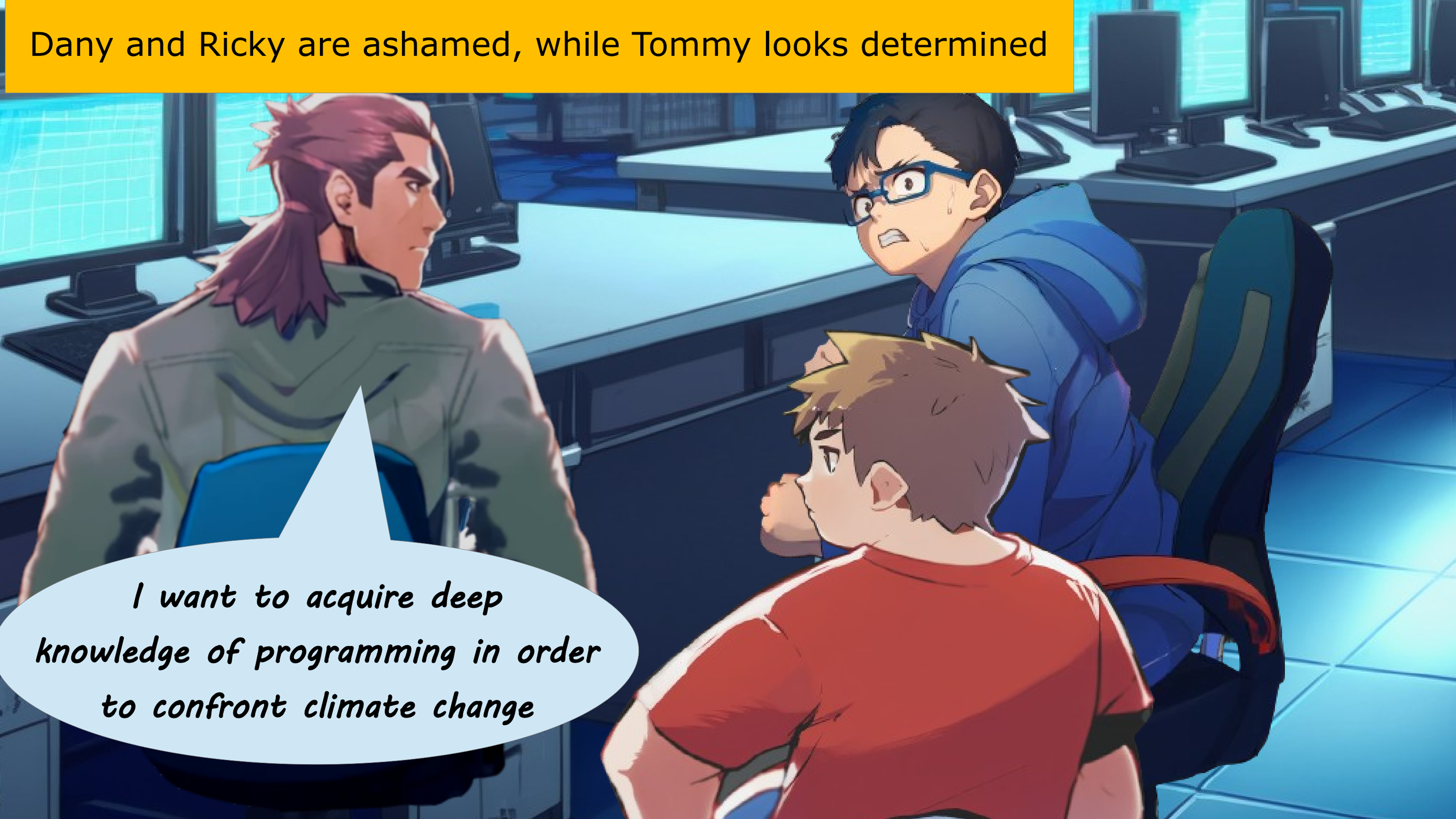
Dany, Ricky, and Tommy are shown sitting together and talking




Dany, Ricky, and Tommy are shown sitting together and talking
So, why are you all here at UPU? What are your goals?
Oh, we just want to make videogames
Yea! I love games!


I want to acquire deep knowledge of programming in order to confront climate change

Dany and Ricky are ashamed, while Tommy looks determined



I've heard great thinks about you Dany; games can't be the full scope of your ambitions

I've heard great thinks about you Dany; games can't be the full scope of your ambitions
I'm too scared to think about higher ambitions.

I've heard great thinks about you Dany; games can't be the full scope of your ambitions
I'm too scared to think about higher ambitions.
Also, I've missed a lot of material while I was sick. I don't know if I can catch up!

I hope they can fill me in on what I've missed.

I hope they can fill me in on what I've missed.
Don't worry Dany, we've been in class but we haven't really understood the topics either

I hope they can fill me in on what I've missed.
Don't worry Dany, we've been in class but we haven't really understood the topics either
I'm ashamed but I have to agree! Being there did not made a difference for me

Dany is in dismay

Dany is in dismay
Great, that makes me feel better !?!

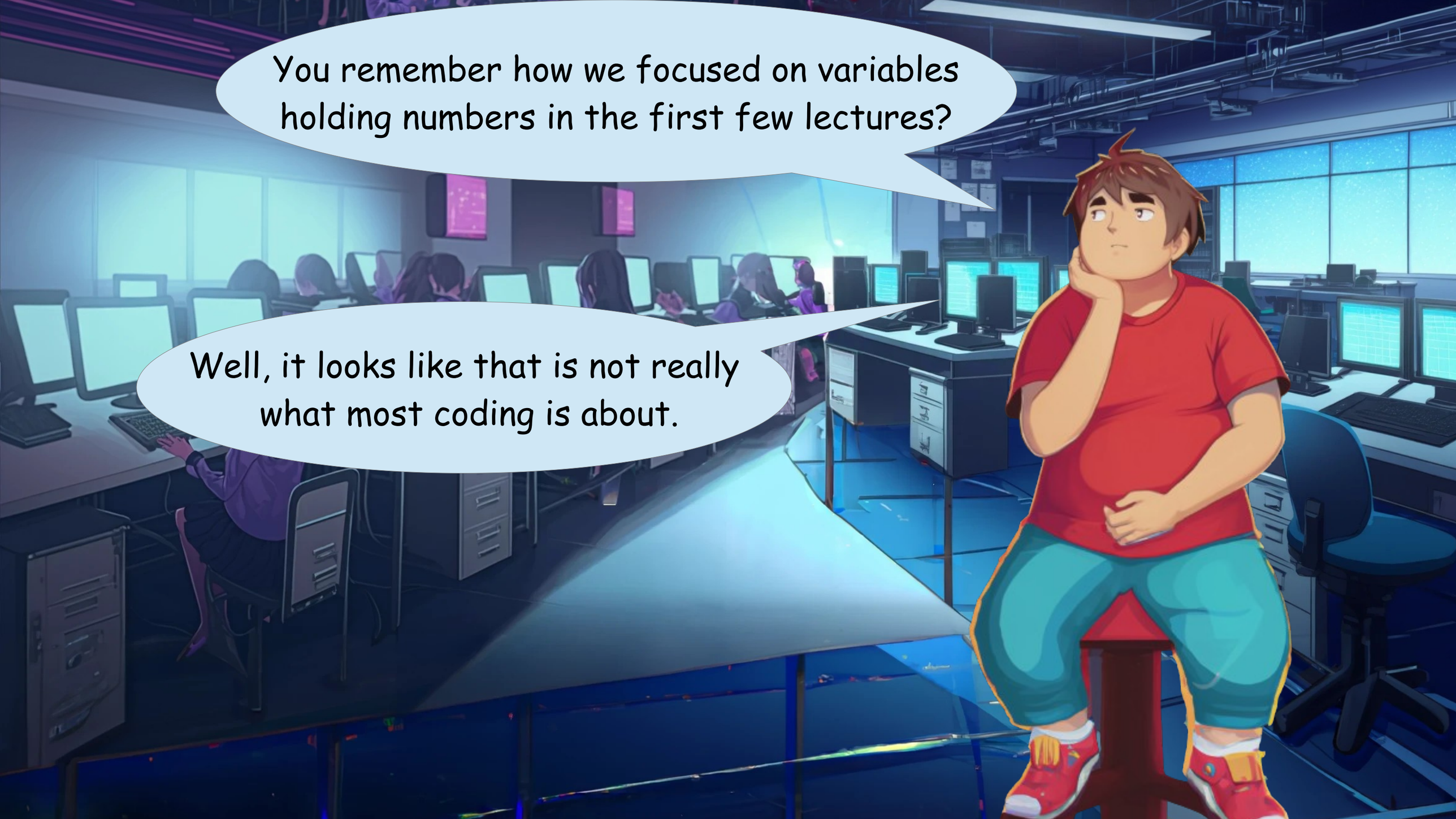
Dany is in dismay
Great, that makes me feel better !?!
Can you at least give me the gist of what I missed?

We discussed this thing called 'structured data'

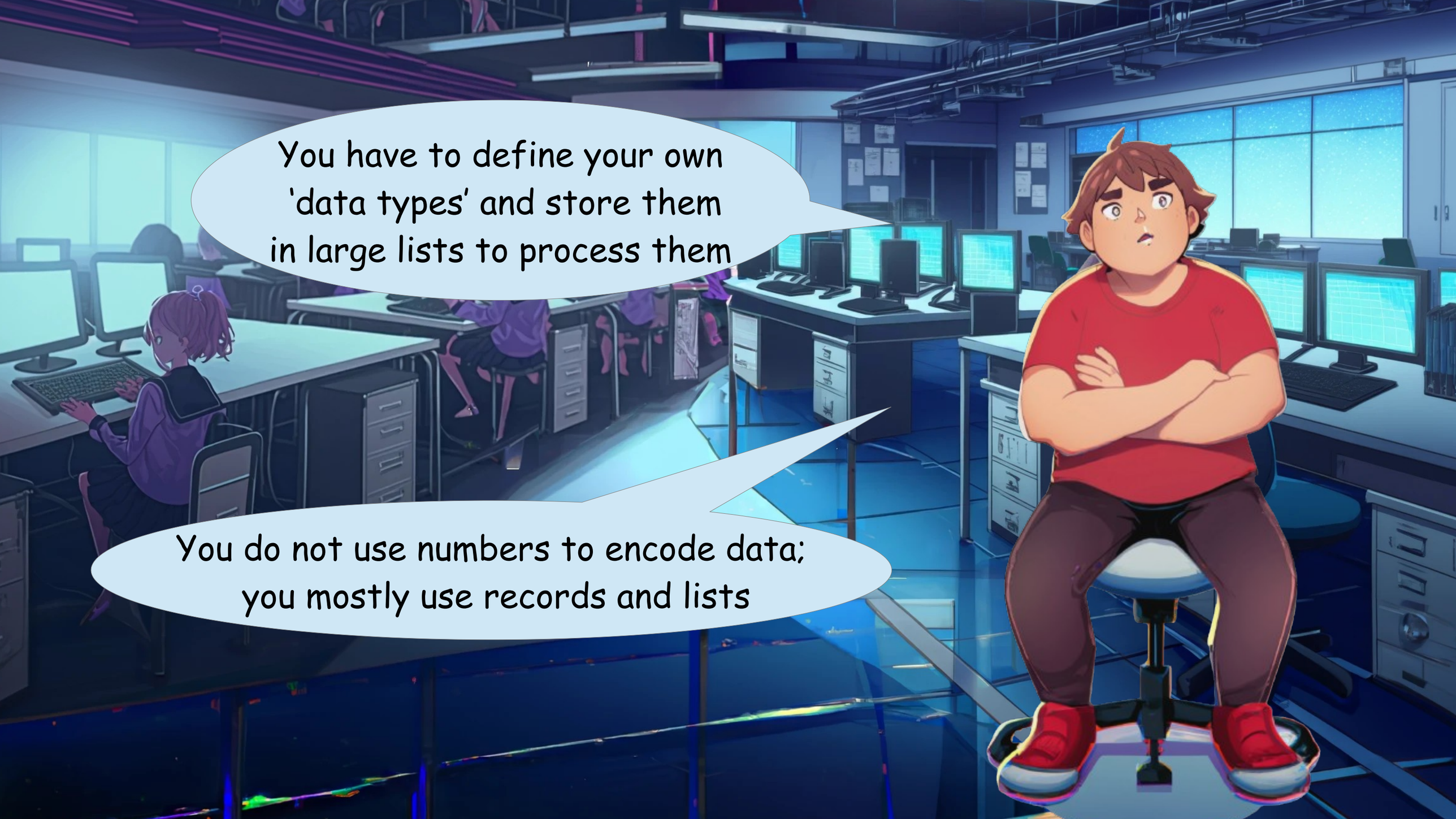
We discussed this thing called 'structured data'
Yeah, they showed us records and lists

We discussed this thing called 'structured data'
Yeah, they showed us records and lists
But even though we've been in class, we're still in the mist about it

You remember how we focused on variables holding numbers in the first few lectures?


You remember how we focused on variables holding numbers in the first few lectures?
Well, it looks like that is not really what most coding is about.

You have to define your own 'data types' and store them in large lists to process them


You have to define your own 'data types' and store them in large lists to process them
You do not use numbers to encode data; you mostly use records and lists

Buzz...

Hello Class !
Nice to mewt you.

