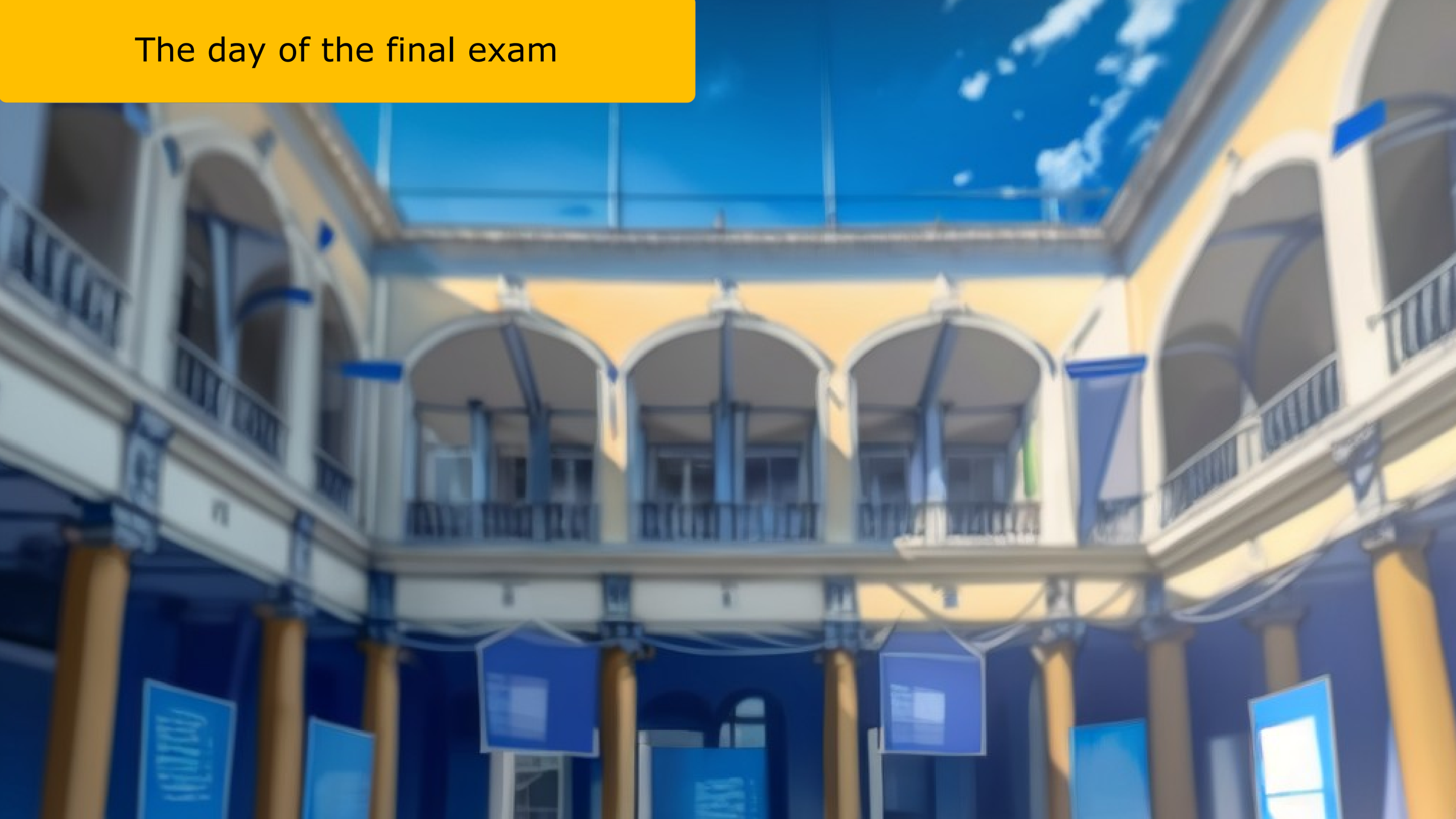
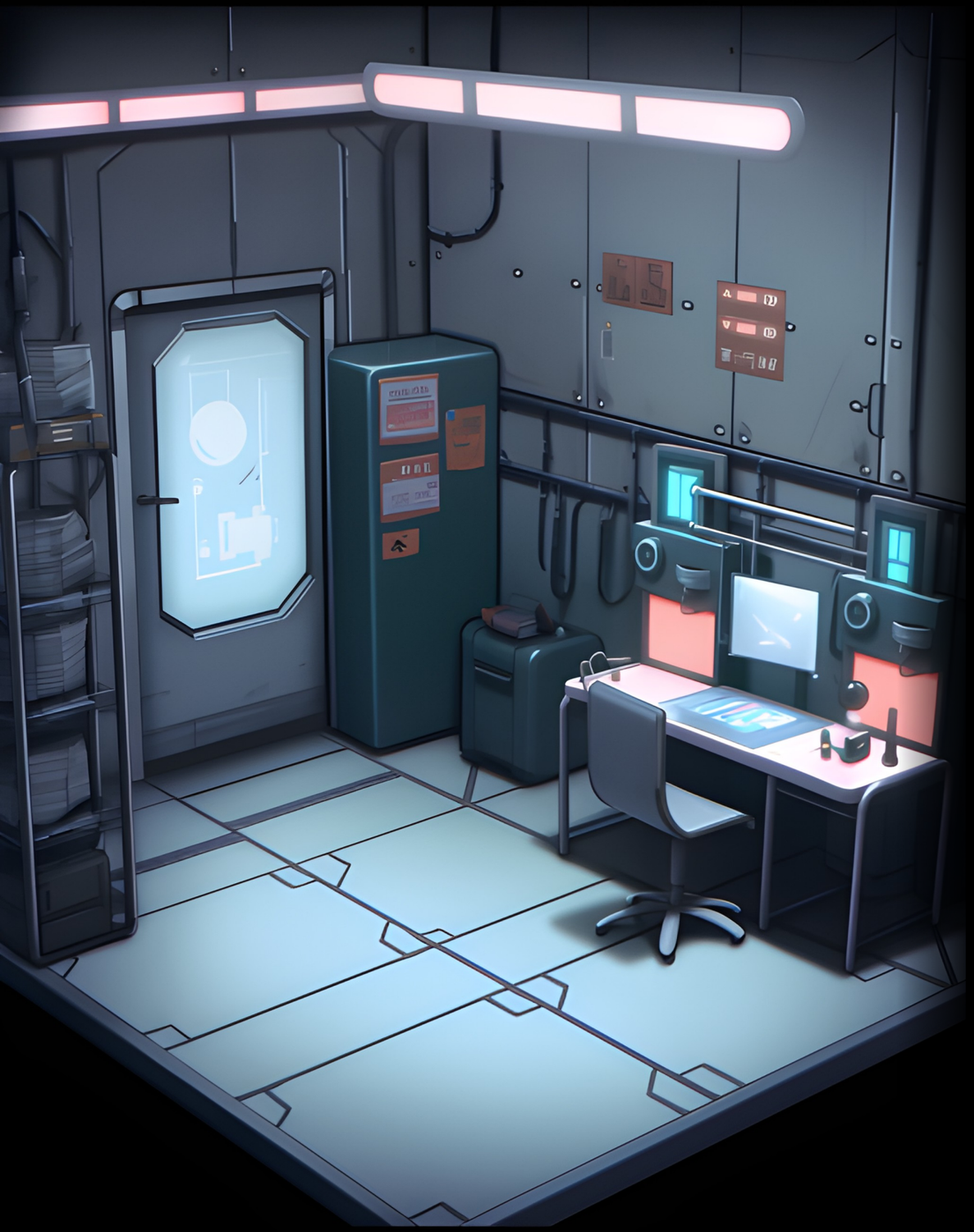
UPU, Freshman campus


The freshman year is getting to an end.


The freshman year is getting to an end.
Today is the day!

The day of the final exam





The final exam started at 9 am

The final exam started at 9 am

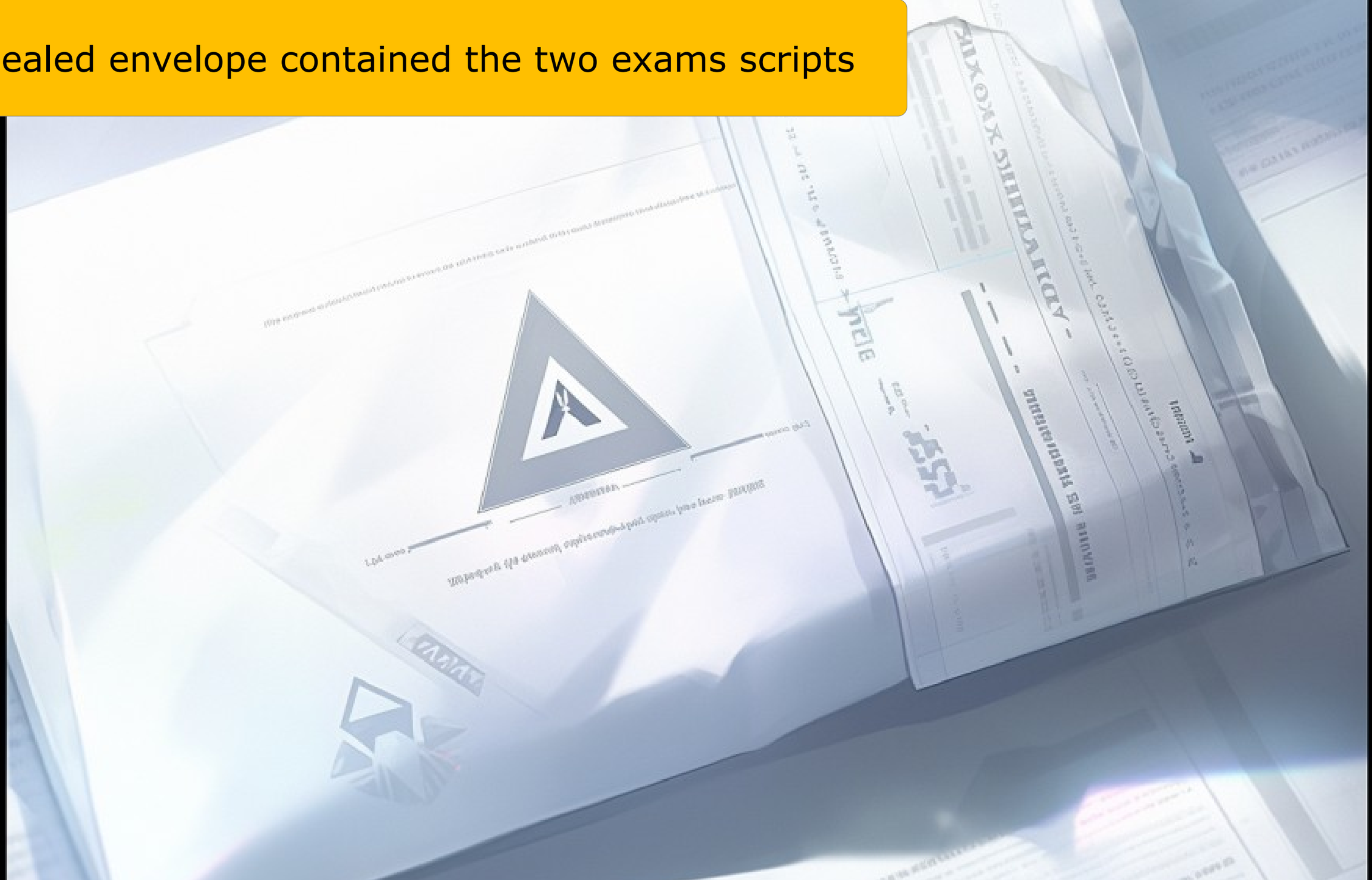
The exam is held in sealed
individual room and it lasts 5 hours

It is now 10:30 am

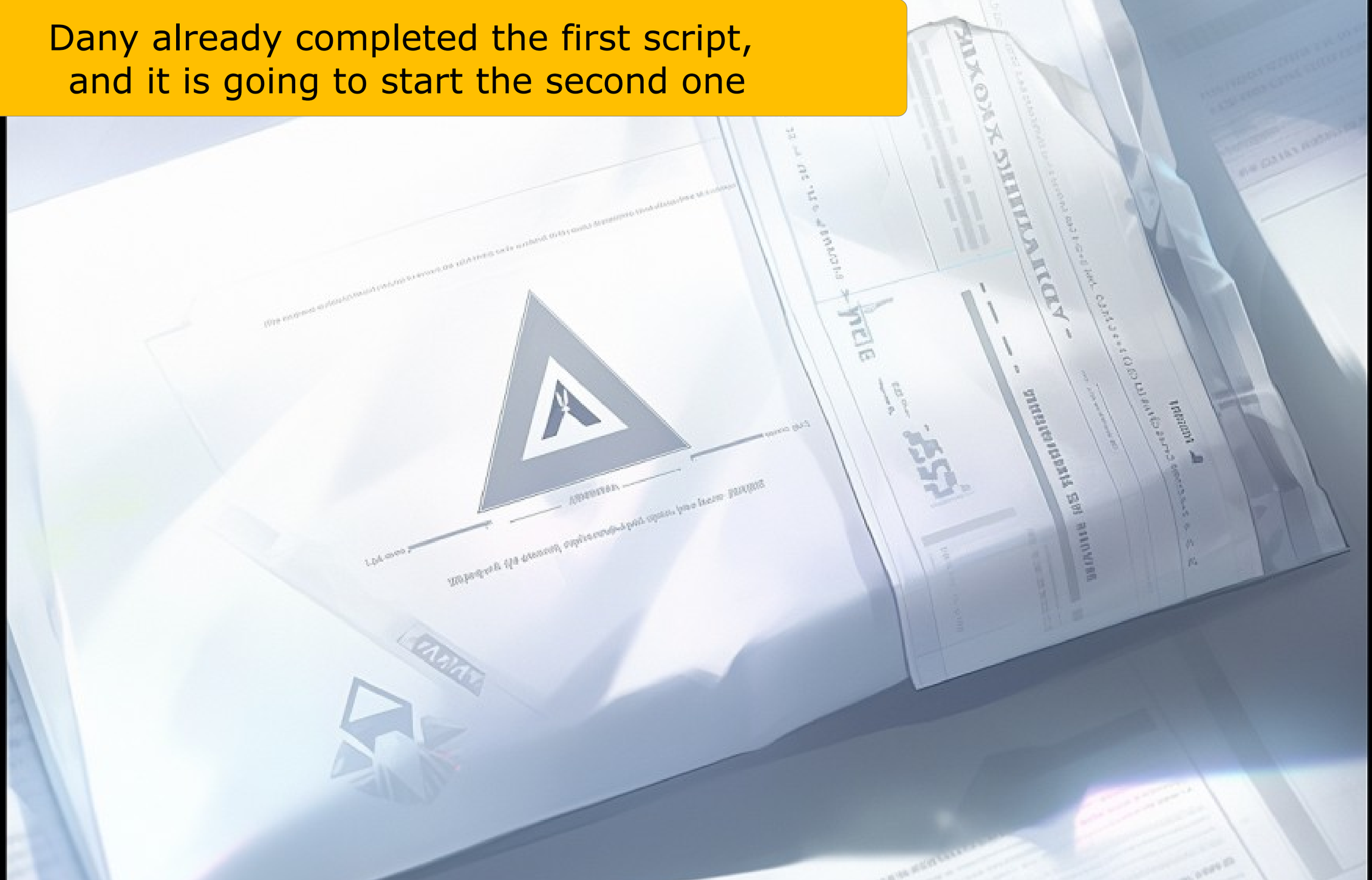
It is now 10:30 am

Three hours and a half of time left

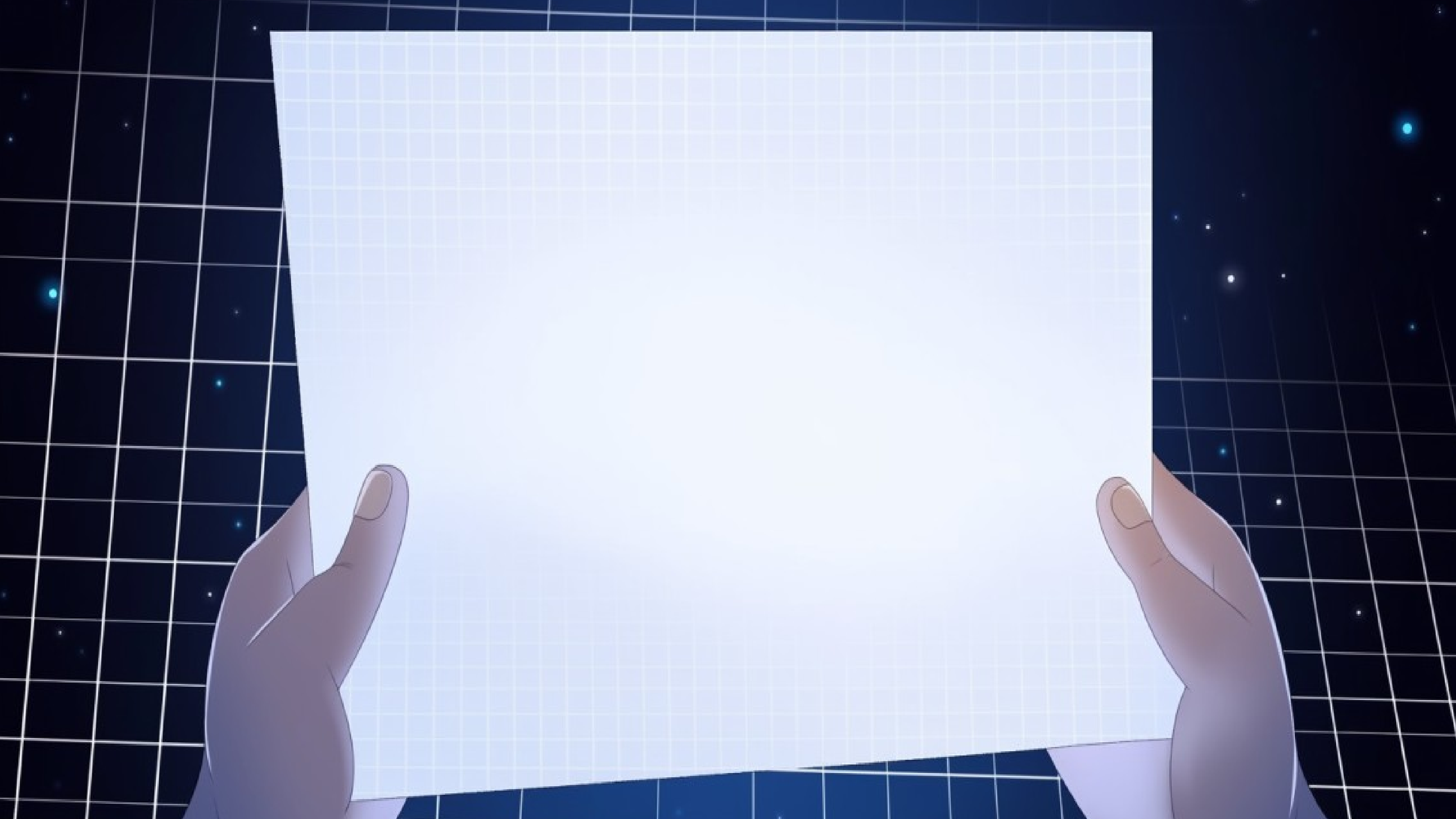
A sealed envelope contained the two exams scripts