C A T

C omputerized
A utomated
T utor

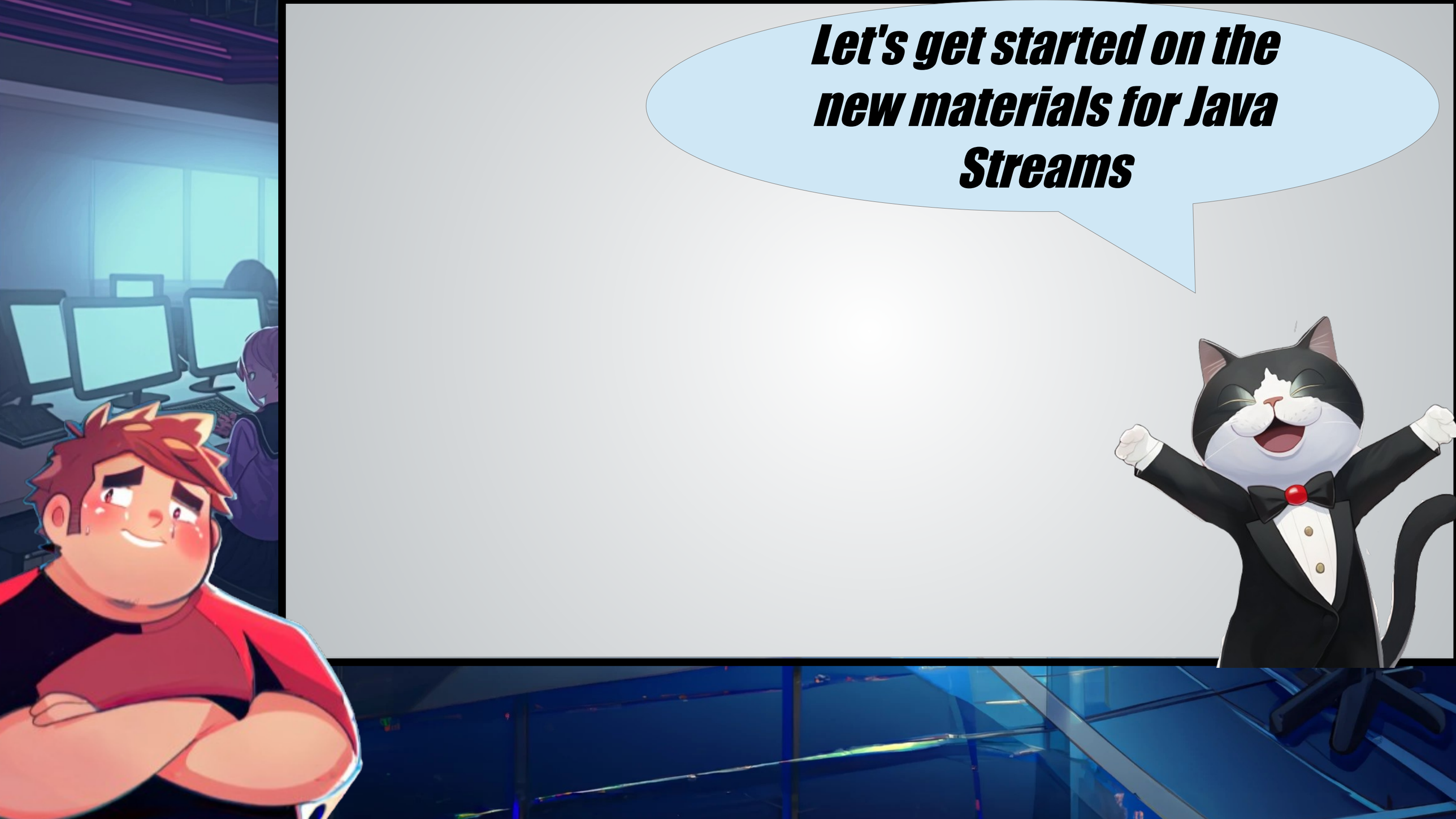
Welcome to your first lab session. I'm CAT, your Computerized Automated Tutor

C A T

C omputerized
A utomated
T utor

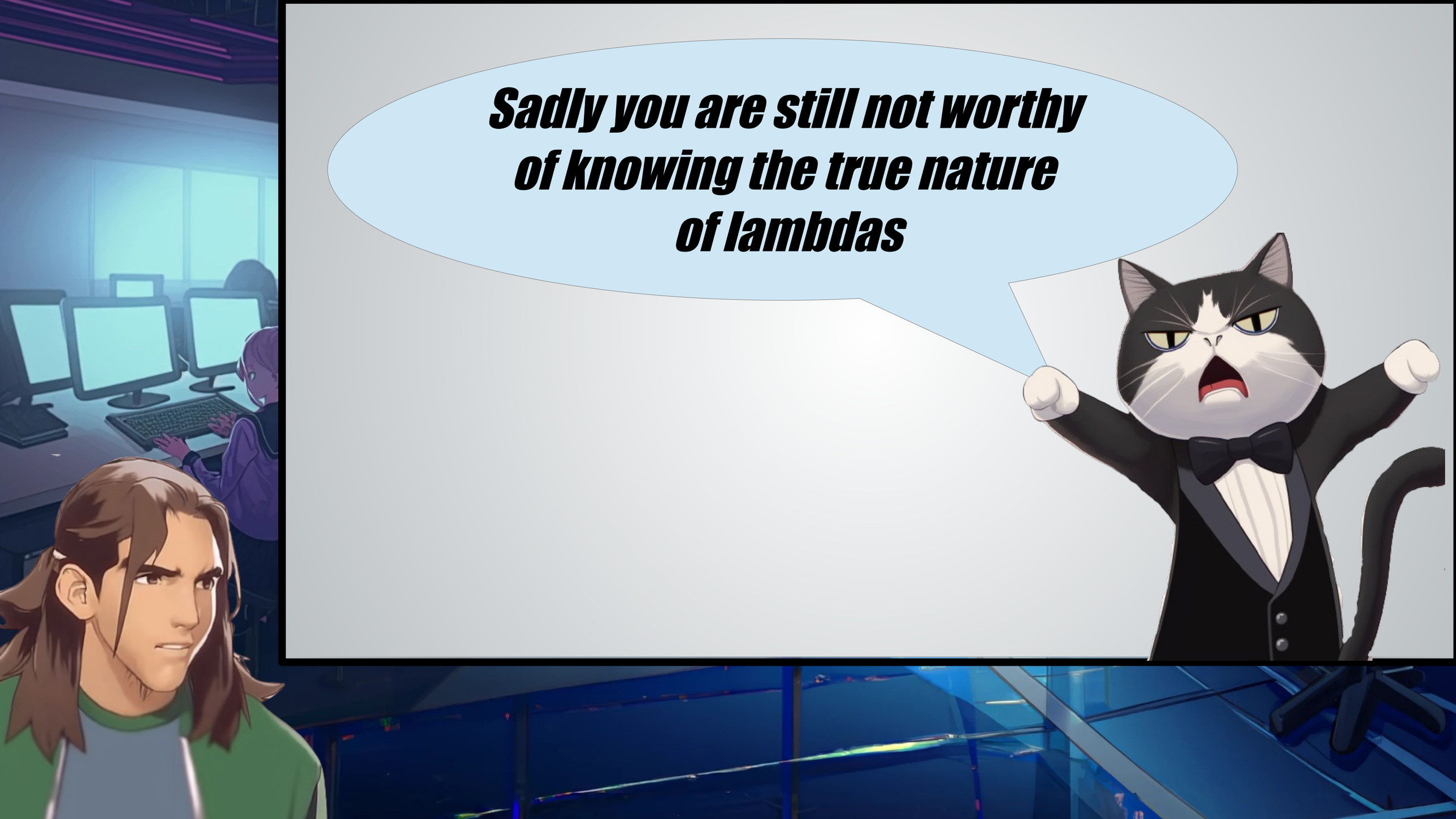
Welcome to your first lab session.
I'm CAT, your Computerized Automated Tutor

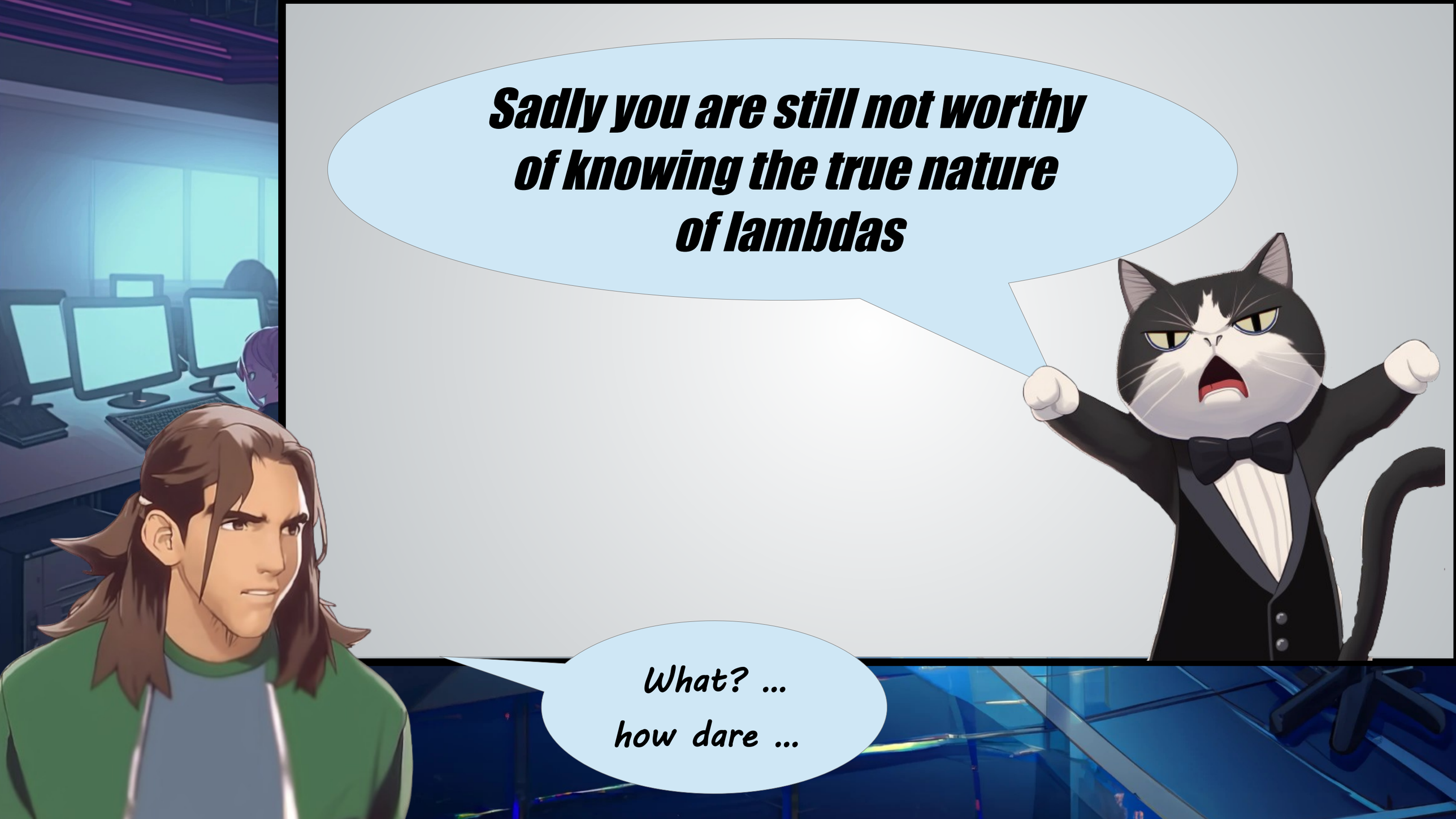
Today,
I will halp you thinking with streams


Let's get started on the new materials for Java Streams

Let's get started on the new materials for Java Streams
Java what now?

Streams are all about lambdas


Sadly you are still not worthy
of knowing the true nature
of lambdas


Sadly you are still not worthy of knowing the true nature of lambdas
What? ... how dare ...

So, for now yowl can just assume that they are a way to encapsulate instructions in a box

So, for now yowl can just assume that they are a way to encapsulate instructions in a box

A box that can be passed around without being instantly executed

Lambdas can have purrameters.

```
() -> System.out.println("Meaw!")
```

Here are some examples of lambdas:

```
() -> System.out.println("Meaw!")
```

Here are some examples of lambdas:

This lambda has zero parameters and will meaw when it is executed.

```
x -> x*4
```

This lambda has one parameter, x

```
x -> x*4
```

This lambda has one parameter, x

. . . and will return x times 4 when it is executed.

```
record Point(int x, int y){}


(x,y) -> new Point(x,y)
```

Assume a record Point as shown

```
record Point(int x, int y){}


(x,y) -> new Point(x,y)
```

Assume a record Point as shown

This lambda has two parameters: x and y. When executed it will return a point with x and y coordinates

```
record Point(int x, int y){}


(x,y) -> new Point(x,y)
```

And now, let's start with streams!

```
record Point(int x, int y){}


(x,y) -> new Point(x,y)
```

And now, let's start with streams!

So, that is the record!
It looks like it’s a way to group data with names

Streams are great
for working with
lists and collections

Streams are great for working with lists and collections

The main thing to keep in meawnd is the SIZE

Streams are great for working with lists and collections

The main thing to keep in meawnd is the SIZE

The input size and the output size

```
List<Person> persons = List.of(...);
List<String> names   = persons.stream()
   .######################
   .toList();
```

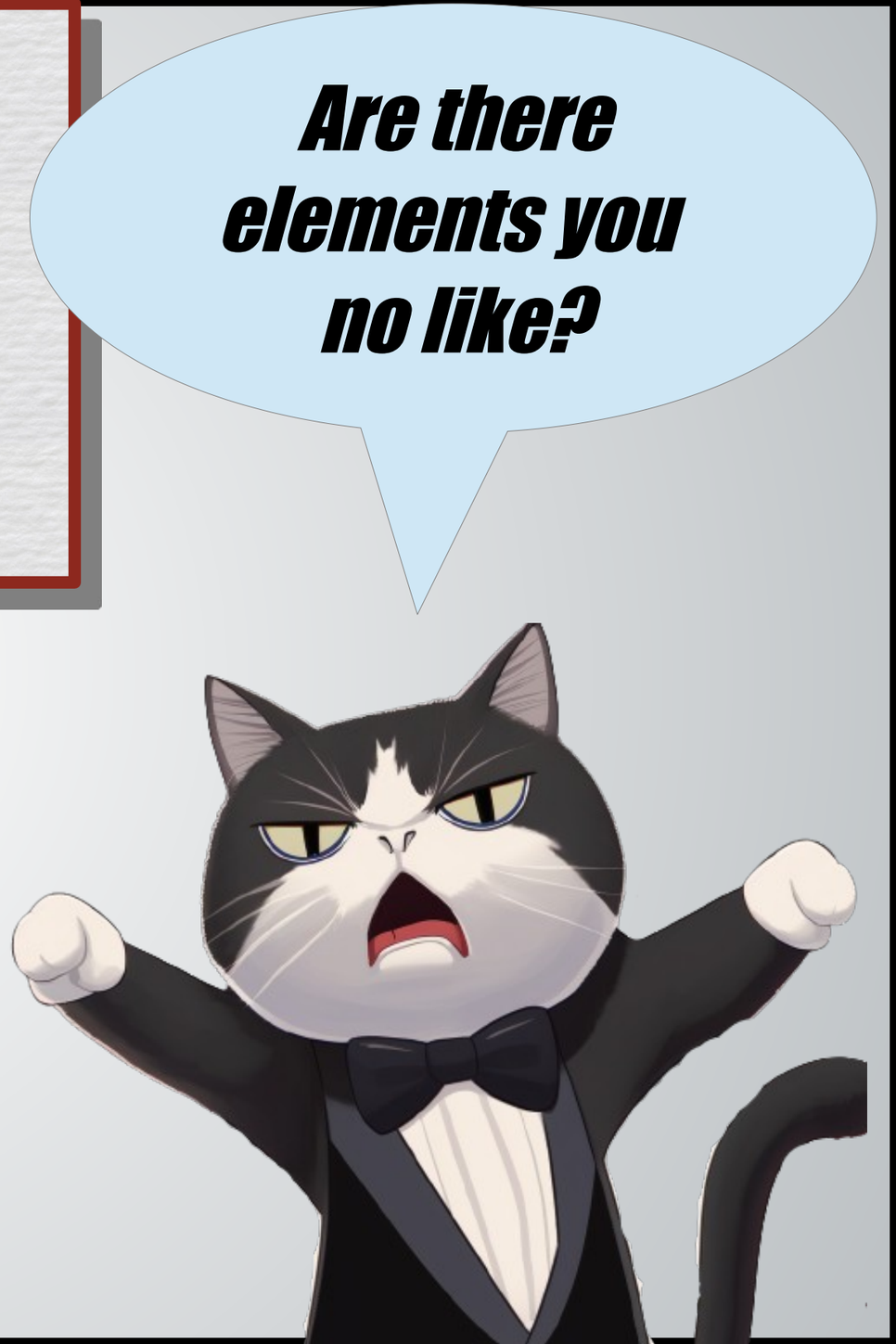
Are these lists the same size? Use map!

```
List<Person> persons = List.of(...);
List<String> names   = persons.stream()
   .map( p -> p.name )
   .toList();
```

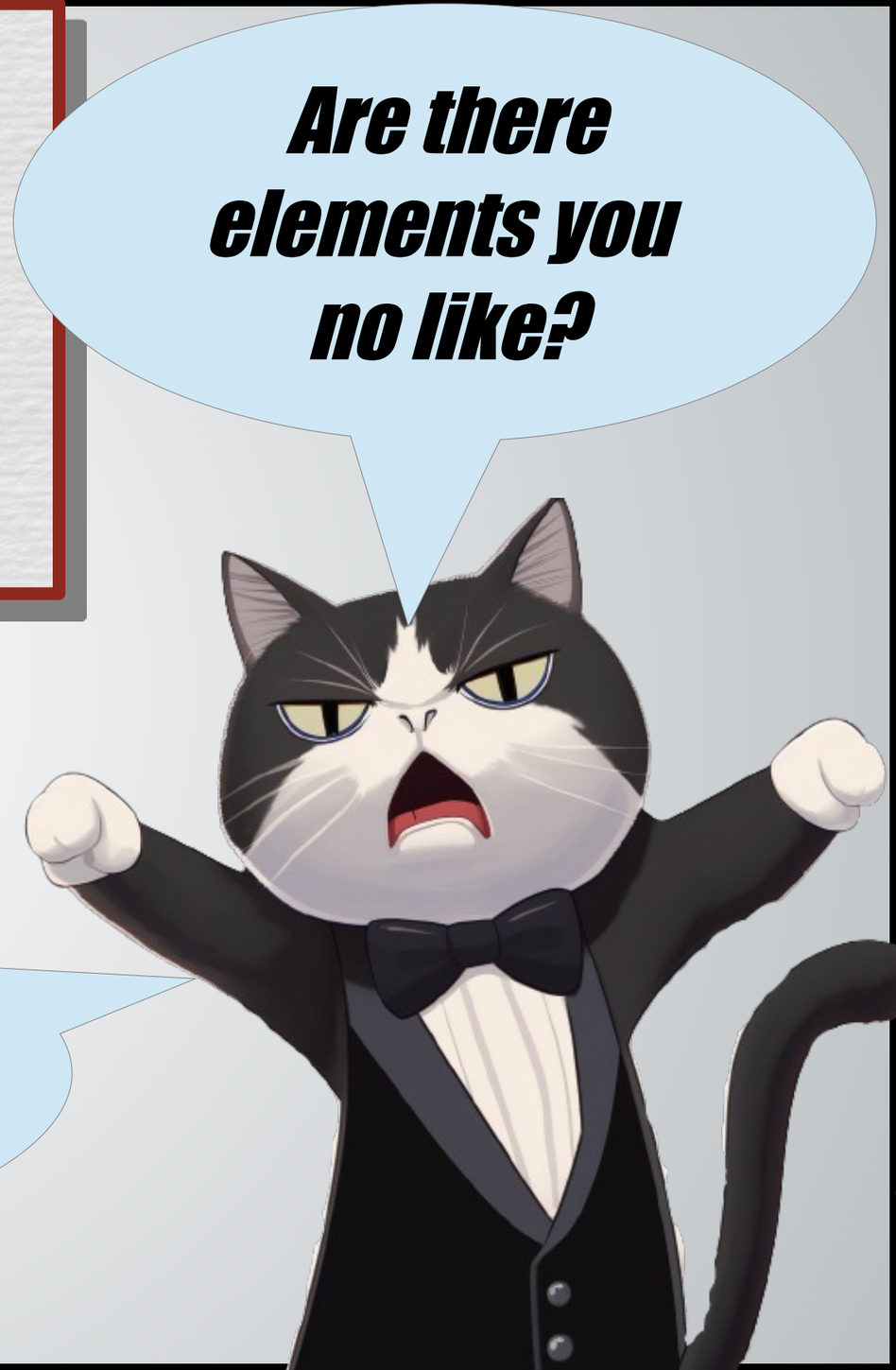
Are these lists the same size? Use map!

--.map -- keeps the same size. One element flows into one element

```
List<Person> persons = List.of(...);
List<Person> nices   = persons.stream()
   .######################
   .toList();
```



```
List<Person> persons = List.of(...);
List<Person> nices   = persons.stream()
   .filter( p -> iCanHasCheezburgerFrom(p) )
   .toList();
```



```
persons.stream()
   .####################
   .forEach( p -> cat.stealCheezburger(p) );

//forEach does an action for all the elements
```

```
persons.stream()
   .flatMap( p -> p.friends().stream() )
   .forEach( p -> cat.stealCheezburger(p) );

//forEach does an action for all the elements
//Here for all the friends of all the persons
```

You want MORE elements ?

-- .flatMap -- is your friend. Just specify a stream of elements to fit in place of a single element.

```
Optional<Person> owner = persons.stream()
   .max((p1,p2)->
     Double.compare(foodOf(p1), foodOf(p2)) );
```

```
Optional<Person> owner = persons.stream()
   .max((p1,p2)->
     Double.compare(foodOf(p1), foodOf(p2)) );
```

You want to get a single result?

Focus on terminal operations.

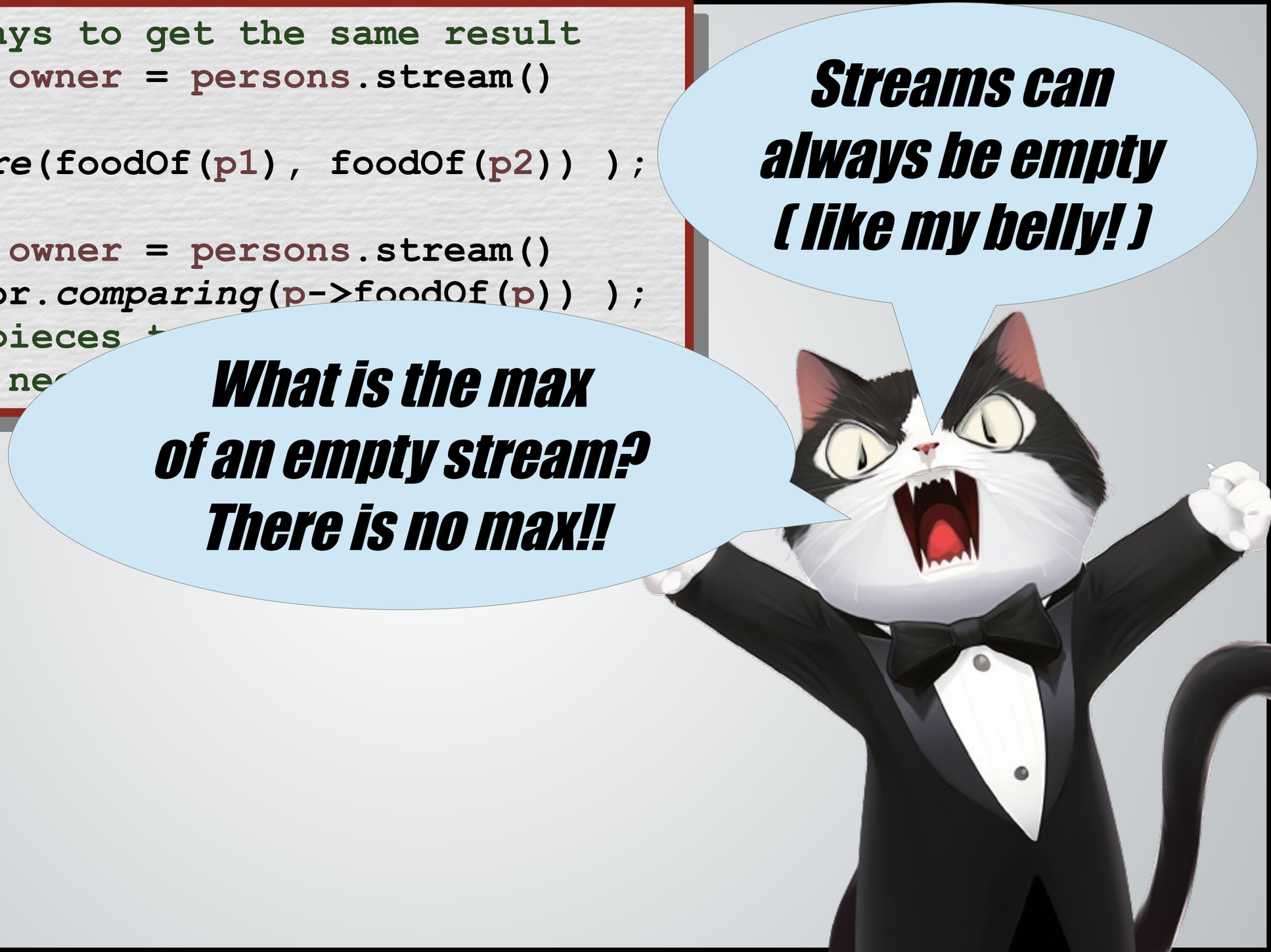
```
Optional<Person> owner = persons.stream()
   .max((p1,p2)->
     Double.compare(foodOf(p1), foodOf(p2)) );
```

You want to get a single result?

Focus on terminal operations. But be careful. This will push you out of streamy-land

```
//Two different ways to get the same result
 Optional<Person> owner = persons.stream()
    .max((p1,p2)->
      Double.compare(foodOf(p1), foodOf(p2)) );

 Optional<Person> owner = persons.stream()
    .max( Comparator.comparing(p->foodOf(p)) );
//Many funky API pieces to combine!
//Memorization is needed here...
```

```
//Two different ways to get the same result
 Optional<Person> owner = persons.stream()
    .max((p1,p2)->
      Double.compare(foodOf(p1), foodOf(p2)) );

 Optional<Person> owner = persons.stream()
    .max( Comparator.comparing(p->foodOf(p)) );
//Many funky API pieces t
//Memorization is nec
```



```
//Two different ways to get the same result
 Optional<Person> owner = persons.stream()
    .max((p1,p2)->
      Double.compare(foodOf(p1), foodOf(p2)) );

 Optional<Person> owner = persons.stream()
    .max( Comparator.comparing(p->foodOf(p)) );
//Many funky API pieces t
//Memorization is ne
```

Streams can always be empty (like my belly!)

What is the max of an empty stream? There is no max!!

That is why max/min/reduce/findFirst and meawny others give us Optional results!

```
boolean home = persons.stream()
   .anyMatch(p->atHome(p));
if(home){/*..*/}
```

```
boolean home = persons.stream()
   .anyMatch(p->atHome(p));
if(home){/*..*/}

assert persons.stream()
   .anyMatch(p->atHome(p));
//Great for asserts!
```

You want to take decisions?

Terminal operations again! Look for --.anyMatch-- and its friends!

```
Optional<Person> friend = persons.stream()
   .######( p -> atHome(p) )
   .###########;
```

```
Optional<Person> friend = persons.stream()
    .filter( p -> atHome(p) )
    .findFirst();
//Optimized:
//  filters until it finds the first
//  and then stops exploring the list
```

To find elements use
-- .filter -- and -- .findFirst --

```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
```

```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
    .###########
    .###########
    .toList();
//'Zipping' two collections. Just use the indexes
//Long stream chains are GOOD.
//Gives the library many optimization options!
```

Down to the indexes!

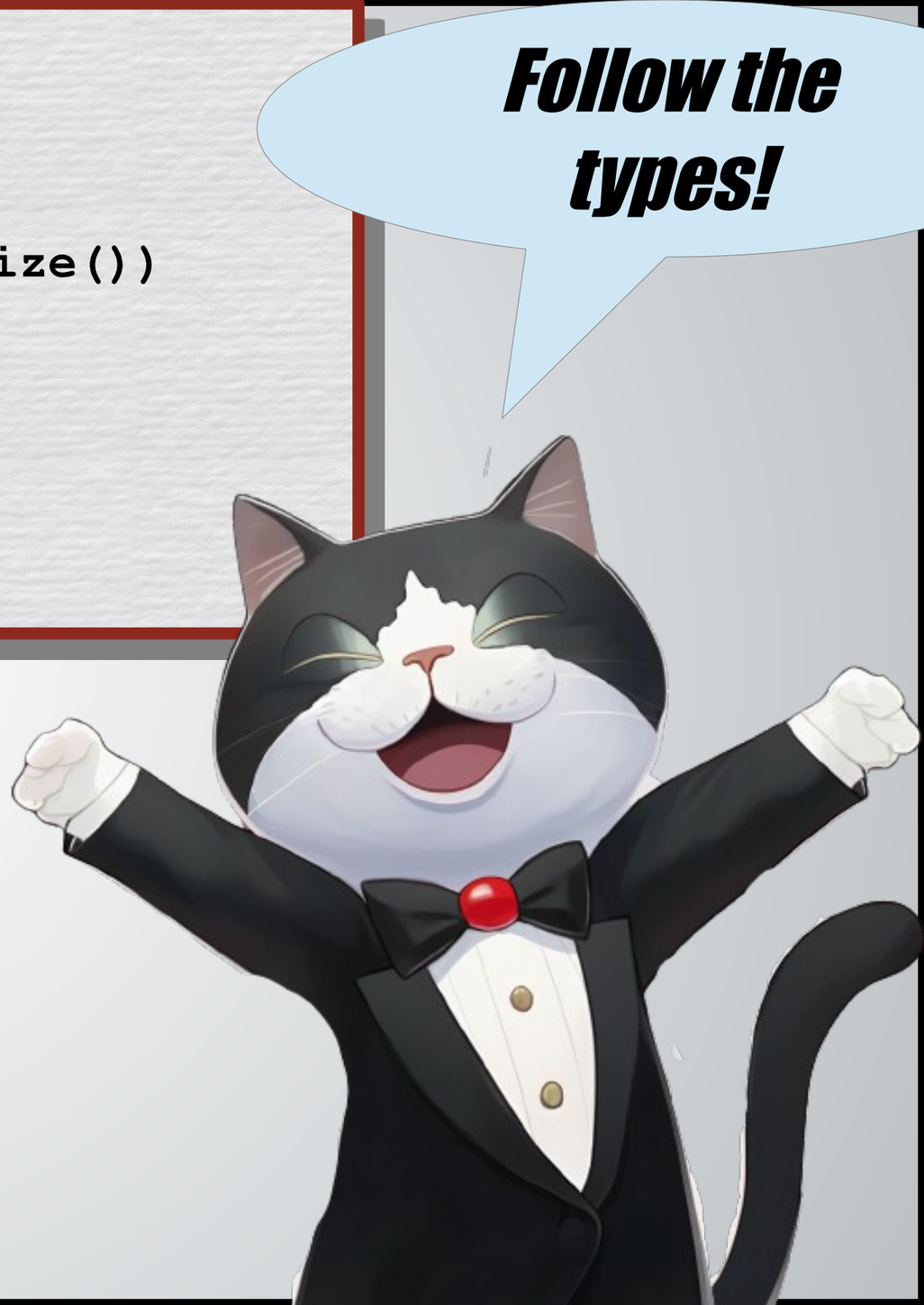
-- .range -- is your way out from many sticky situations!

```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
   .mapToObj(i->new Person(names.get(i), surnames.get(i)))
   .map( p->coerceFoodFrom(p) )
   .toList();
//'Zipping' two collections. Just use the indexes
//Long stream chains are GOOD.
//Gives the library many optimization options!
```