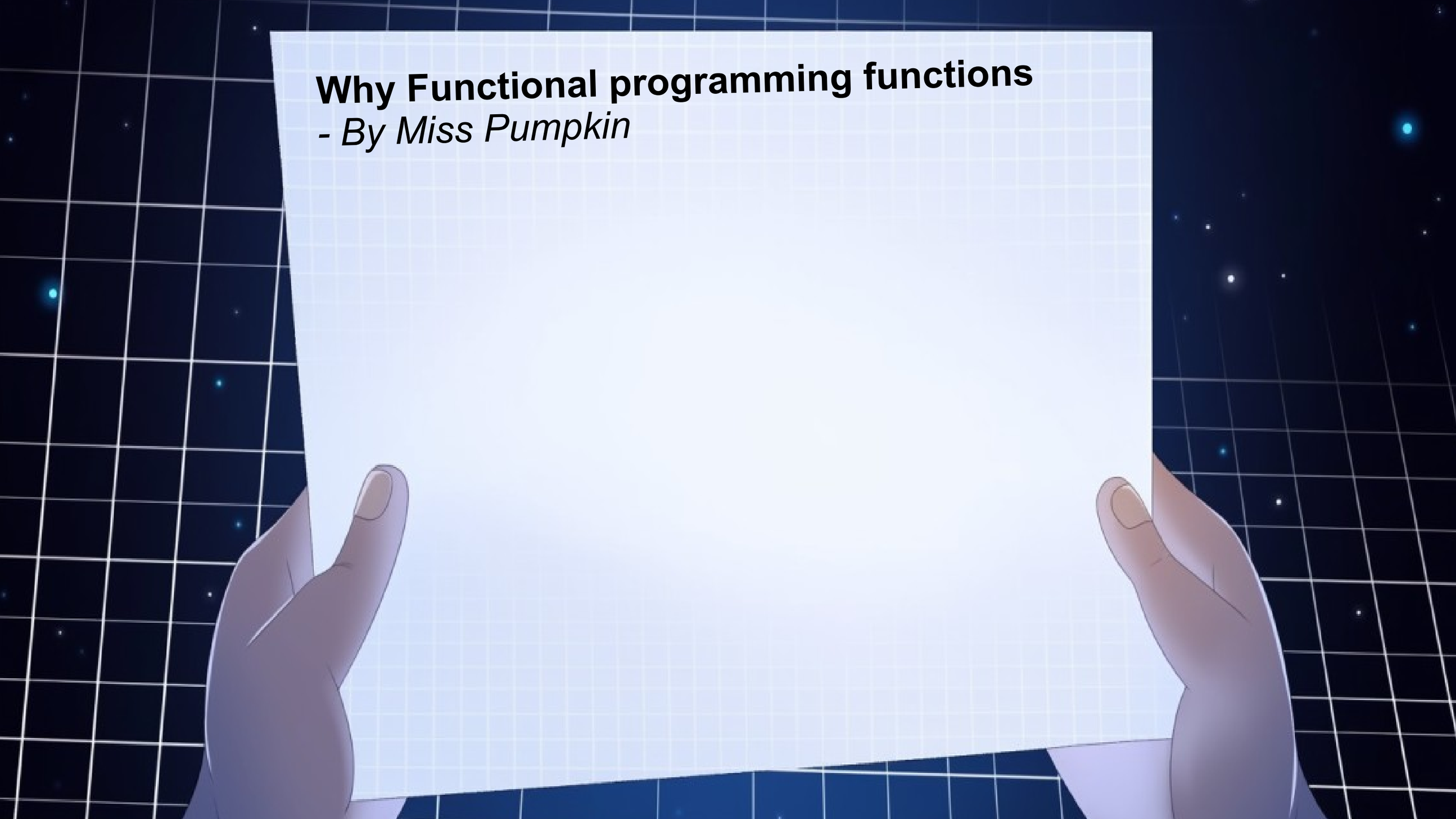


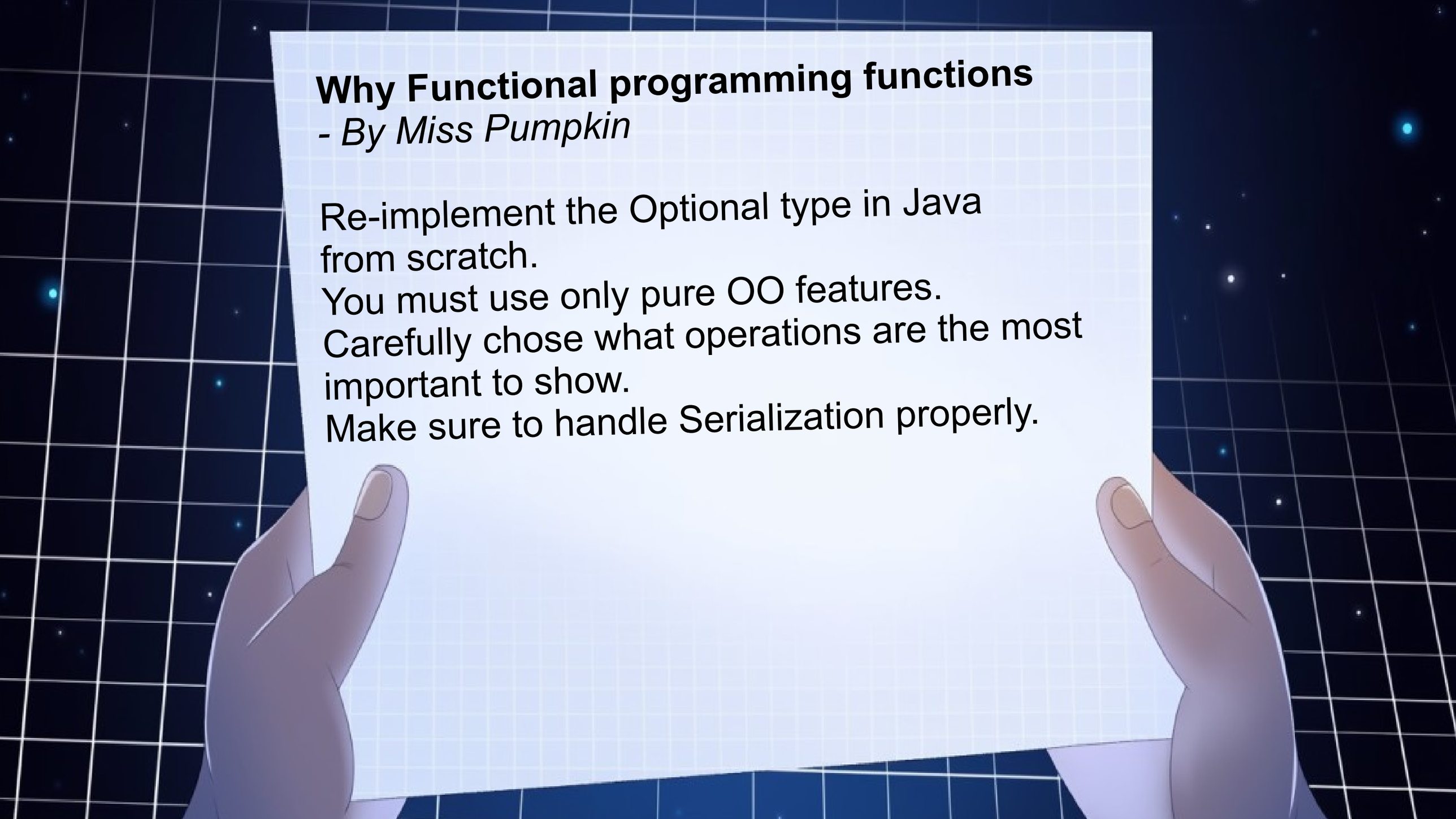
Dany already completed the first script,
and it is going to start the second one

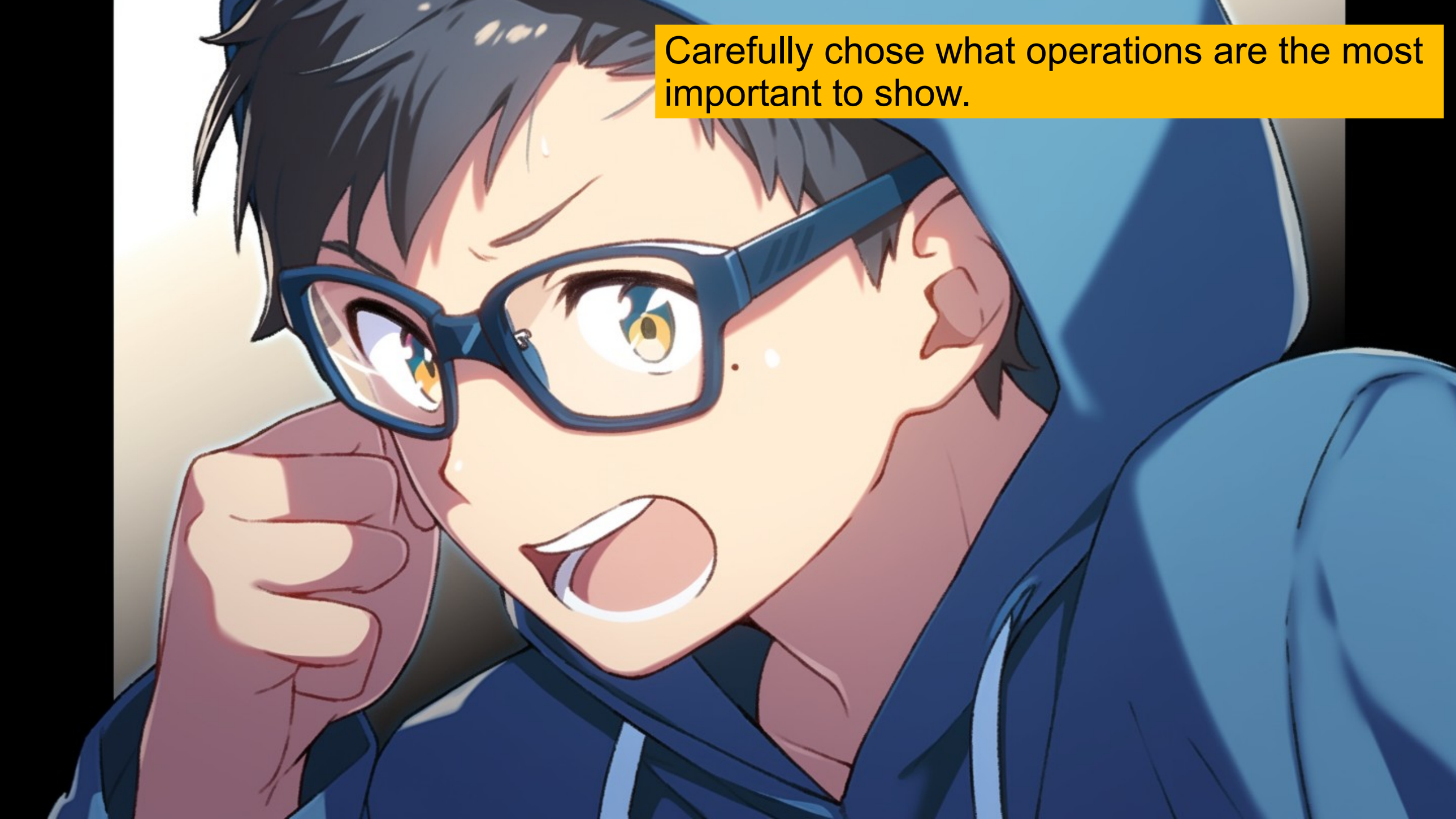


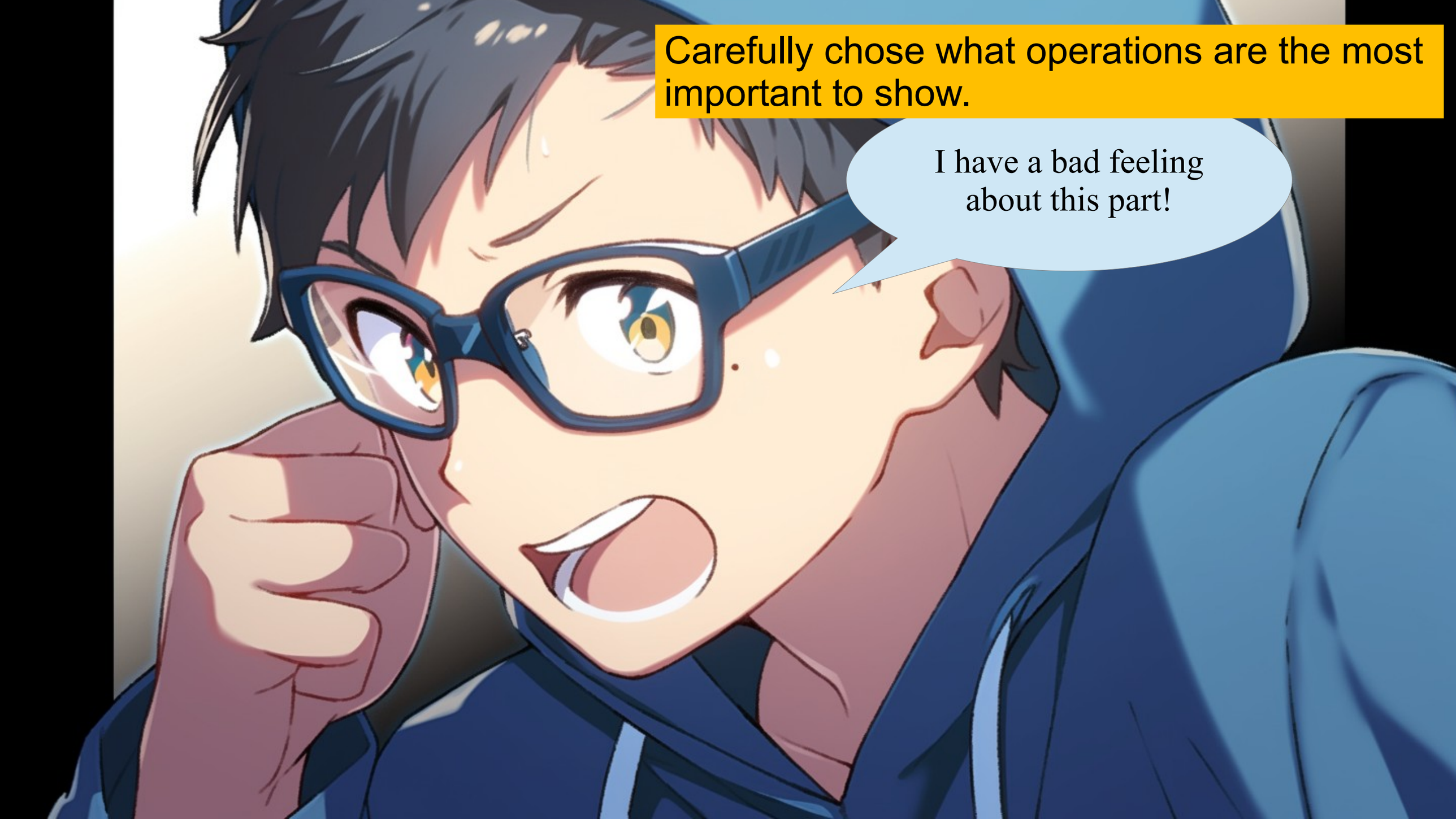
Let see the second script

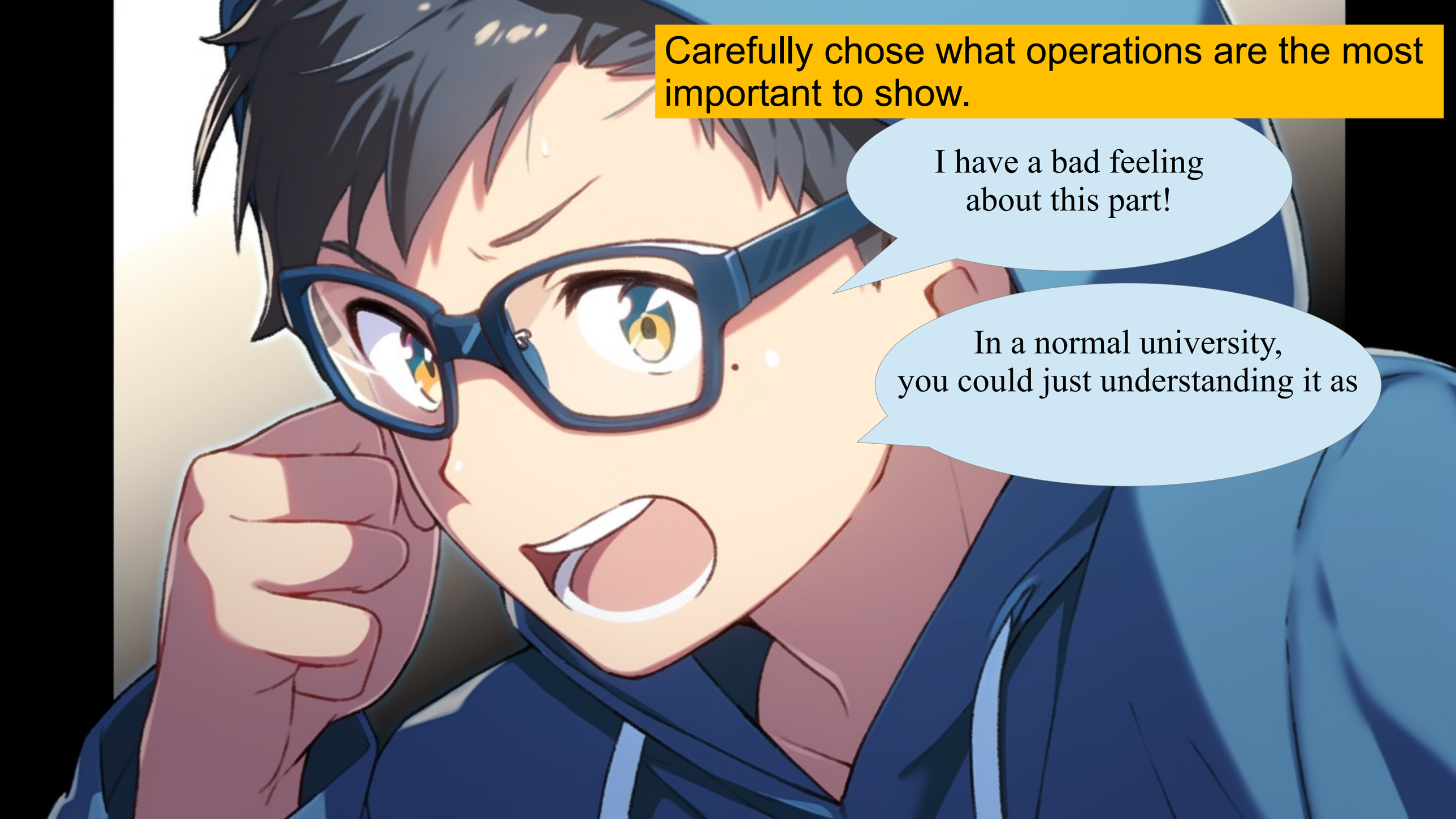


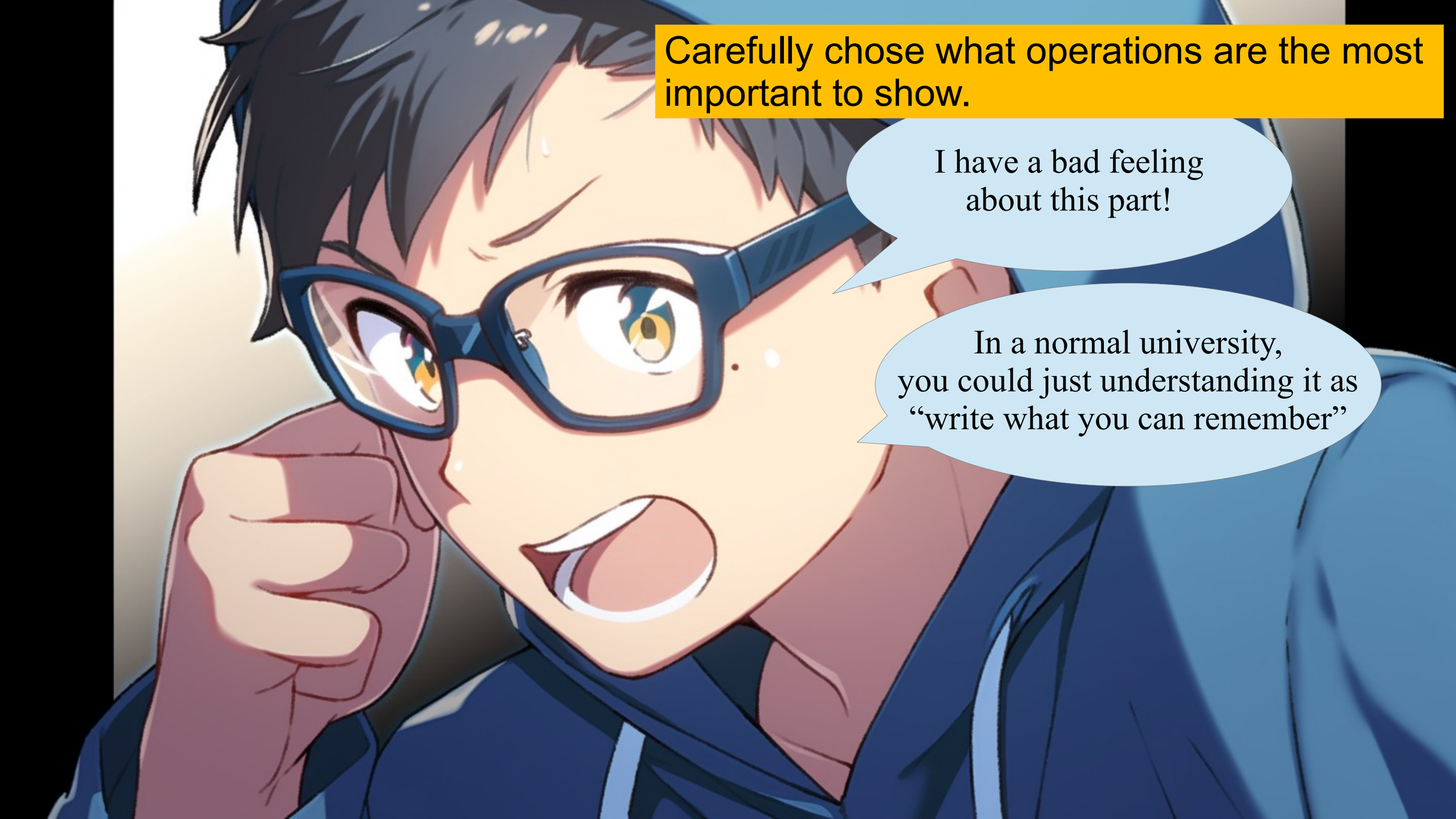

Why Functional programming functions
- By Miss Pumpkin

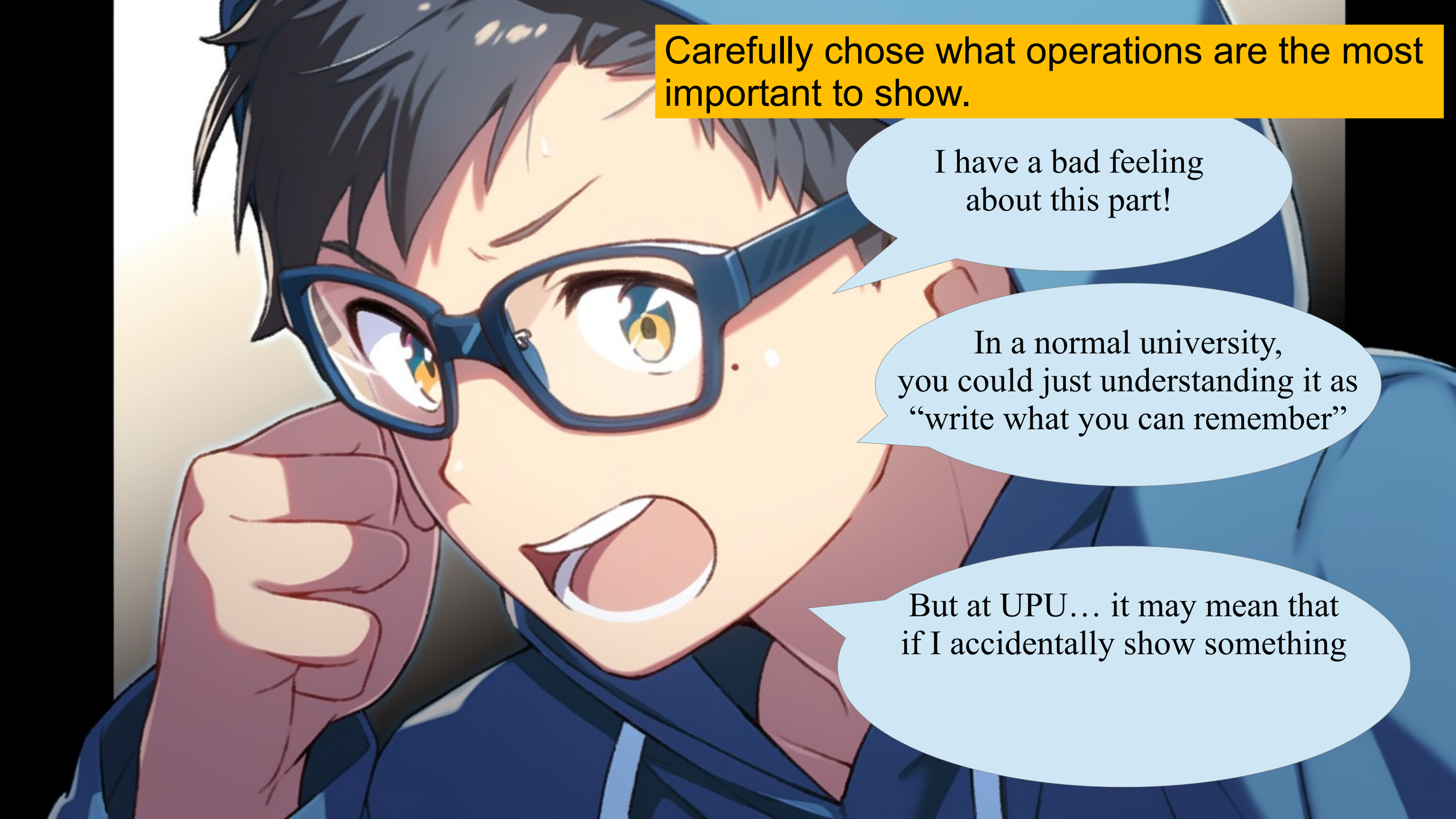

Why Functional programming functions
- By Miss Pumpkin
Re-implement the Optional type in Java from scratch.
You must use only pure OO features.
Carefully chose what operations are the most important to show.
Make sure to handle Serialization properly.