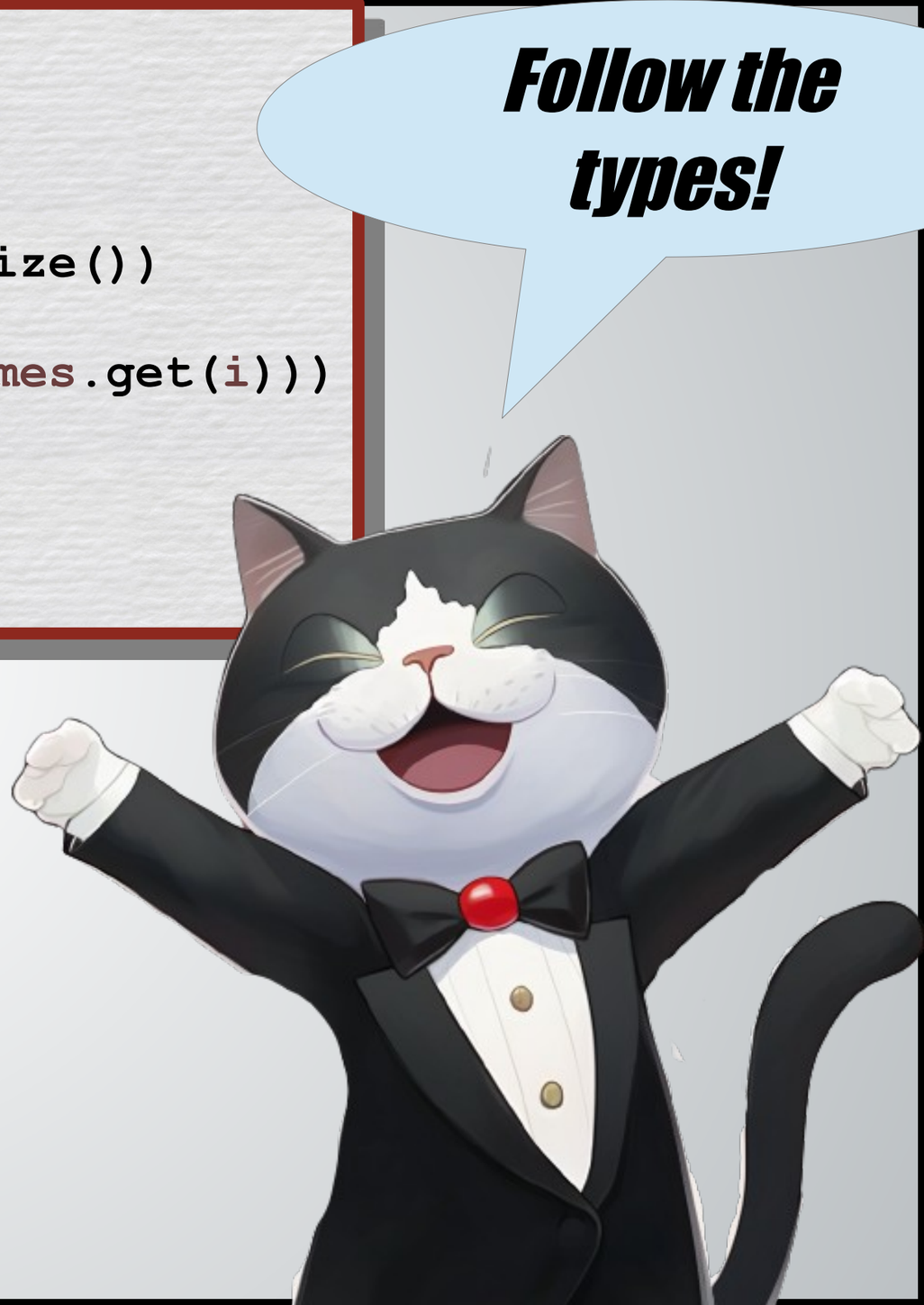
Down to the indexes!

-- .range -- is your way out from many sticky situations!

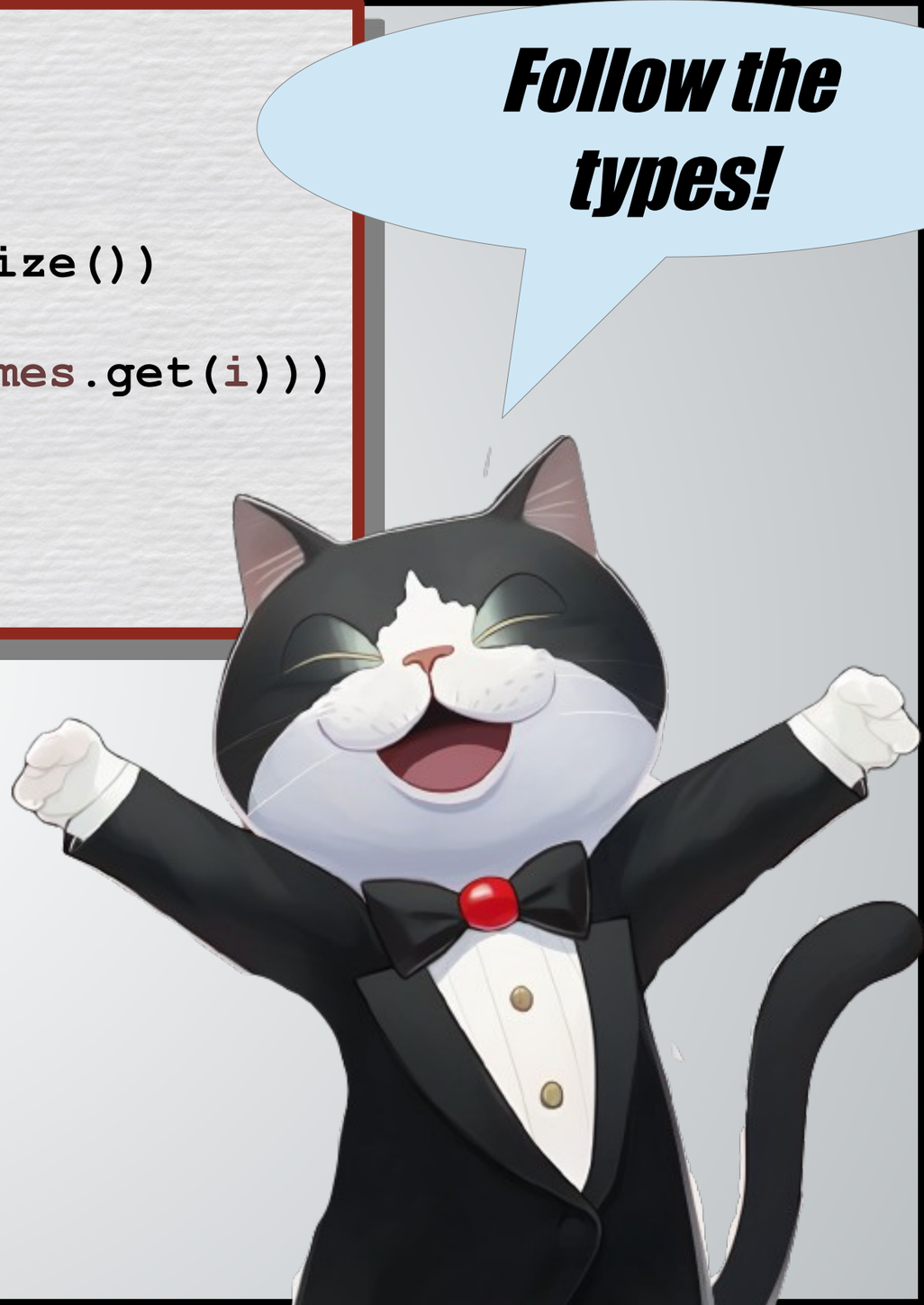
```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
  //IntStream ~= Stream<Integer>
```



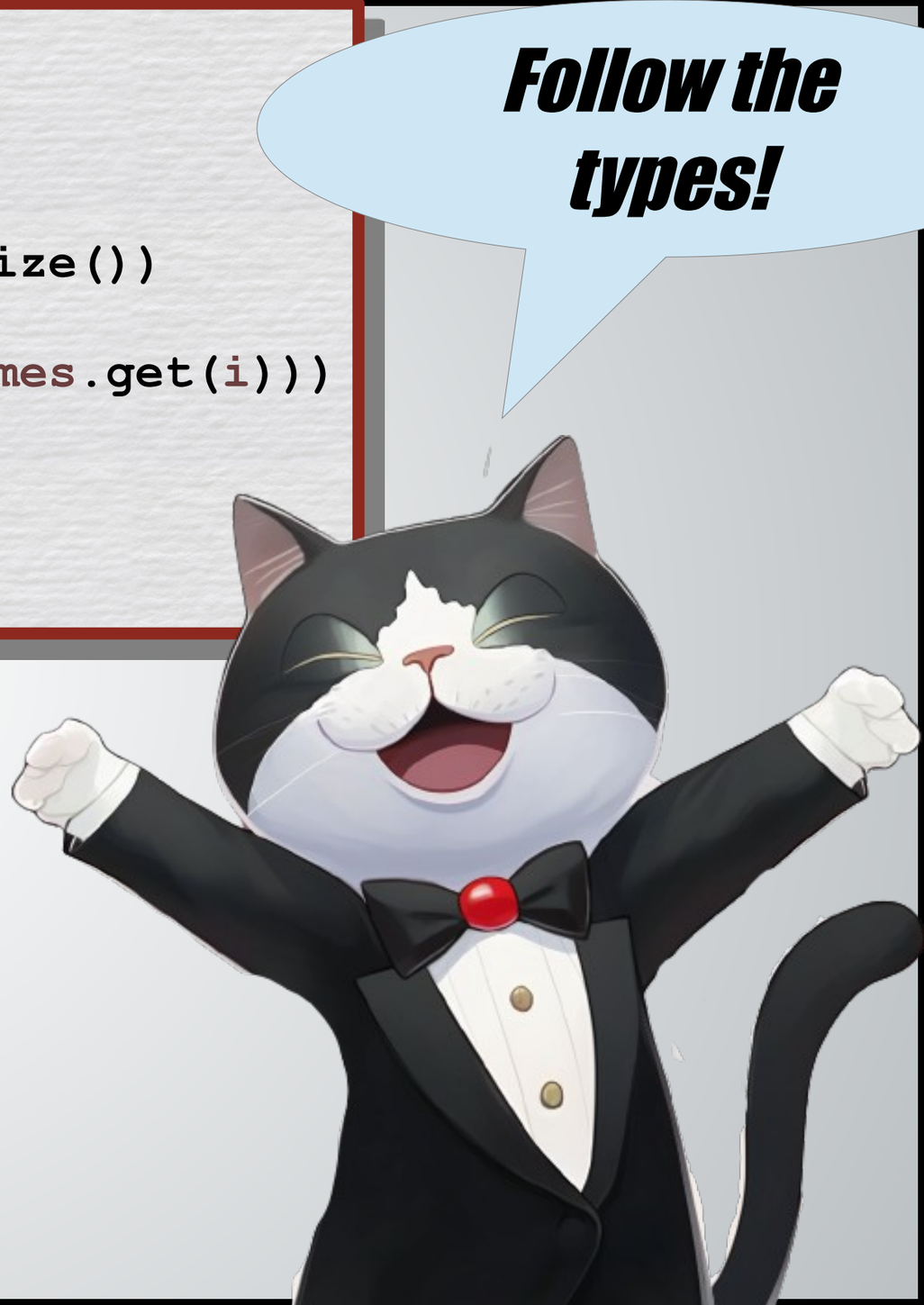
```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
  //IntStream ~= Stream<Integer>
  .mapToObj(i->new Person(names.get(i), surnames.get(i)))
  //Stream<Person>
```



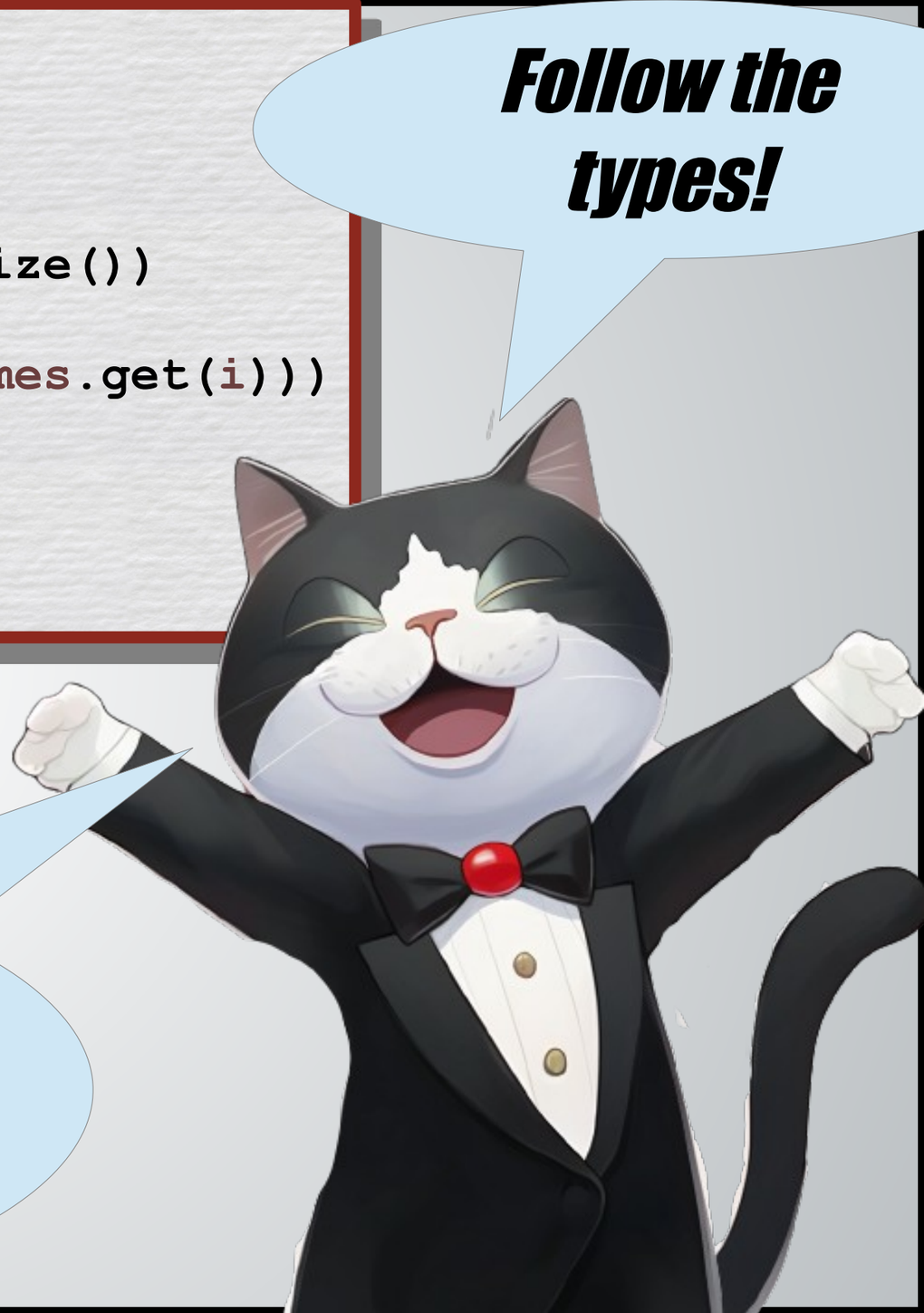
```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
  //IntStream ~= Stream<Integer>
  .mapToObj(i->new Person(names.get(i), surnames.get(i)))
  //Stream<Person>
  .map( p->coerceFoodFrom(p) )
  //Stream<Food>
```



```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
  //IntStream ~= Stream<Integer>
  .mapToObj(i->new Person(names.get(i), surnames.get(i)))
  //Stream<Person>
  .map( p->coerceFoodFrom(p) )
  //Stream<Food>
  .toList();
```



```
List<String> names = List.of();
List<String> surnames = List.of();
assert names.size() == surnames.size();
List<Food> yummy = IntStream.range(0, names.size())
  //IntStream ~= Stream<Integer>
  .mapToObj(i->new Person(names.get(i), surnames.get(i)))
  //Stream<Person>
  .map( p->coerceFoodFrom(p) )
  //Stream<Food>
  .toList();
```



Dany, Ricky, and Tommy are struggling

Dany, Ricky, and Tommy are struggling
I'm never going to pass this class

Dany, Ricky, and Tommy are struggling
I'm never going to pass this class
What does this even mean?