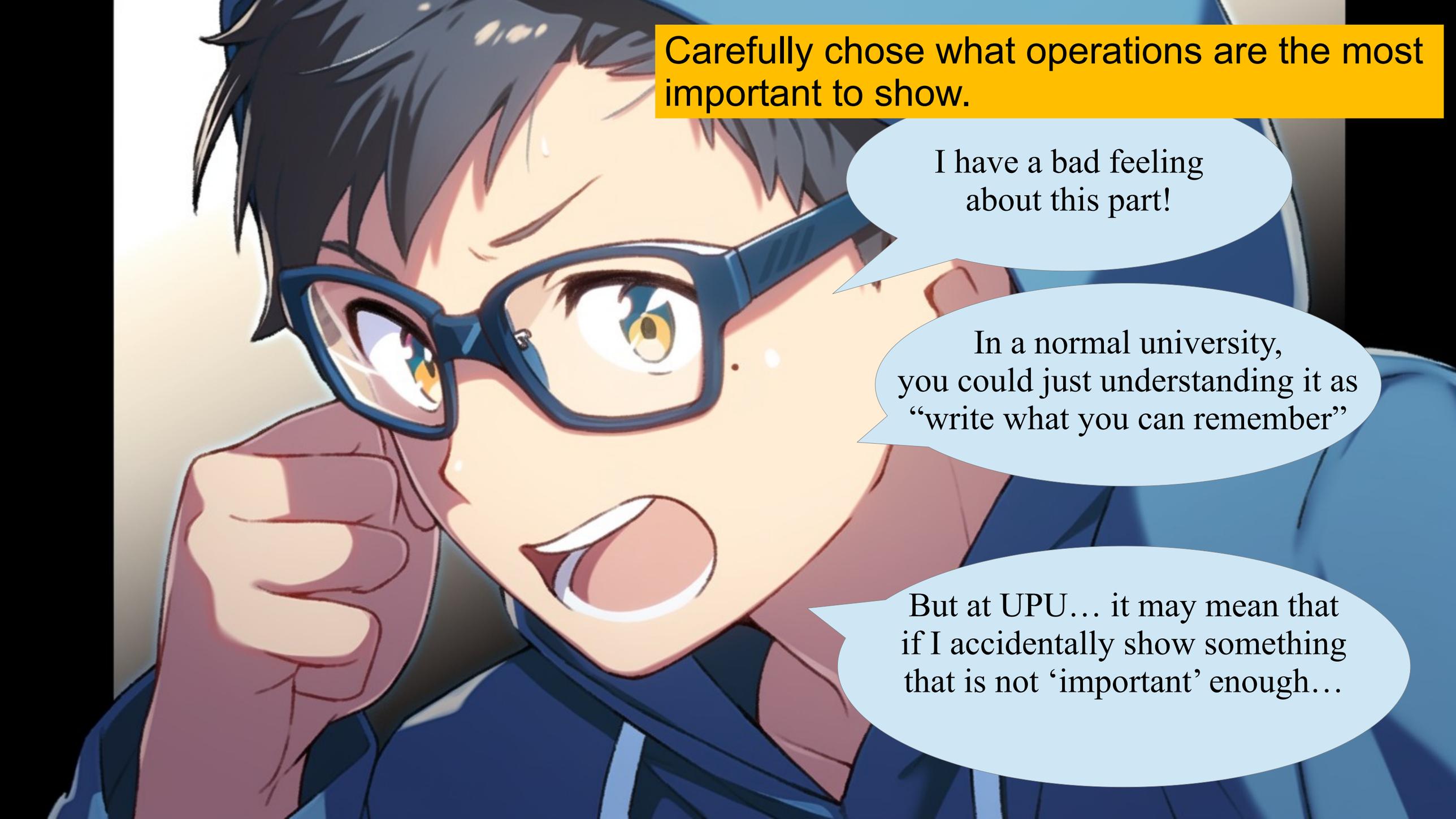

Carefully chose what operations are the most important to show.

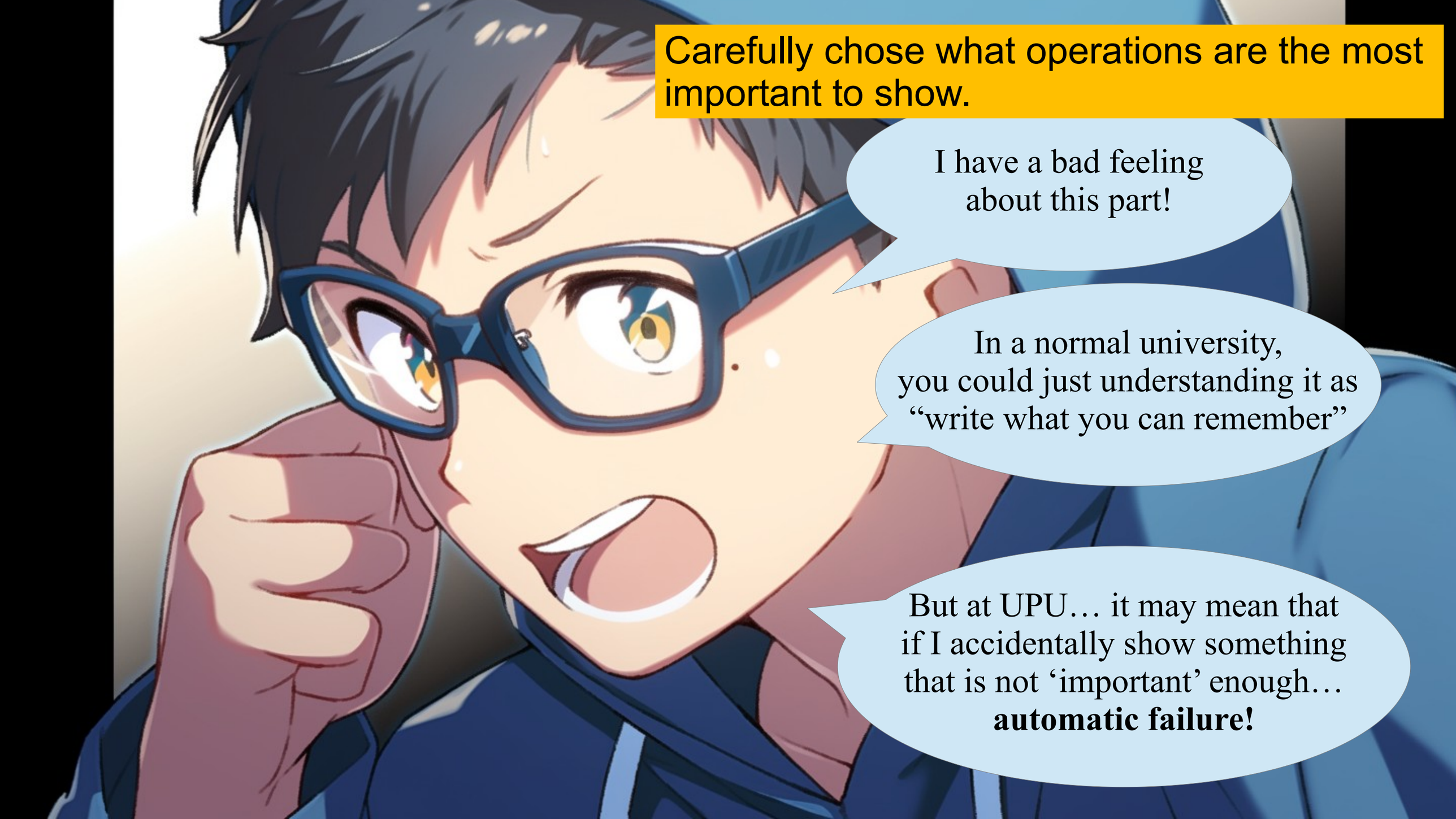

Carefully chose what operations are the most important to show.
I have a bad feeling about this part!


Carefully chose what operations are the most important to show.
I have a bad feeling about this part!
In a normal university, you could just understanding it as


Carefully chose what operations are the most important to show.
I have a bad feeling about this part!
In a normal university, you could just understanding it as “write what you can remember”


Carefully chose what operations are the most important to show.
I have a bad feeling about this part!
In a normal university, you could just understanding it as “write what you can remember”
But at UPU… it may mean that if I accidentally show something


Carefully chose what operations are the most important to show.
I have a bad feeling about this part!
In a normal university, you could just understanding it as “write what you can remember”
But at UPU… it may mean that if I accidentally show something that is not ‘important’ enough…


Carefully chose what operations are the most important to show.
I have a bad feeling about this part!
In a normal university, you could just understanding it as “write what you can remember”
But at UPU… it may mean that if I accidentally show something that is not ‘important’ enough… **automatic failure!**

Optional is one of the best
case studies for dynamic dispatch

Optional is one of the best case studies for dynamic dispatch

An old style programmer may be tempted to make an Optional<T> class with a potentially ‘null’ T field

Optional is one of the best
case studies for dynamic dispatch

An old style programmer may be
tempted to make an Optional<T> class
with a potentially ‘null’ T field

That would be ridiculous:
it would force all methods inside of
Optional to start with an ‘if’

The most important
thing about Optional,
is the general code structure
and the division in types

Optional is our top level concept

The most important
thing about Optional,
is the general code structure
and the division in types

Optional is our top level concept

Optional<T>

The most important
thing about Optional,
is the general code structure
and the division in types

Optional is our top level concept

Optional<T>

An Optional can be either empty or contain a value

The most important
thing about Optional,
is the general code structure
and the division in types

Optional is our top level concept

Optional<T>

An Optional can be either empty or contain a value

Empty<T>

Some<T>

The most important
thing about Optional,
is the general code structure
and the division in types

In code, we can represent it like this!

In code, we can represent it like this!

A sealed interface allowing only Empty and Some

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{


}
```

In code, we can represent it like this!

A sealed interface allowing only Empty and Some

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{


}
```

In code, we can represent it like this!