Dany, Ricky, and Tommy are struggling
I'm never going to pass this class
What does this even mean?
I'm sooo lost

```
.stream  //list or set into stream
```

```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
```

```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
```

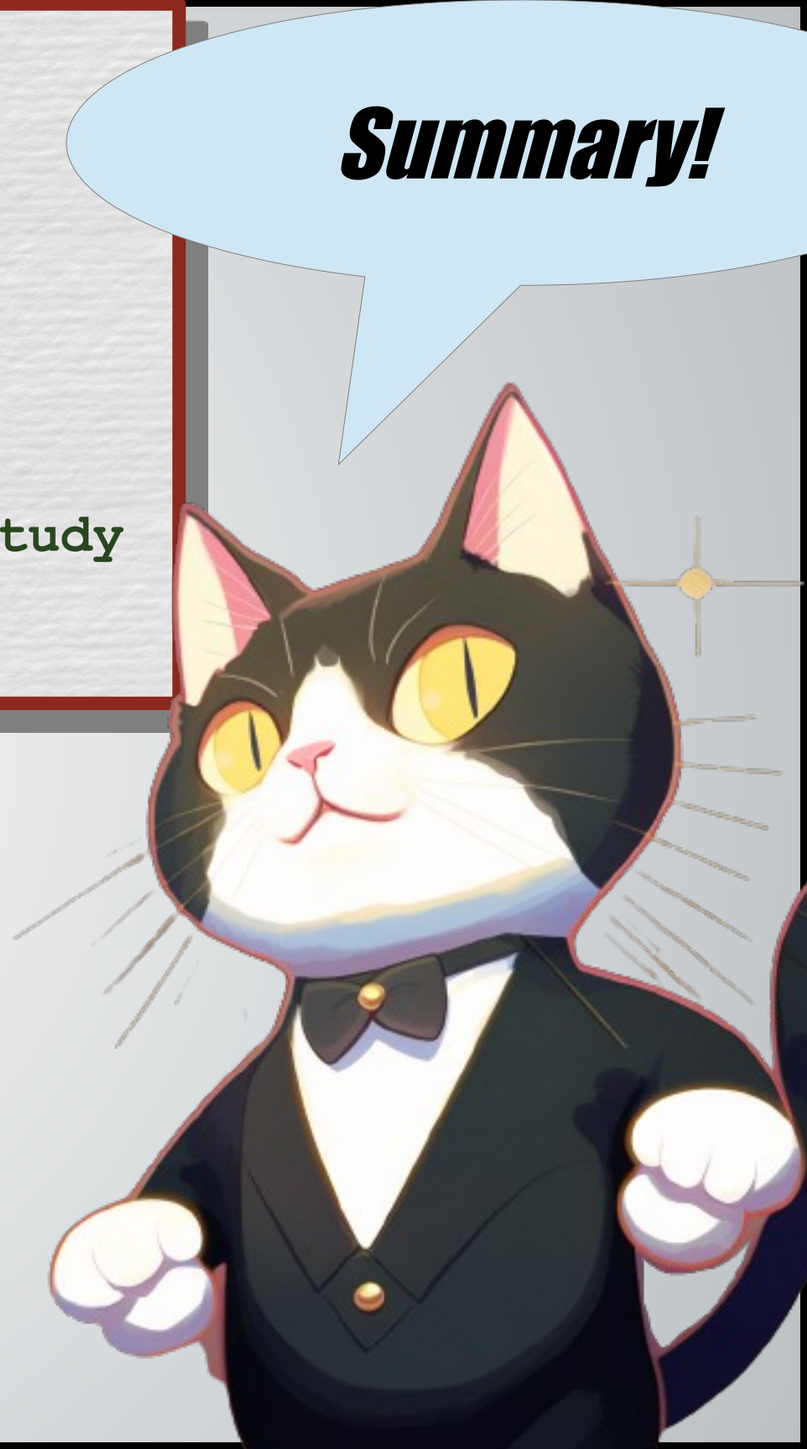
```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
.flatMap //1 to 0..n transformations
```

```
.stream   //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map      //1 to 1 transformations
.flatMap  //1 to 0..n transformations
.filter   //take some stuff out of the pipeline
```

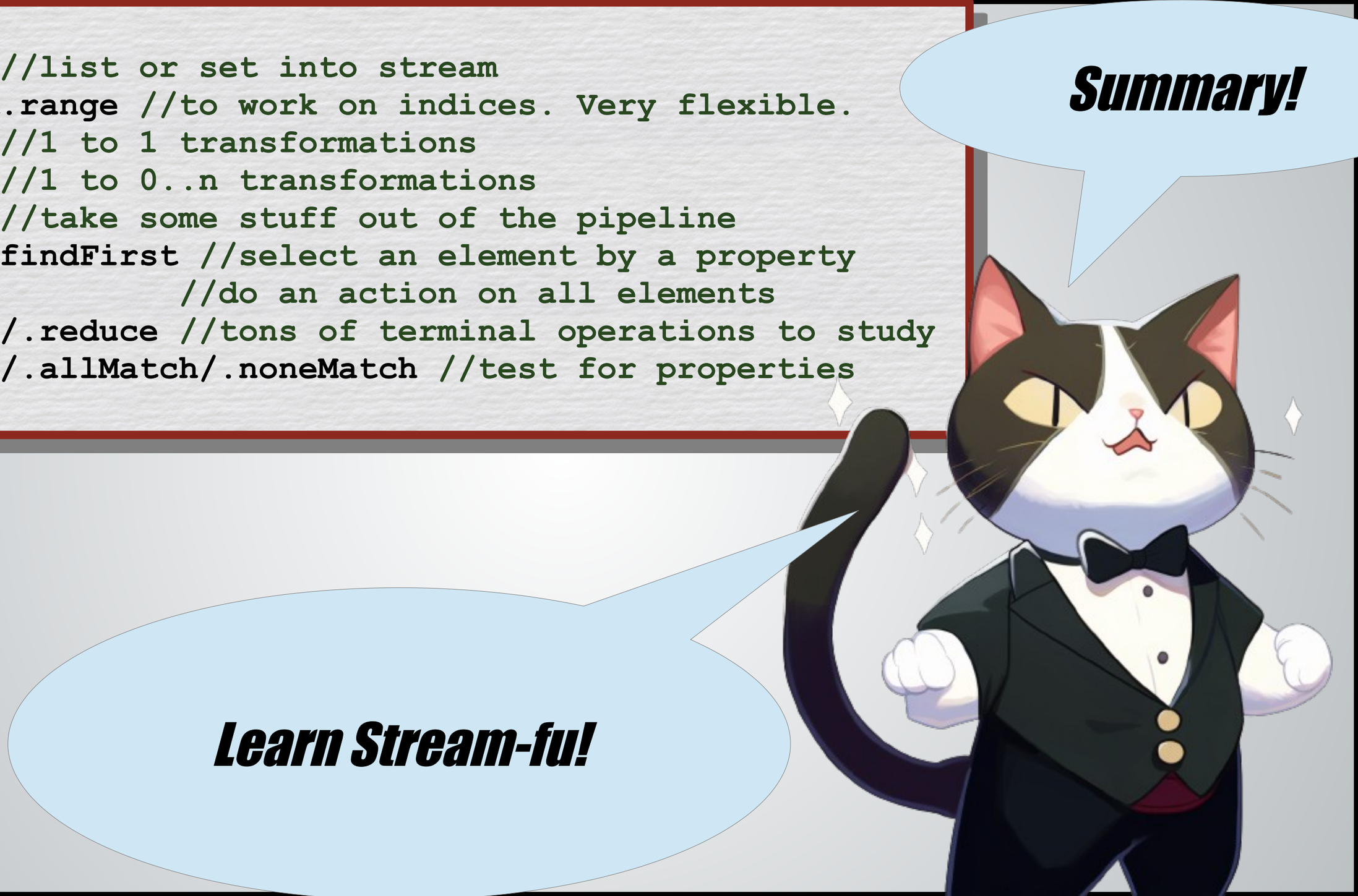
```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
.flatMap //1 to 0..n transformations
.filter  //take some stuff out of the pipeline
.filter+.findFirst //select an element by a property
```

```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
.flatMap //1 to 0..n transformations
.filter  //take some stuff out of the pipeline
.filter+.findFirst //select an element by a property
.forEach           //do an action on all elements
```

```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
.flatMap //1 to 0..n transformations
.filter  //take some stuff out of the pipeline
.filter+.findFirst //select an element by a property
.forEach           //do an action on all elements
.max/.min/.reduce //tons of terminal operations to study
```

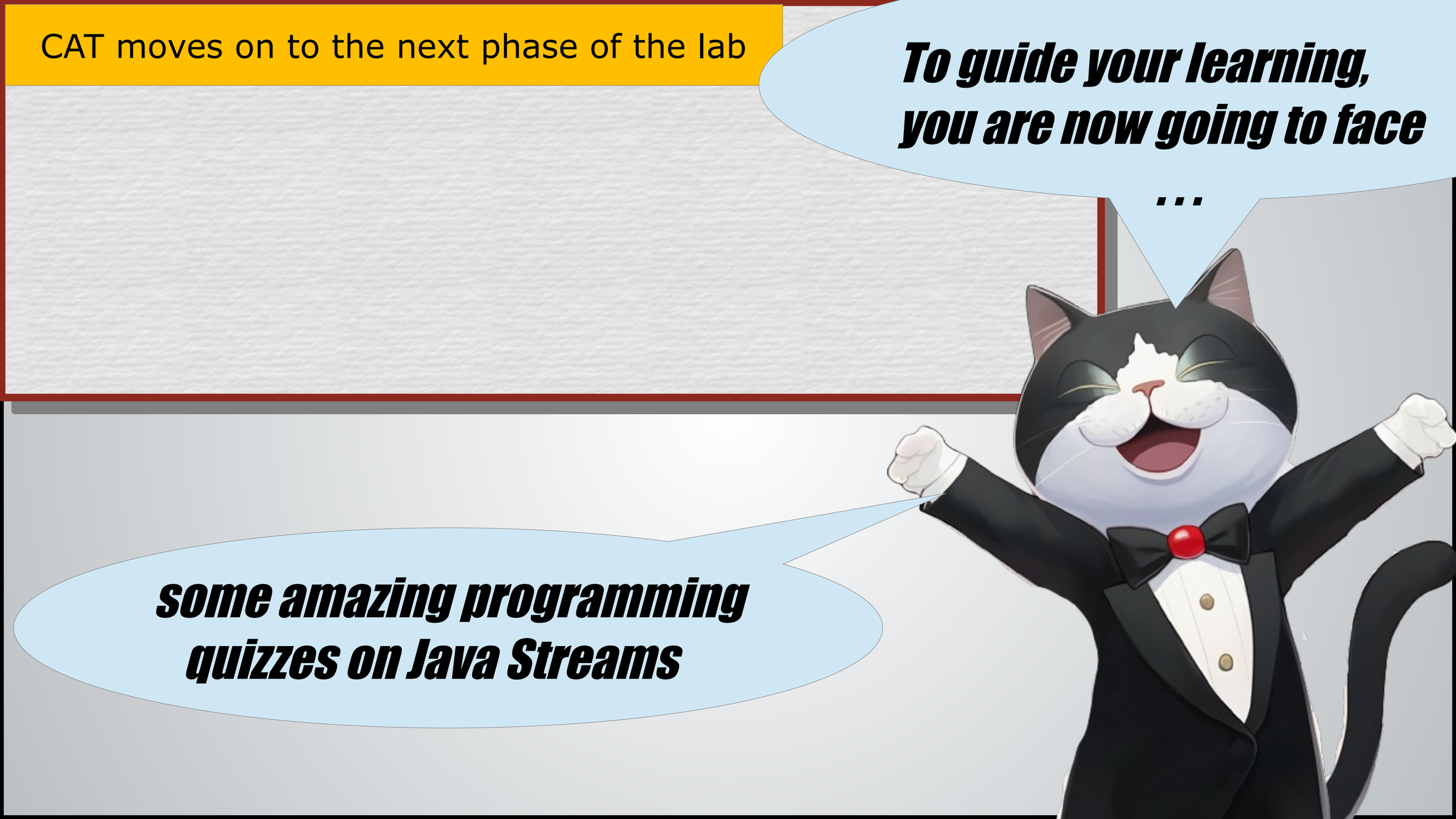


```
.stream  //list or set into stream
IntStream.range //to work on indices. Very flexible.
.map     //1 to 1 transformations
.flatMap //1 to 0..n transformations
.filter  //take some stuff out of the pipeline
.filter+.findFirst //select an element by a property
.forEach           //do an action on all elements
.max/.min/.reduce //tons of terminal operations to study
.anyMatch/.allMatch/.noneMatch //test for properties
```



CAT moves on to the next phase of the lab

CAT moves on to the next phase of the lab

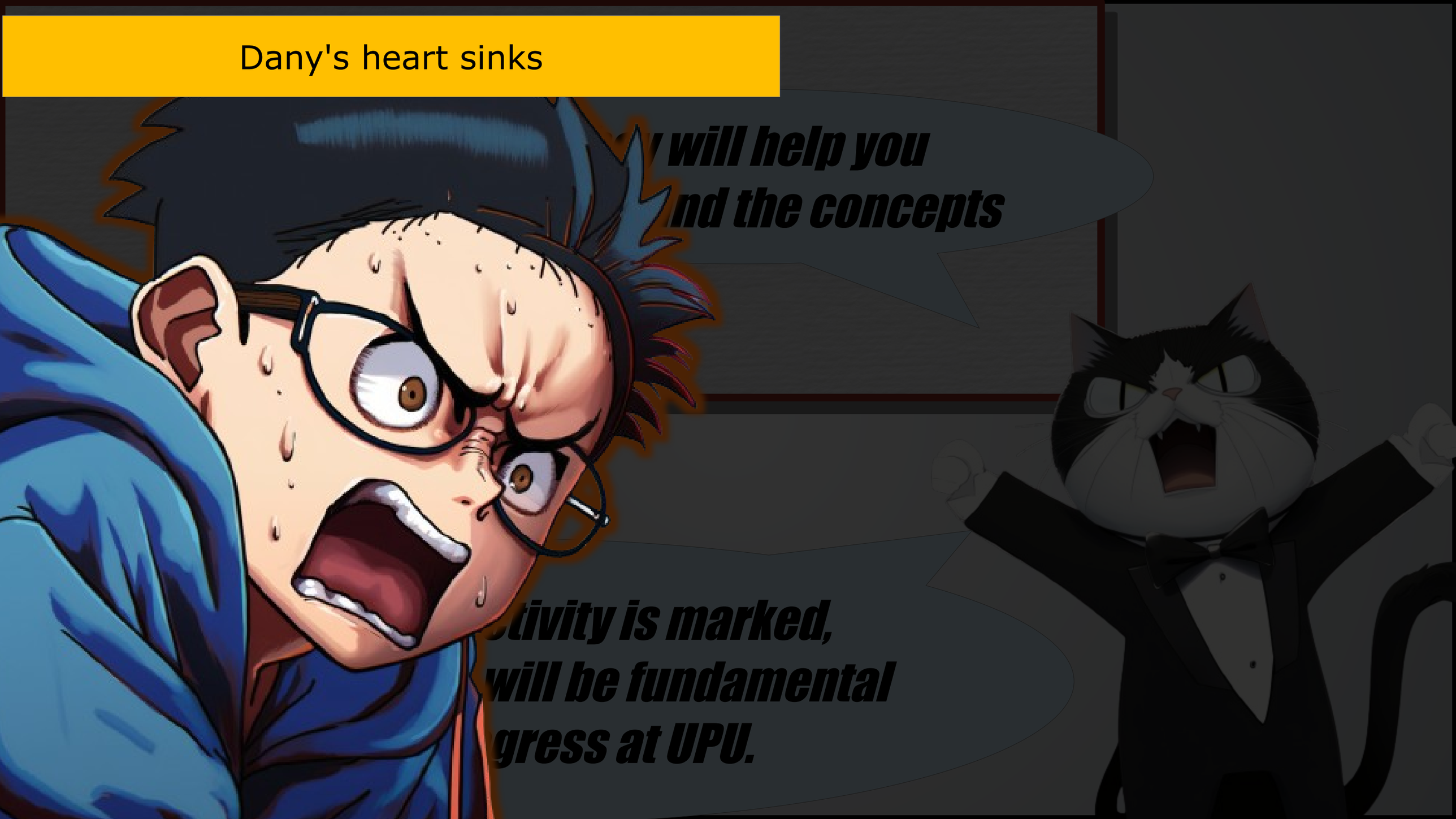


They will help you understand the concepts

They will help you understand the concepts

Of course this activity is marked, and a good result will be fundamental for your progress at UPU.