A sealed interface allowing only Empty and Some

And then implementations for Empty and Some

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{

}

final class Empty<T> implements Optional<T>{

}

record Some<T>(T get) implements Optional<T>{

}
```

In code, we can represent it like this!

A sealed interface allowing only Empty and Some

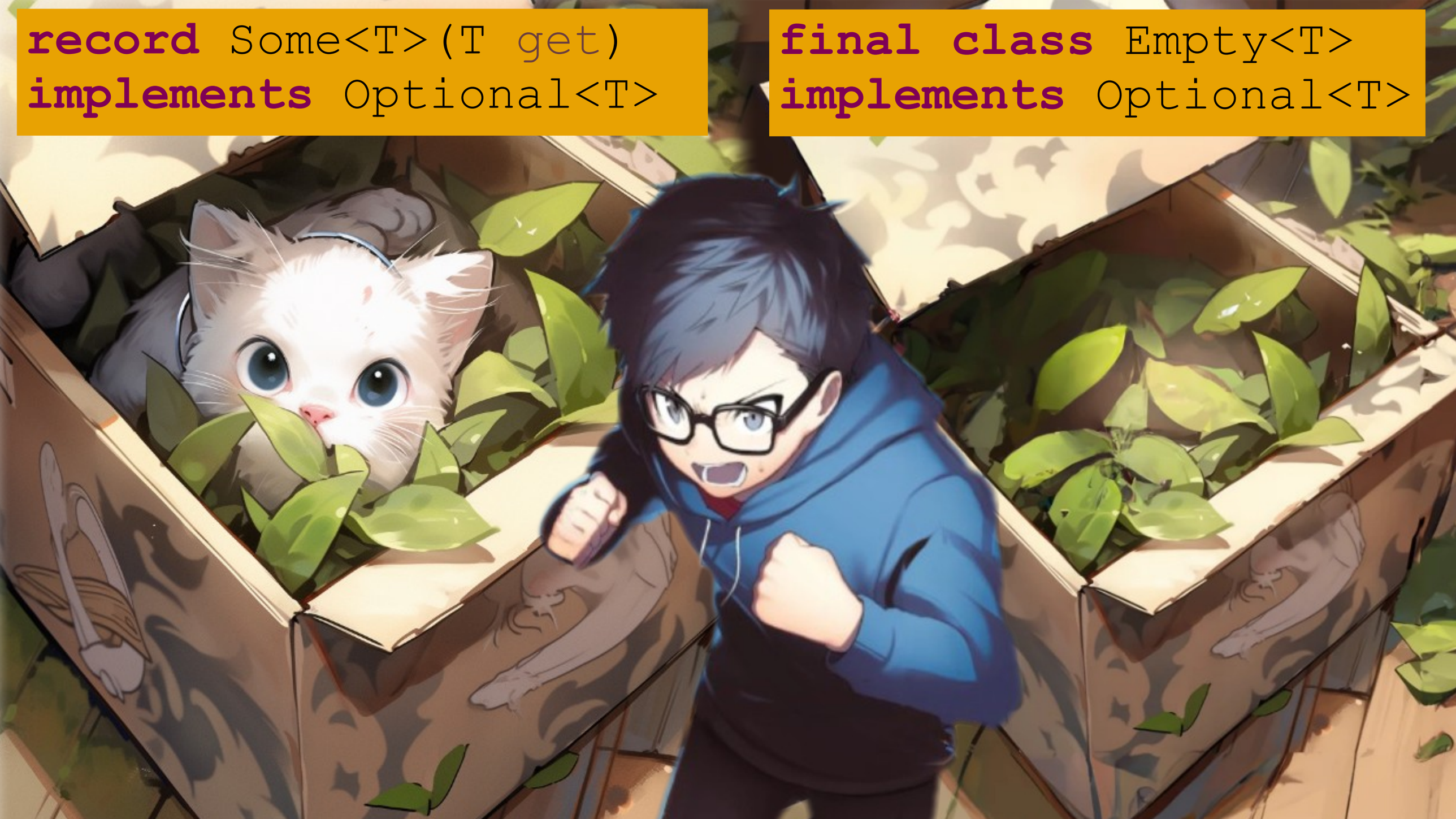
And then implementations for Empty and Some

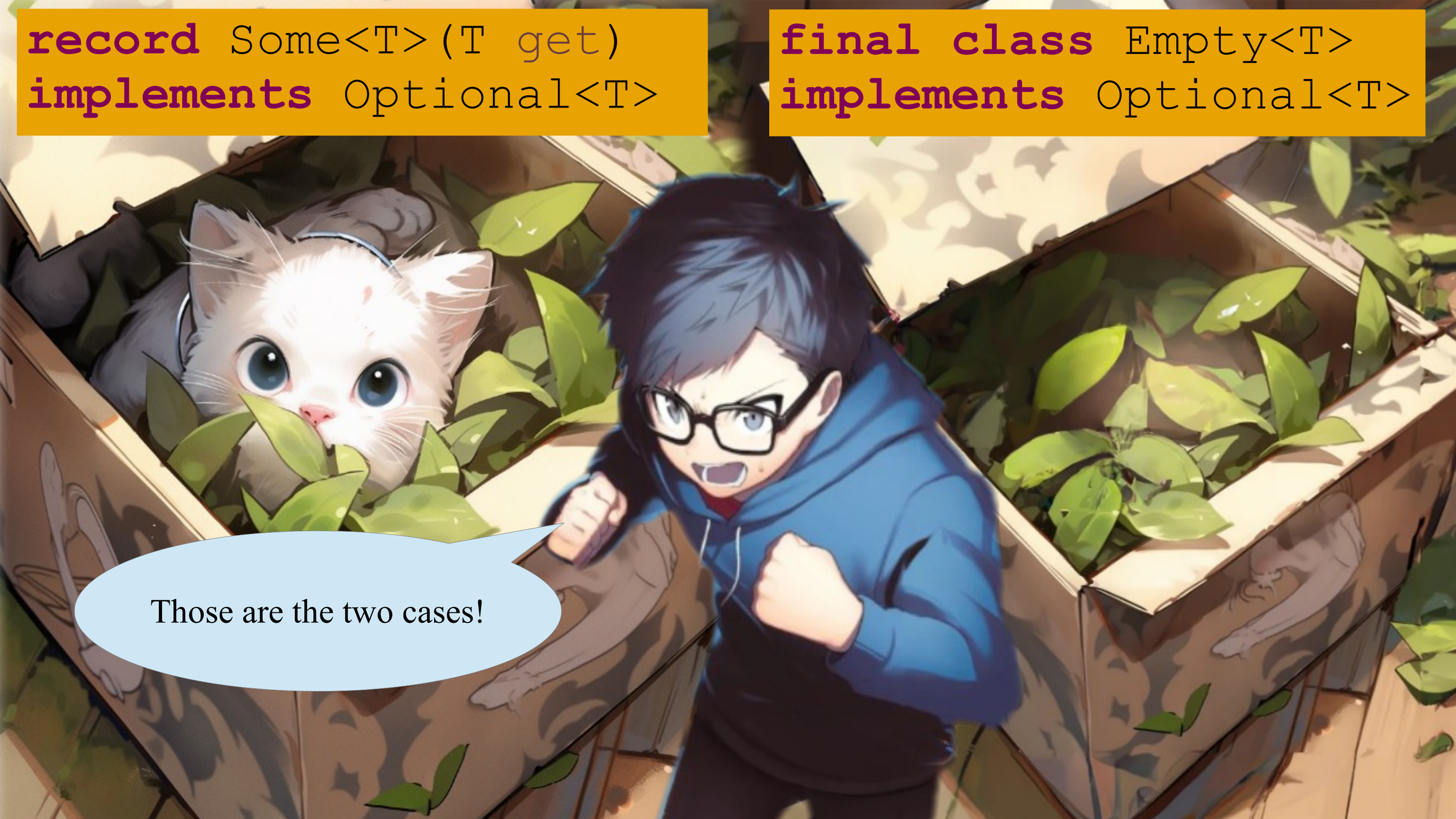

Optional will be public

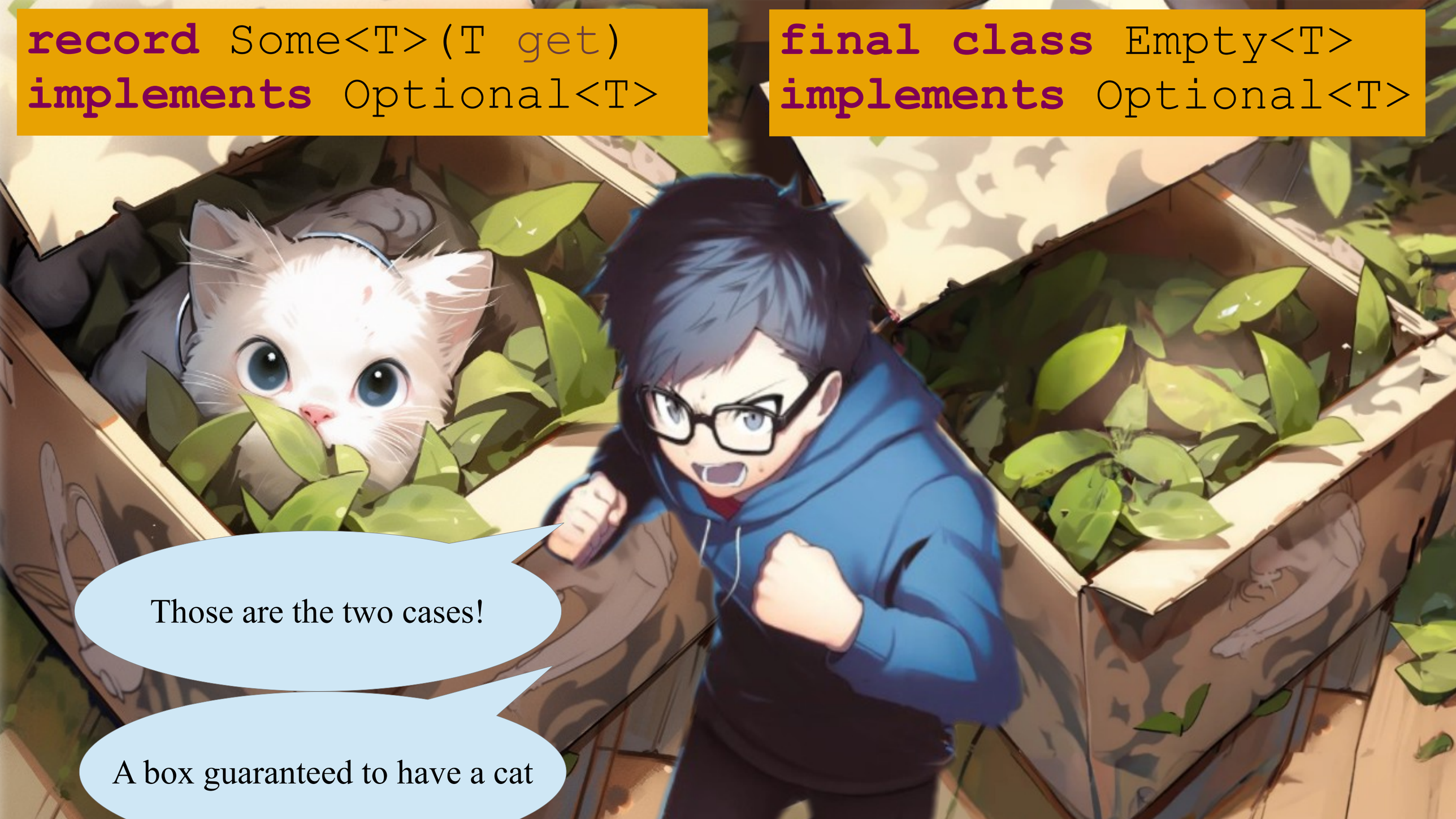

Optional will be public
Empty and Some
should be package private

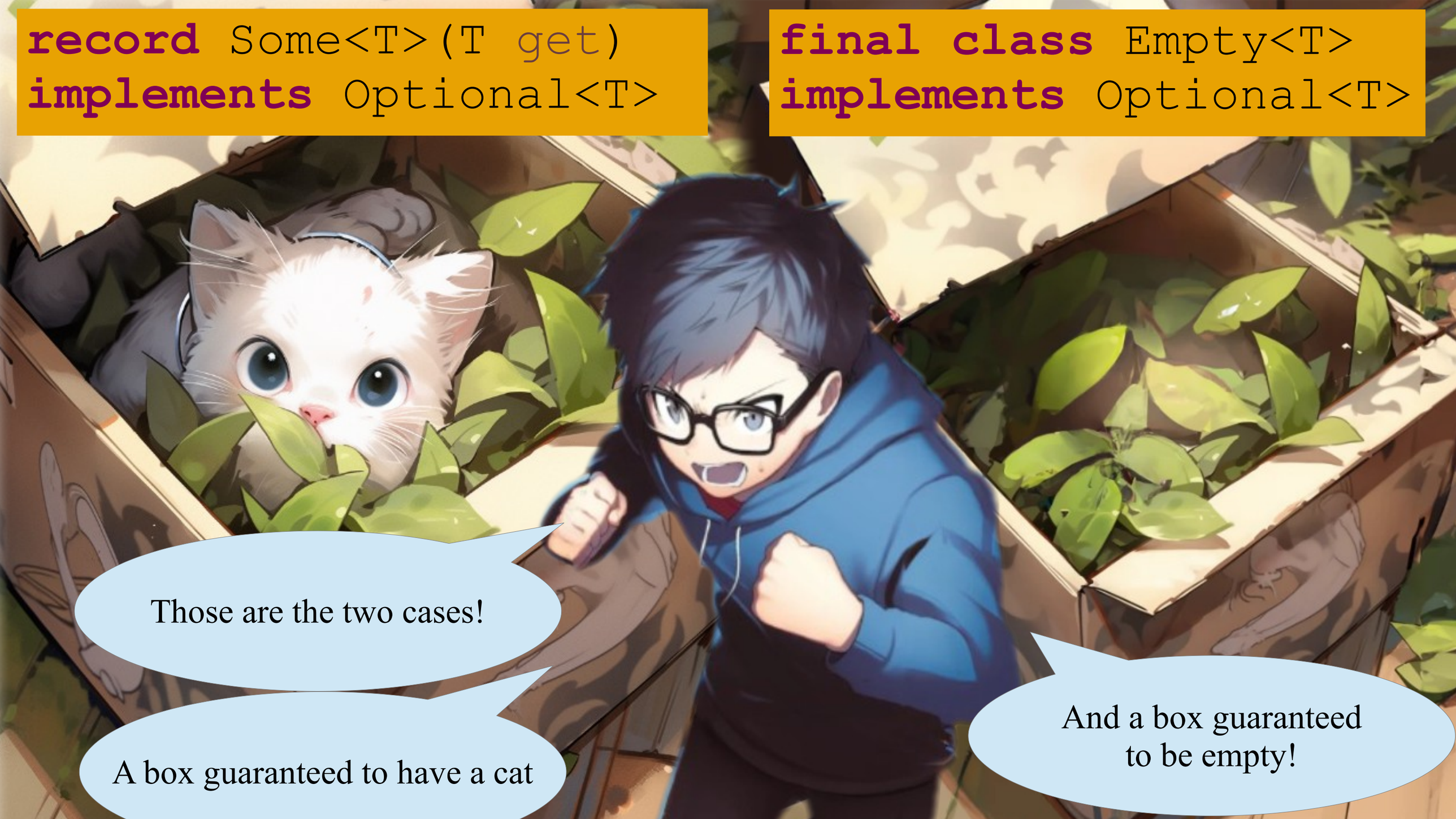

Optional will be public
Empty and Some should be package private
The user of Optional should not need to know about those


record Some<T>(T get)
implements Optional<T>
final class Empty<T>
implements Optional<T>


record Some<T>(T get)
implements Optional<T>
final class Empty<T>
implements Optional<T>


record Some<T>(T get)
implements Optional<T>
final class Empty<T>
implements Optional<T>
Those are the two cases!

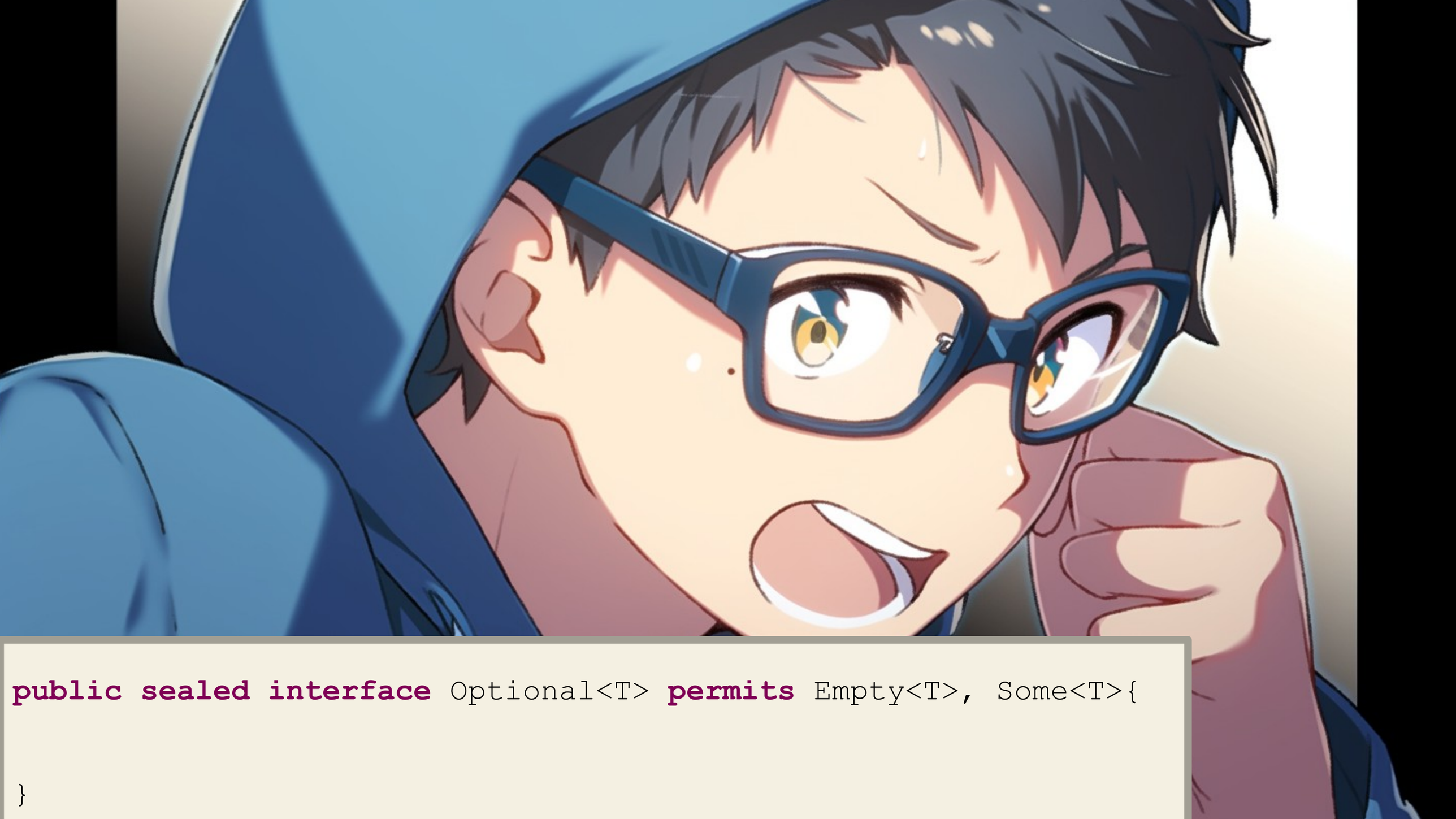

record Some<T>(T get)
implements Optional<T>
final class Empty<T>
implements Optional<T>
Those are the two cases!
A box guaranteed to have a cat


record Some<T>(T get)
implements Optional<T>
final class Empty<T>
implements Optional<T>
Those are the two cases!
A box guaranteed to have a cat
And a box guaranteed
to be empty!


The first step:
how to create optionals!


The first step:
how to create optionals!
How to create
empty optionals


The first step:
how to create optionals!
How to create
empty optionals
How to create
optionals with content



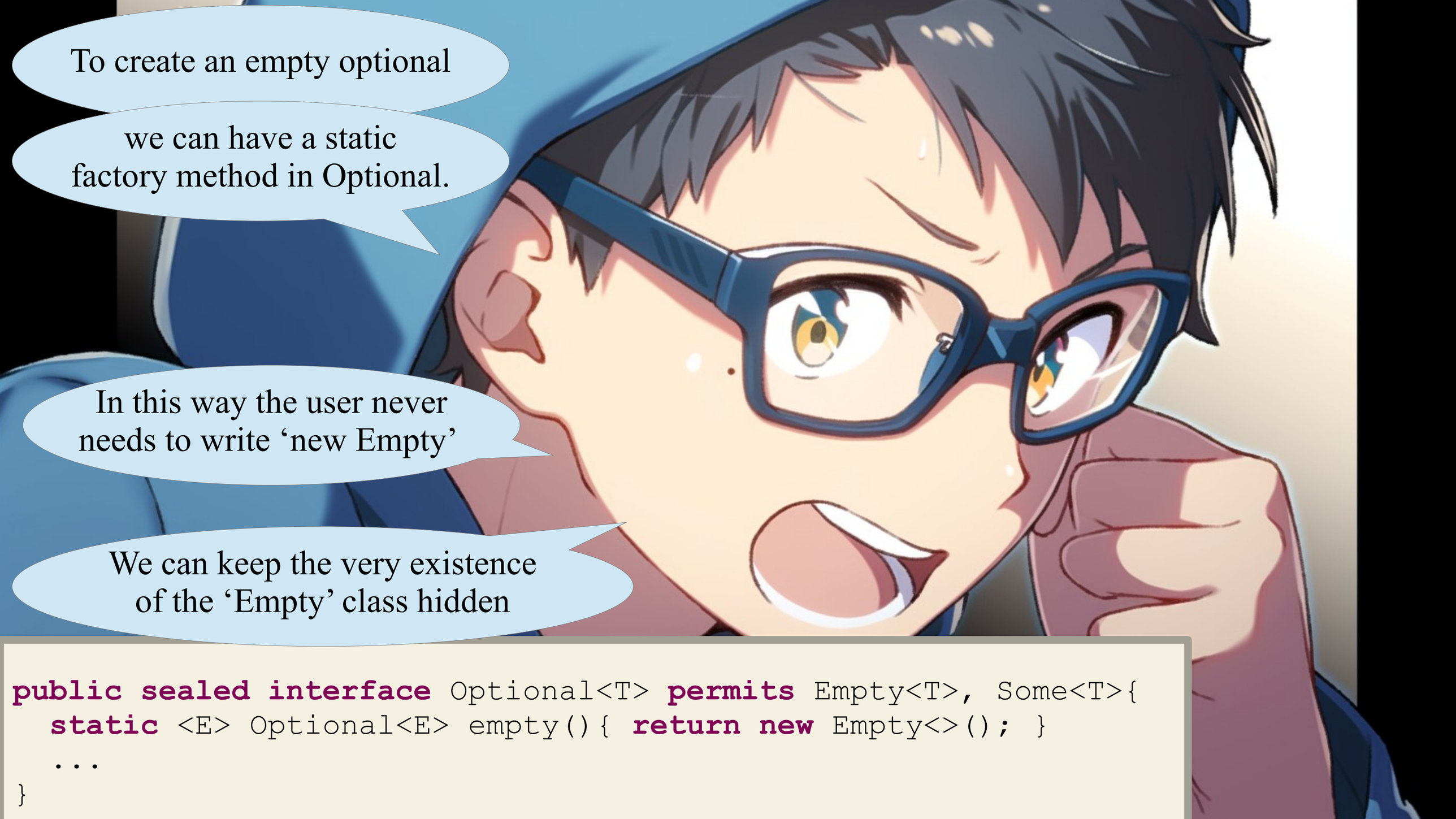
```
public sealed interface Optional<T> permits Empty<T>, Some<T>{