Dany's heart sinks



Ricky and Tommy are determined to survive this new challenge
Let's do it!

Ricky and Tommy are determined to survive this new challenge
Let's do it!
it is not as if we have any alternative!

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

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

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

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

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

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

to complete this Java method

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

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

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

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

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

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

```
.get()
```

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

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

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

```
.get()
```

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

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

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

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

```
.get()
```

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

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

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

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

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

```
.get()
```

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

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

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

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

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

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

```
.get()
```

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

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

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

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

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

```
.findFirst()
```

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

```
.get()
```

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

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

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

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

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

```
.findFirst()
```

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

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

```
.get()
```

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

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

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

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

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

```
.findFirst()
```

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

```
.filter(s -> s.height() > 6)
```

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

```
.get()
```

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

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

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

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

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

```
.findFirst()
```

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

```
.filter(s -> s.height() > 6)
```

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

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

```
.get()
```

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

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

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

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

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

```
.findFirst()
```

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

```
.filter(s -> s.height() > 6)
```

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

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

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

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

```
.get()
```

```
return
```

```
.max(
```

```
.forE
```

```
.find
```

```
.redu
```

```
.filt
```

Pause the video. Can you solve this one?

Note: not all snippets are needed.

Yes, you can not drag and drop code on a paused youtube video, but you can imagine doing it. Similar visualization exercises have been shown to be very effective at improving programming.

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

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

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

```
.get()
```

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

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

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

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

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

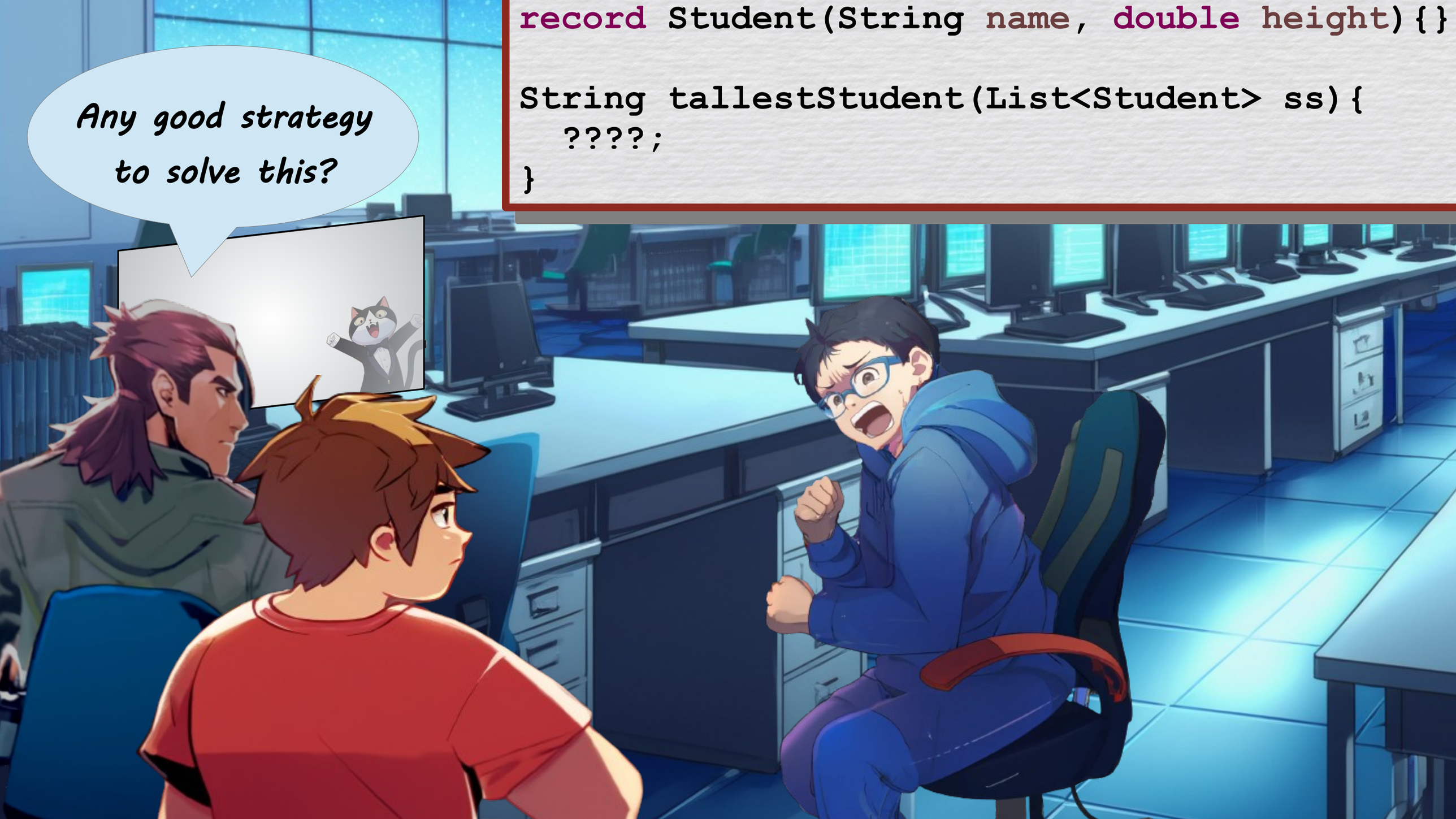
```
.findFirst()
```

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

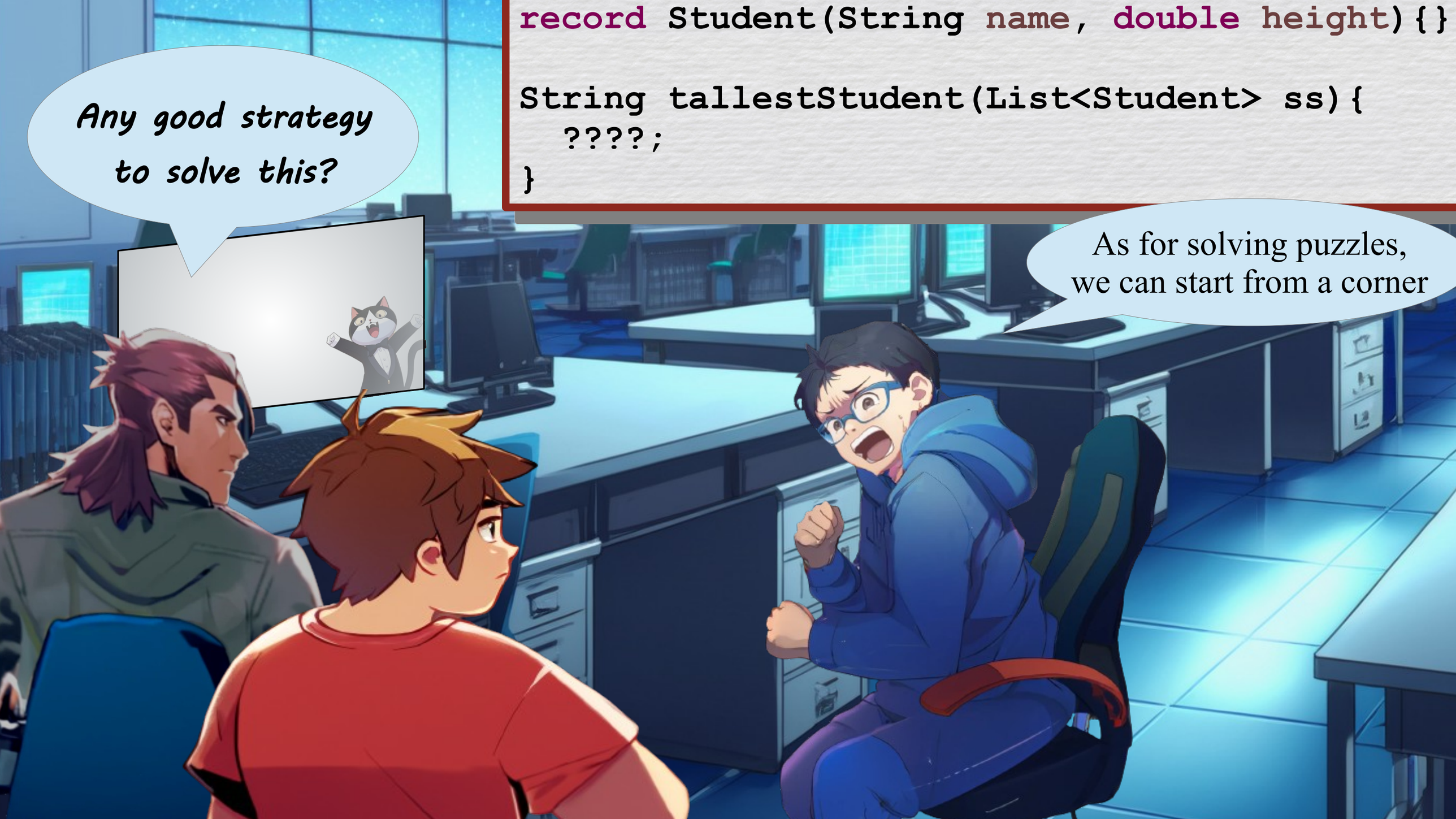
```
.filter(s -> s.height() > 6)
```

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

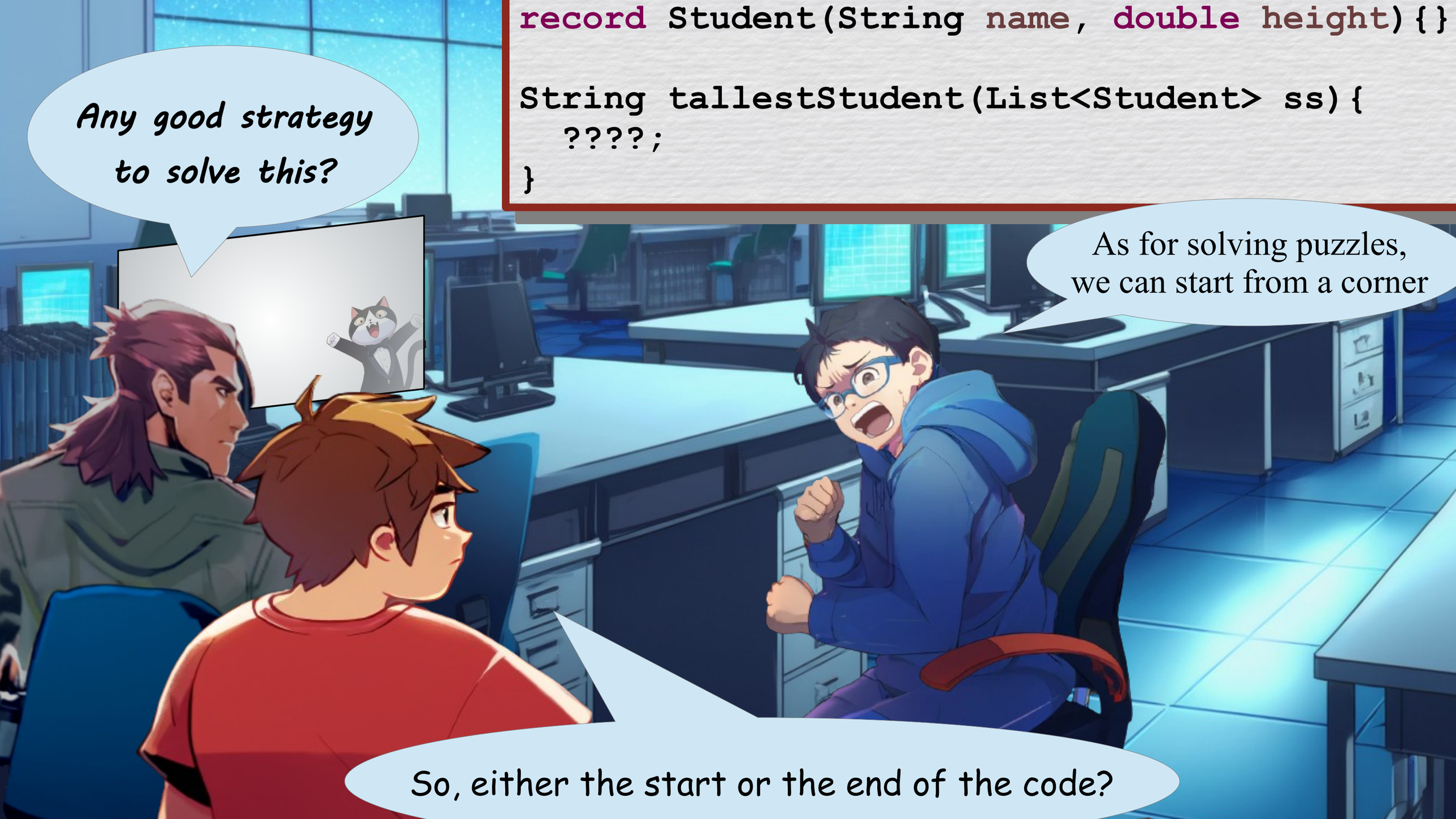
```
record Student(String name, double height){}

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

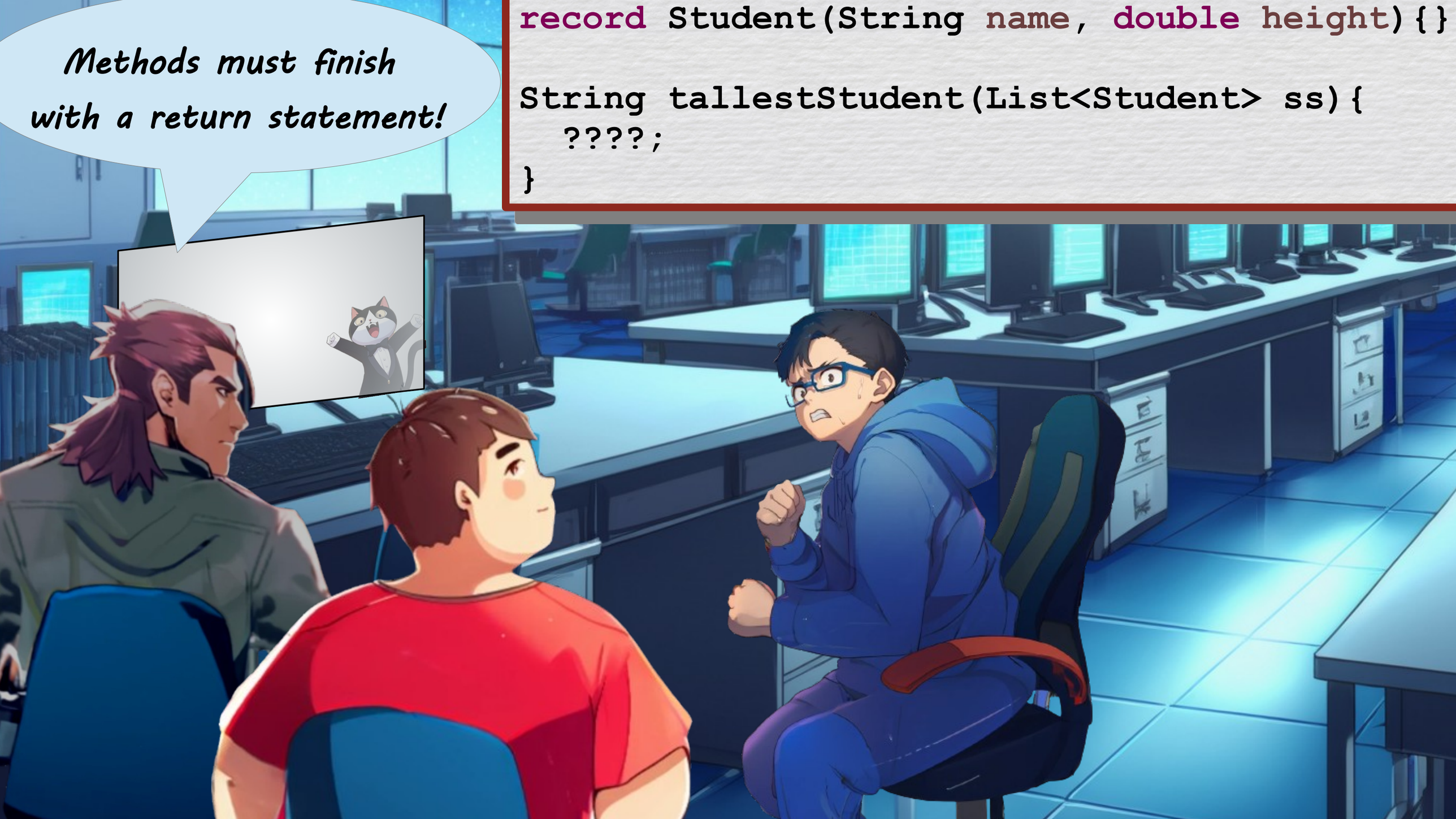


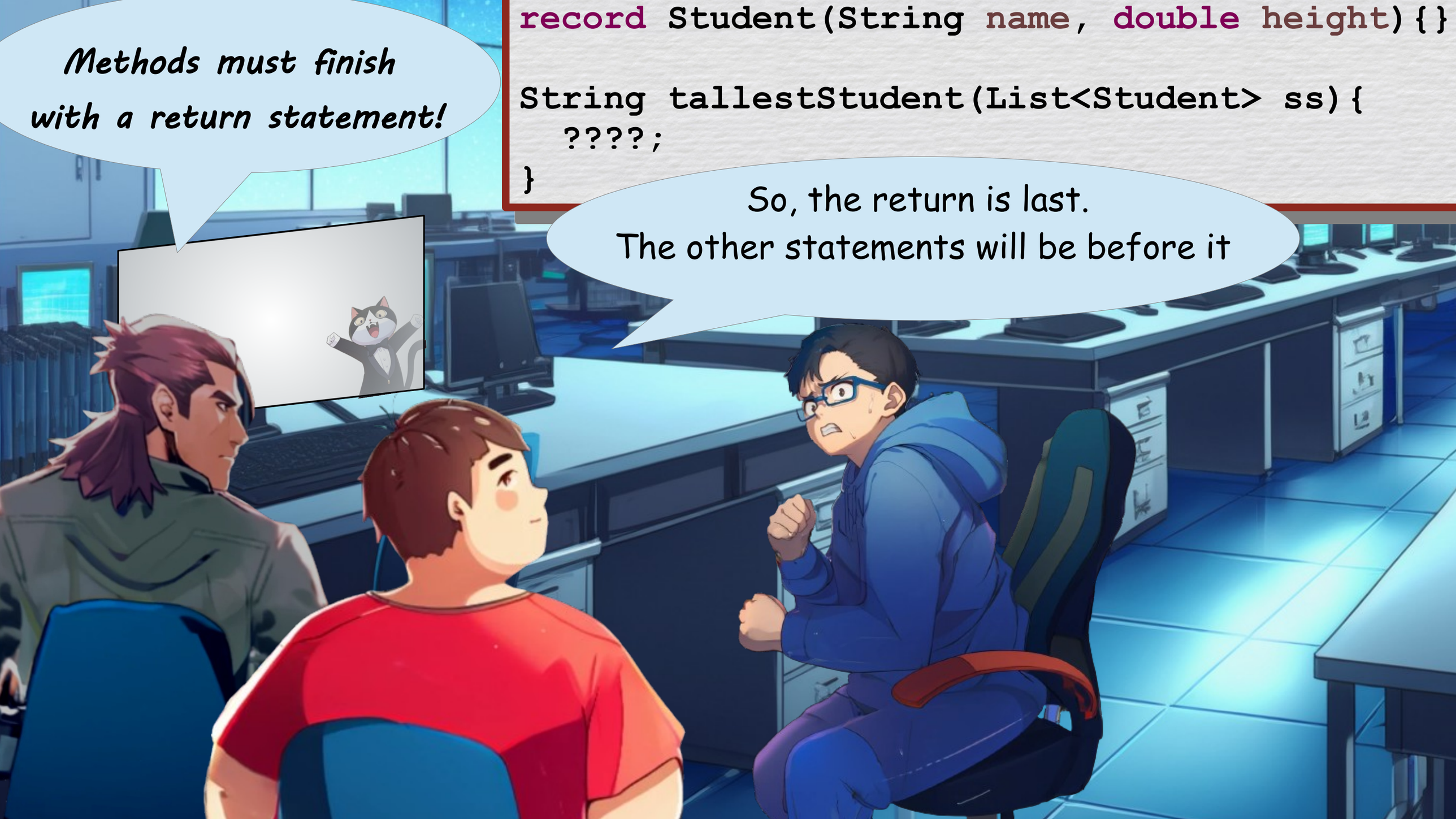
```
record Student(String name, double height){}

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

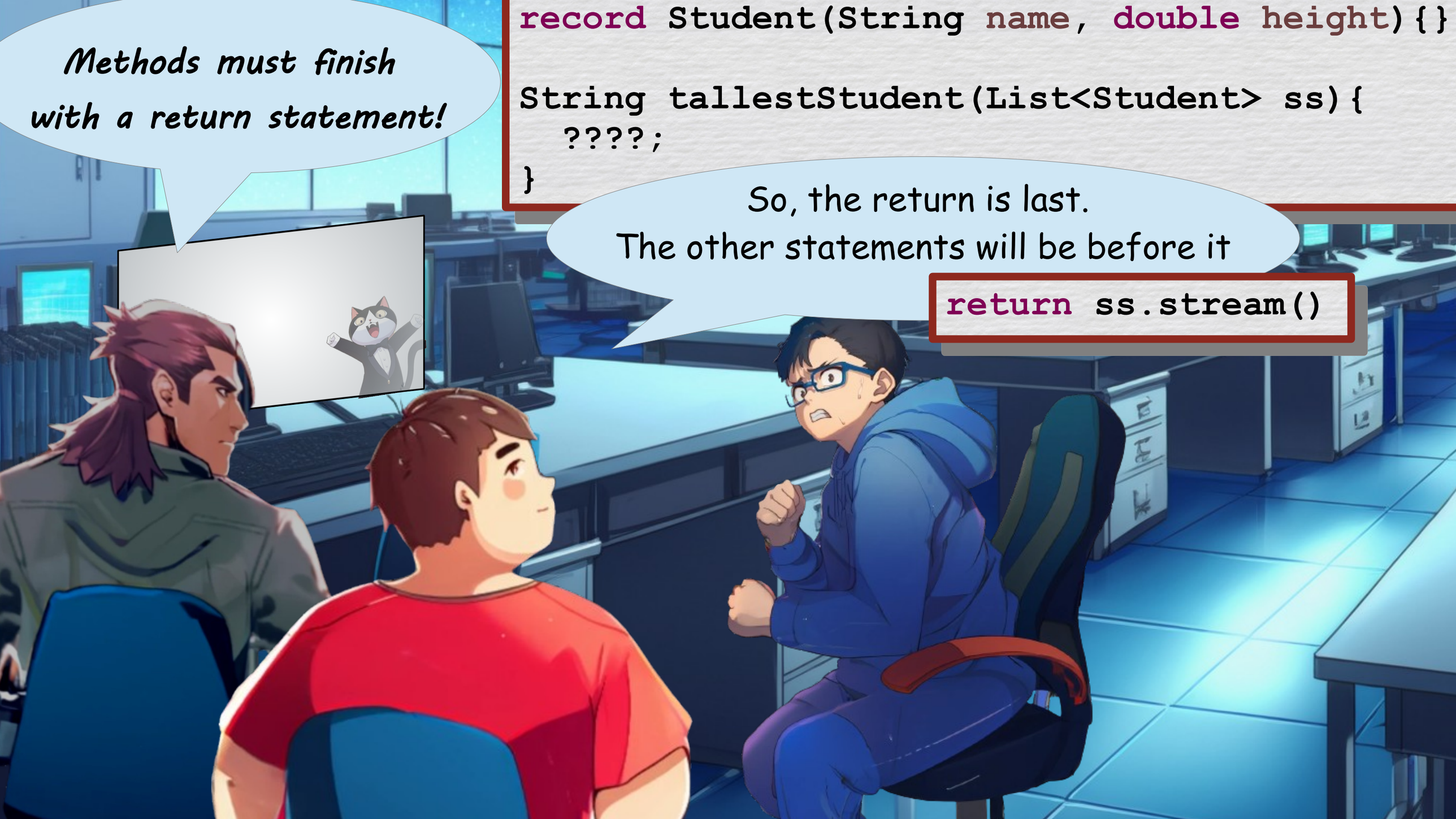


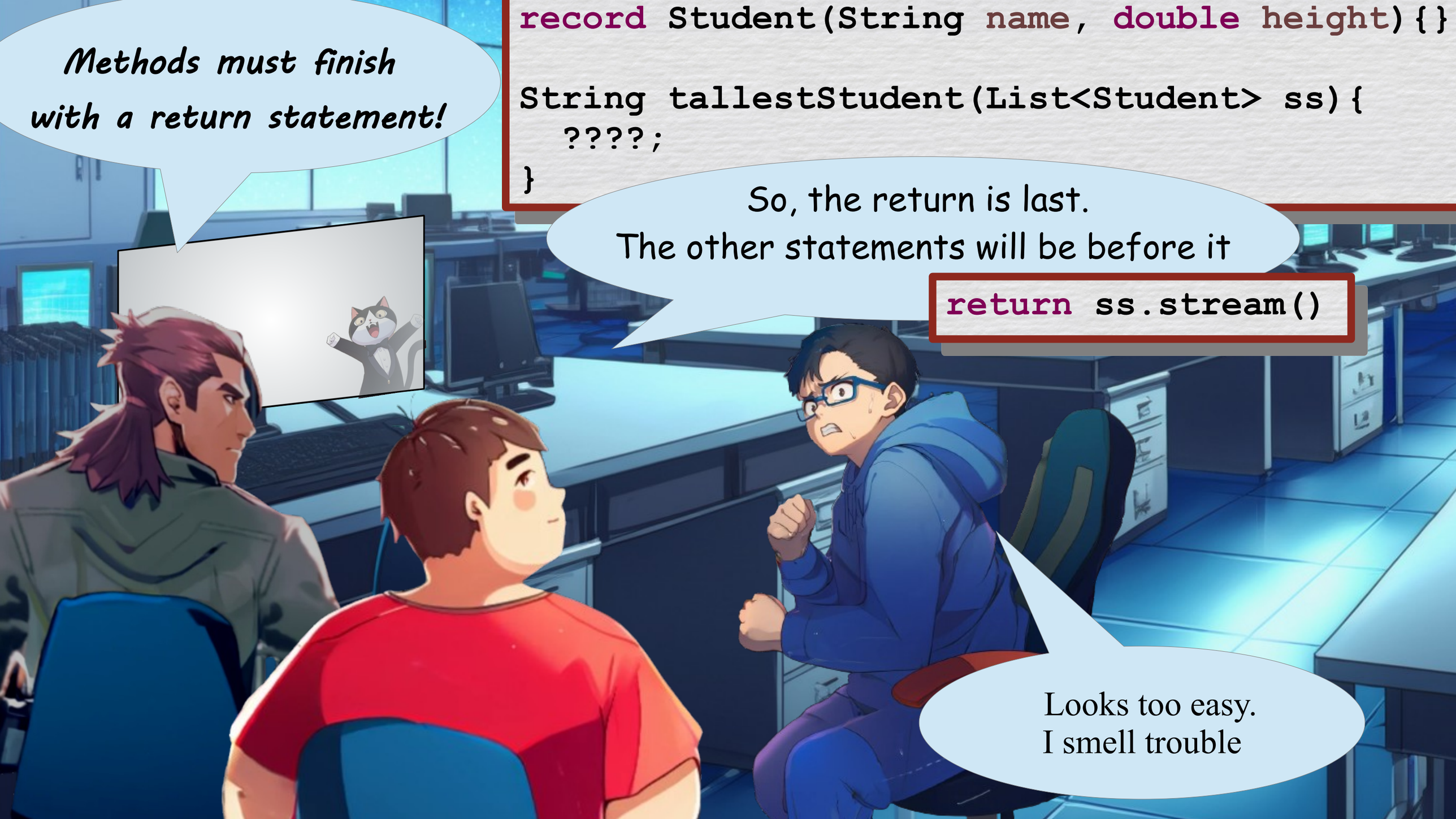
```
record Student(String name, double height){}

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




Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So, the return is last.
The other statements will be before it


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So, the return is last.
The other statements will be before it
return ss.stream()


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So, the return is last.
The other statements will be before it
return ss.stream()
Looks too easy.
I smell trouble

Looks too easy.
I smell trouble

Looks too easy.
I smell trouble
Do not be so negative all the time!
Return is the last statement!
return ss.stream()

Looks too easy.
I smell trouble
Do not be so negative all the time!
Return is the last statement!
return ss.stream()
Statements finish
with semicolons;

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