}
```


```
public sealed interface Optional<T> permits Empty<T>, Some<T>{


}
```

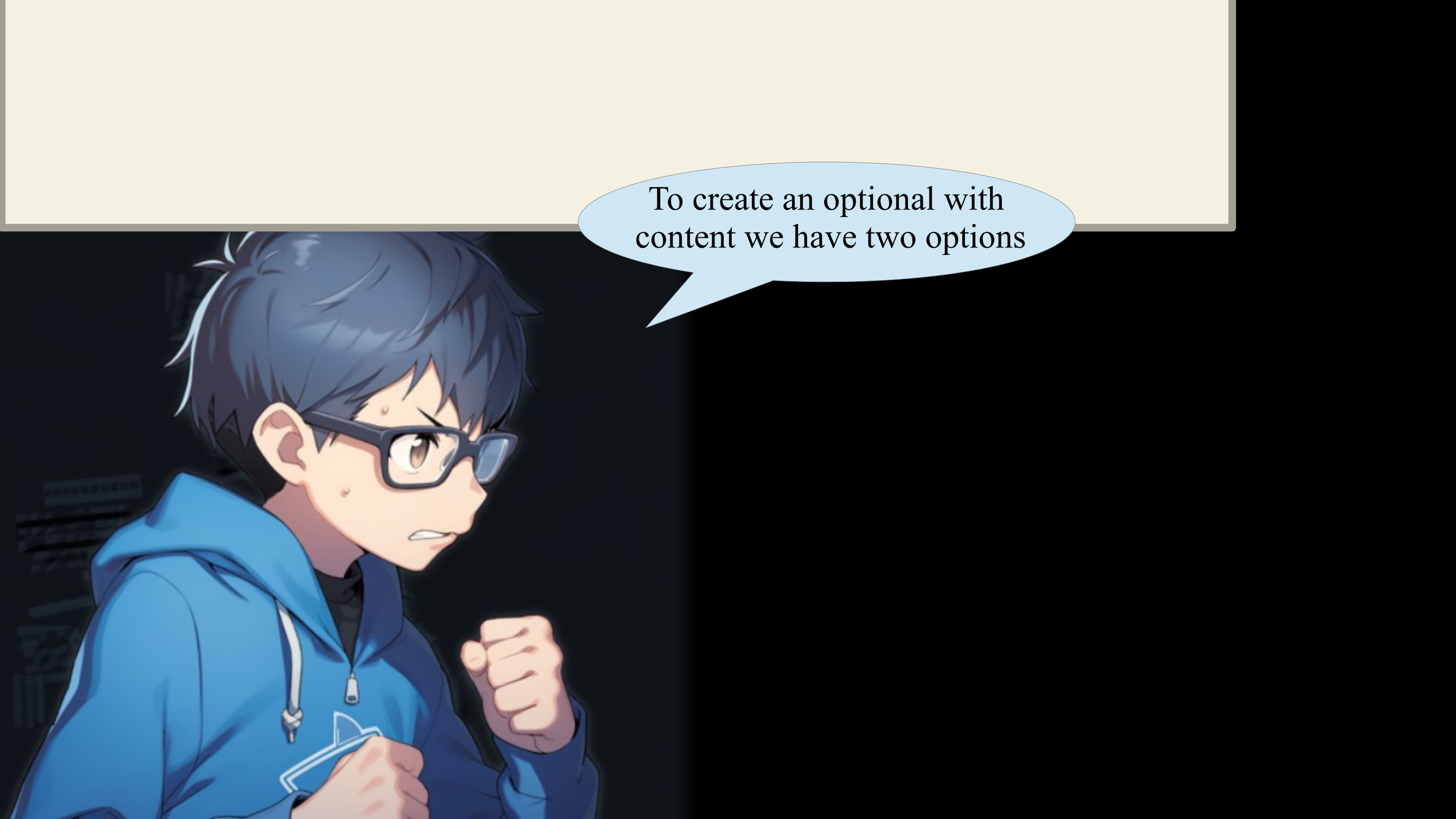

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   static <E> Optional<E> empty(){ return new Empty<>(); }
   ...
}
```


```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   static <E> Optional<E> empty(){ return new Empty<>(); }
   ...
}
```