```
.get()
```

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

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

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

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

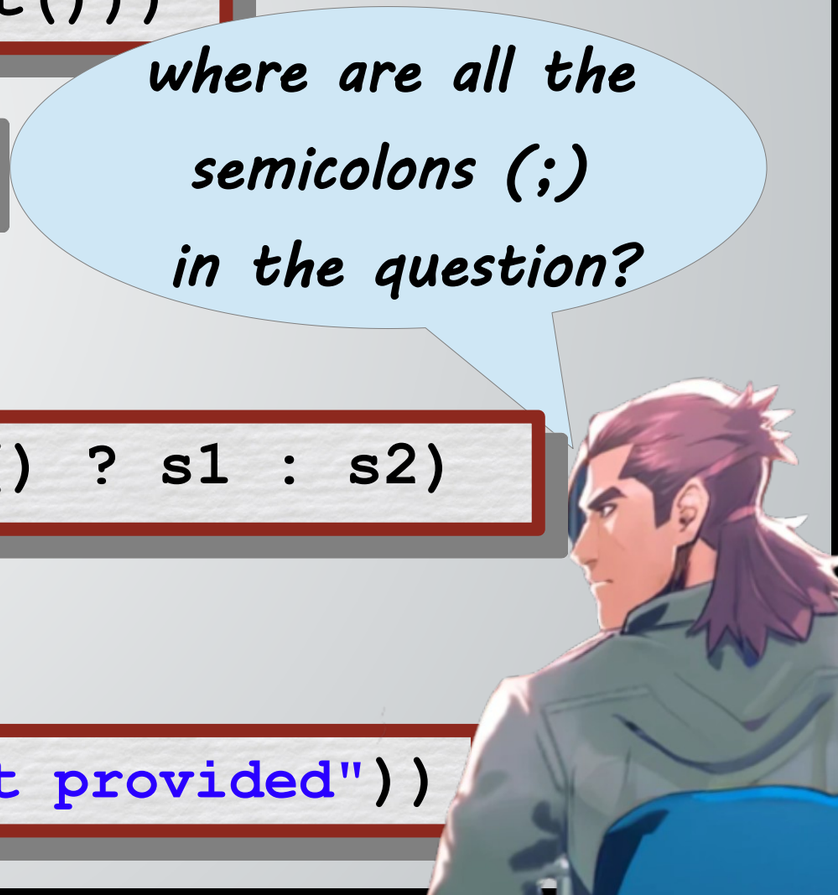
```
.forEach(s -> System.out.println(s.name()))
```

```
.findFirst()
```

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

```
.filter(s -> s.height() > 6)
```

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



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

```
.get()
```

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

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

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

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

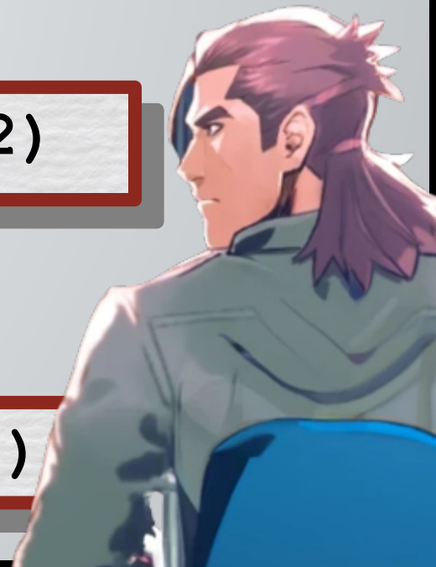
```
.  tem.out.println(s.name()))
```

```
-> s1 height  ) ? s1 : s2)
```

```
.height() > 6)
```

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

The only semicolon is already in the method body



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

```
.get()
```

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

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

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

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

```
.  tem.out.println(s.name()))
```

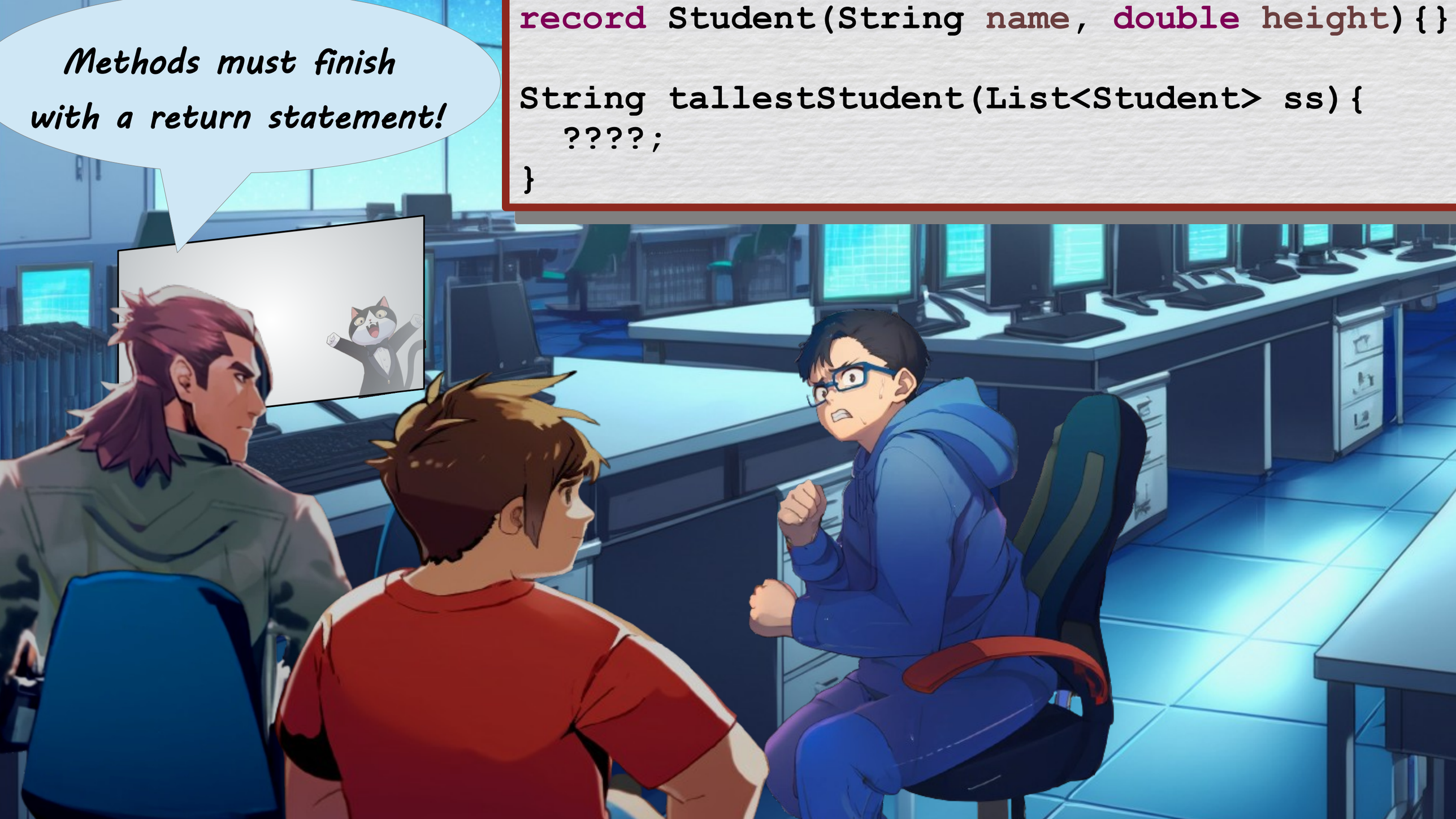
```
-> s1 height  ) ? s1 : s2)
```

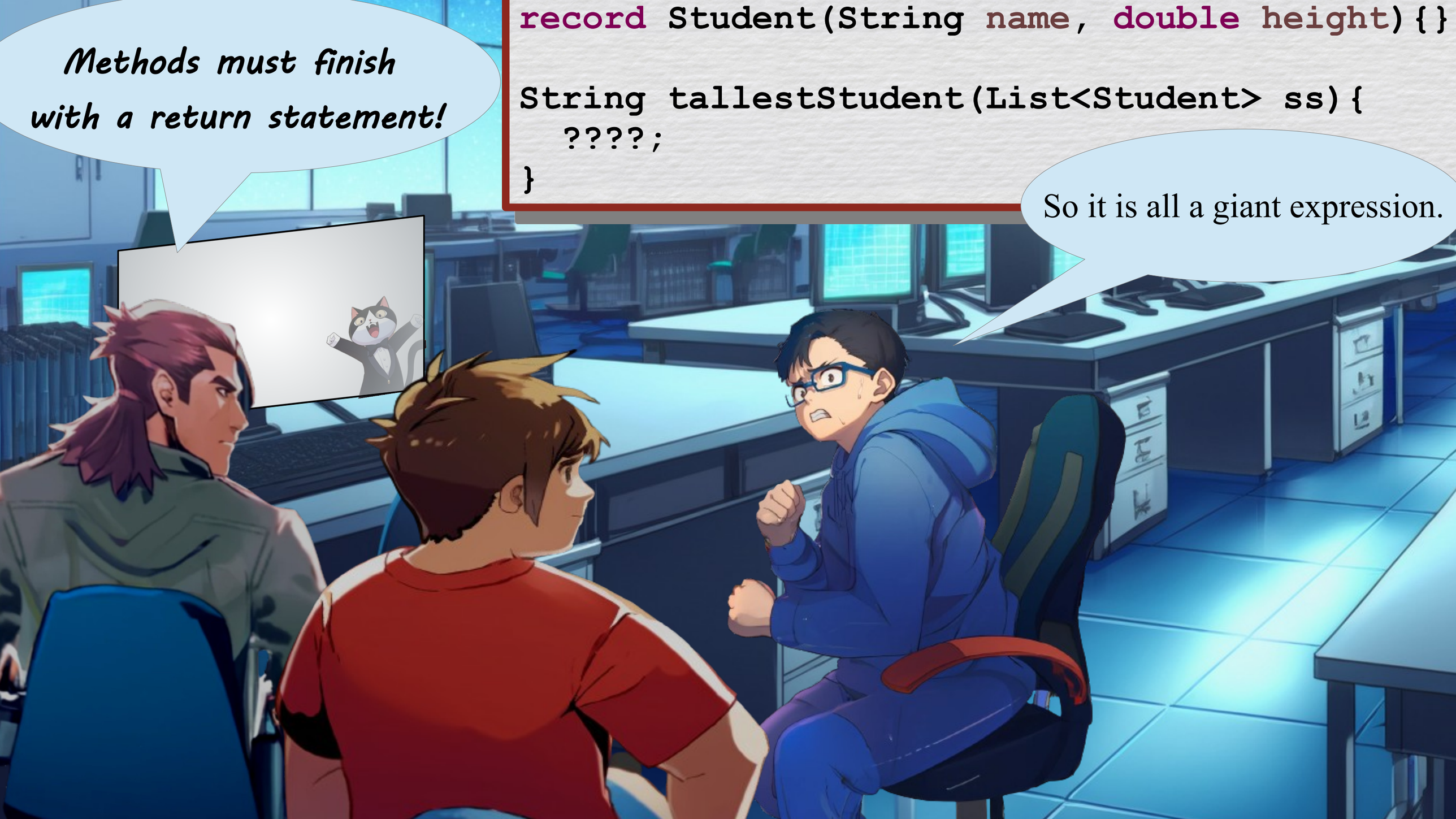
```
.height() > 6)
```

```
(()->new Error("E  ovided"))
```

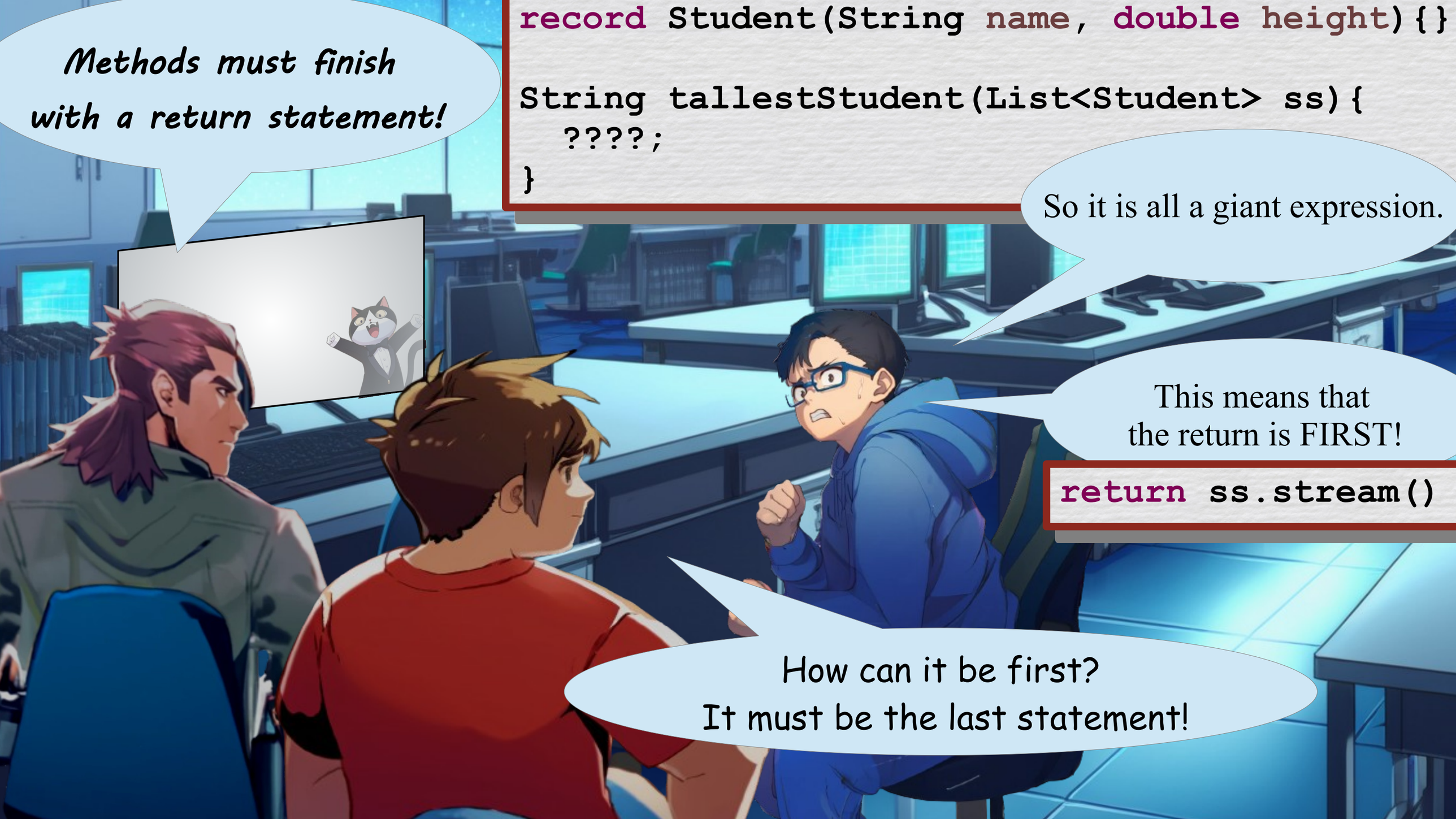
The only semicolon is already in the method body

Thus there is only one statement!


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So it is all a giant expression.

Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So it is all a giant expression.
This means that the return is FIRST!
return ss.stream()


Methods must finish with a return statement!
record Student(String name, double height){}
String tallestStudent(List<Student> ss){
????;
}
So it is all a giant expression.
This means that the return is FIRST!
return ss.stream()
How can it be first? It must be the last statement!

It is the ONLY statement!
It is both first and last!

It is the ONLY statement!
It is both first and last!
Both the first and the last Alpha and Omega. Return!

It is the ONLY statement!
It is both first and last!
Both the first and the last Alpha and Omega. Return!
So... it's all a giant expression!

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

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

What is next then?
We are searching for a tall student

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

```
.get()
```

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

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

```
.filter(s -> s.height() > 6)
```

```
.findFirst()
```

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

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

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

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

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

```
.get()
```

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

```
.forEach(s -> System
```

```
.findFirst()
```

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

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

What is next then?
We are searching for a tall student

Should we filter for all the students taller then 6?

```
.filter(s -> s.height() > 6)
```