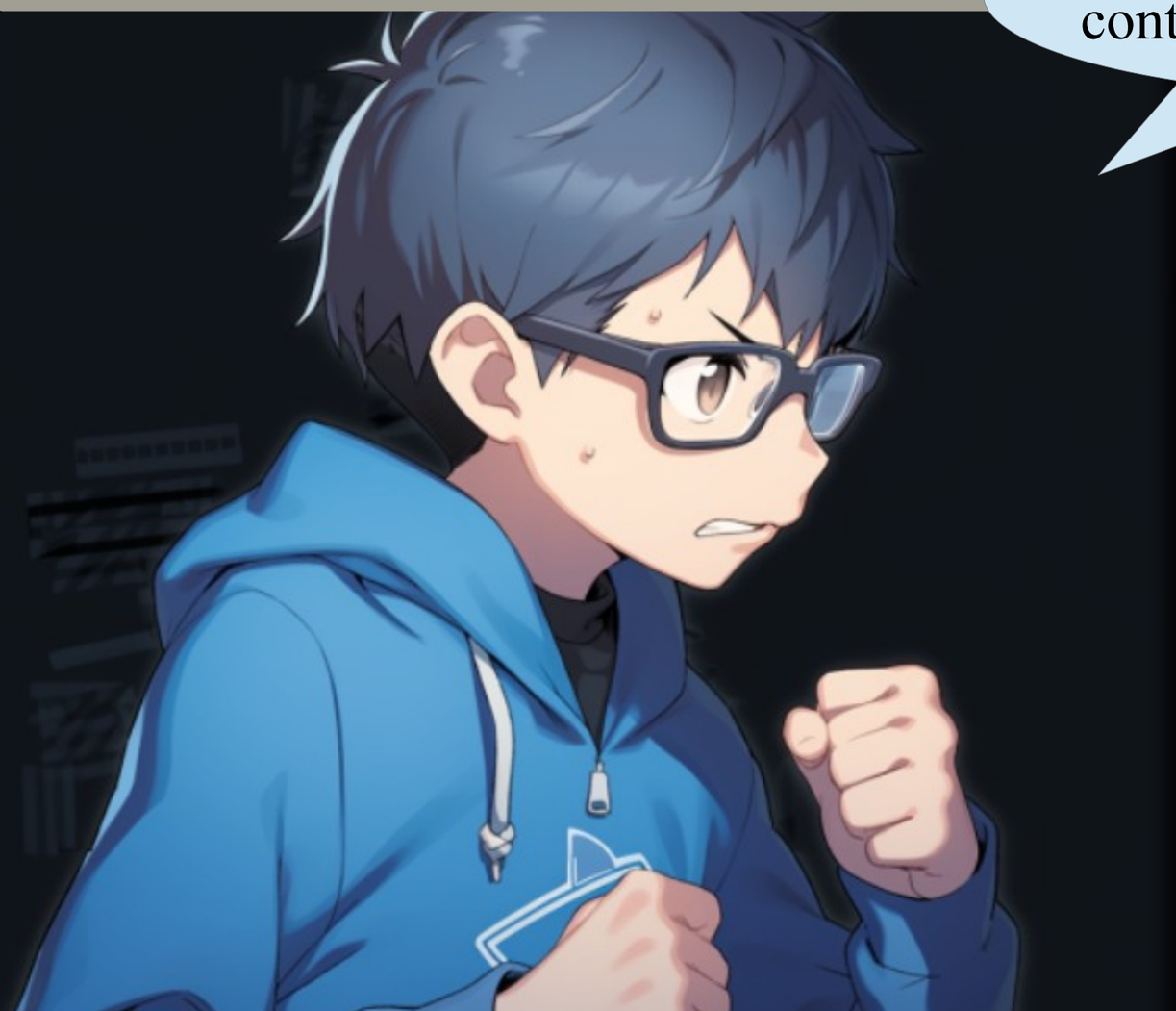

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   static <E> Optional<E> empty(){ return new Empty<>(); }
   ...
}
```



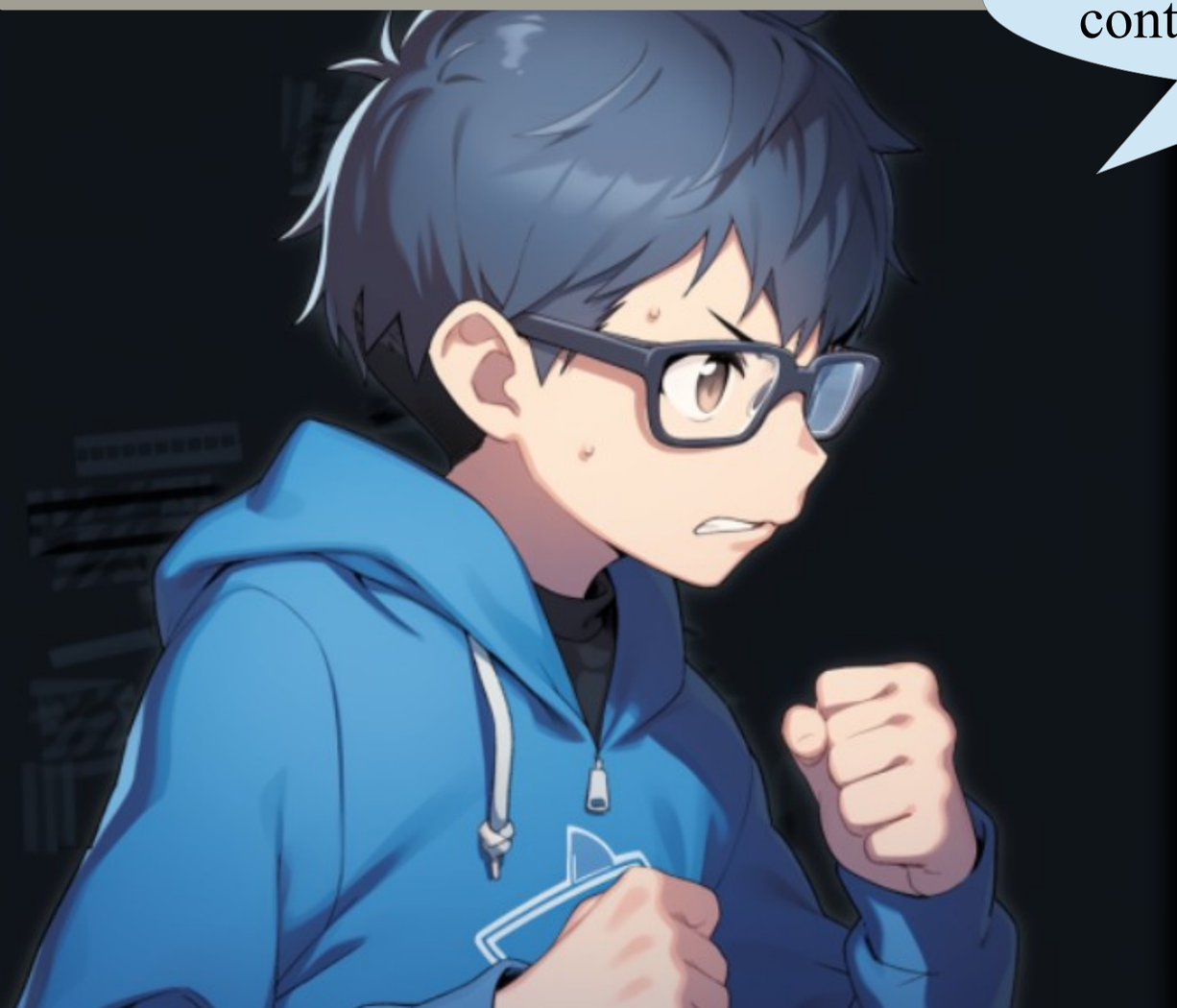

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  ...
}
```


To create an optional with content we have two options

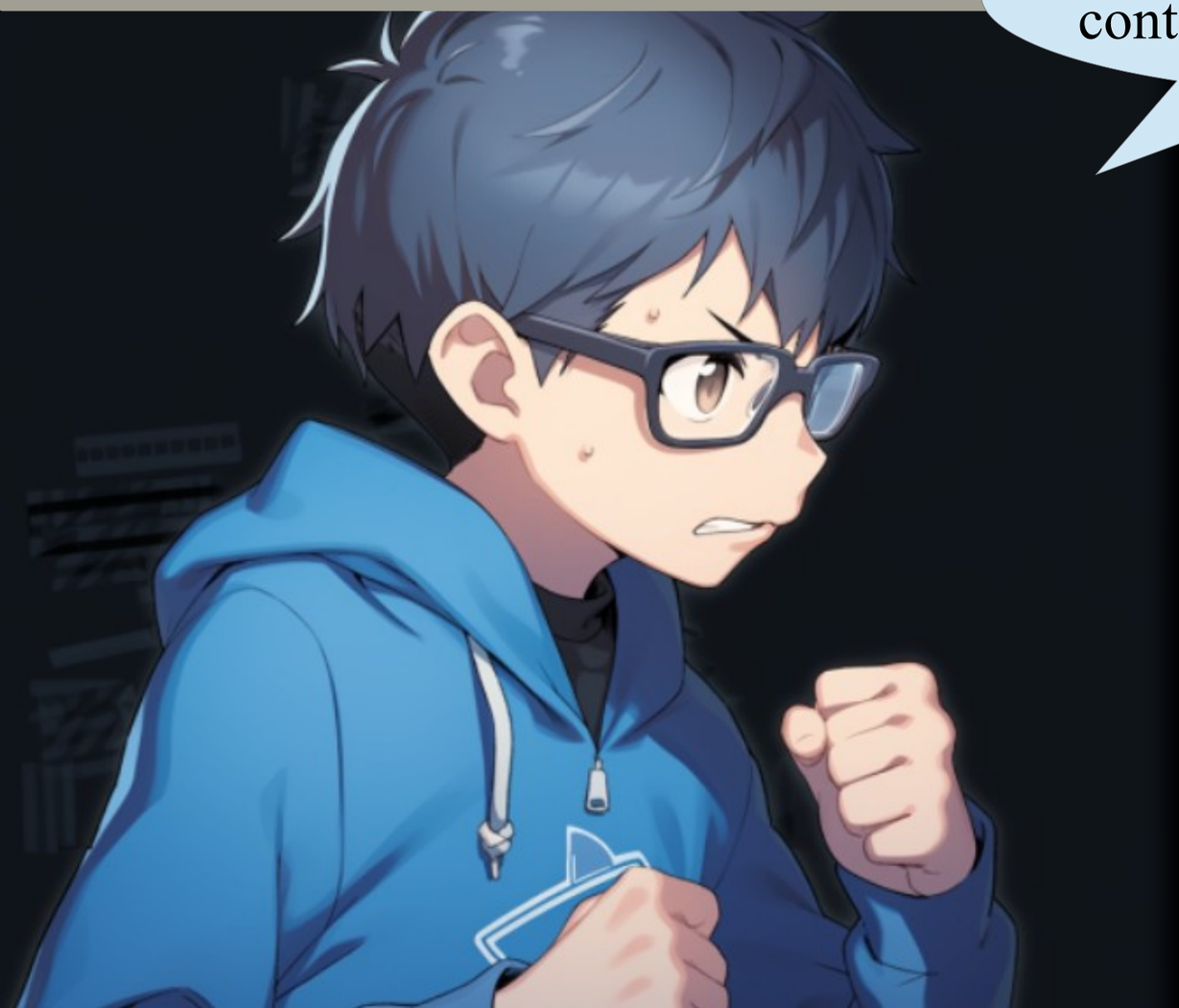
```
public sealed interface Optional<T> permits Empty<T>, Some<T>{



}
```



```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }



}
```

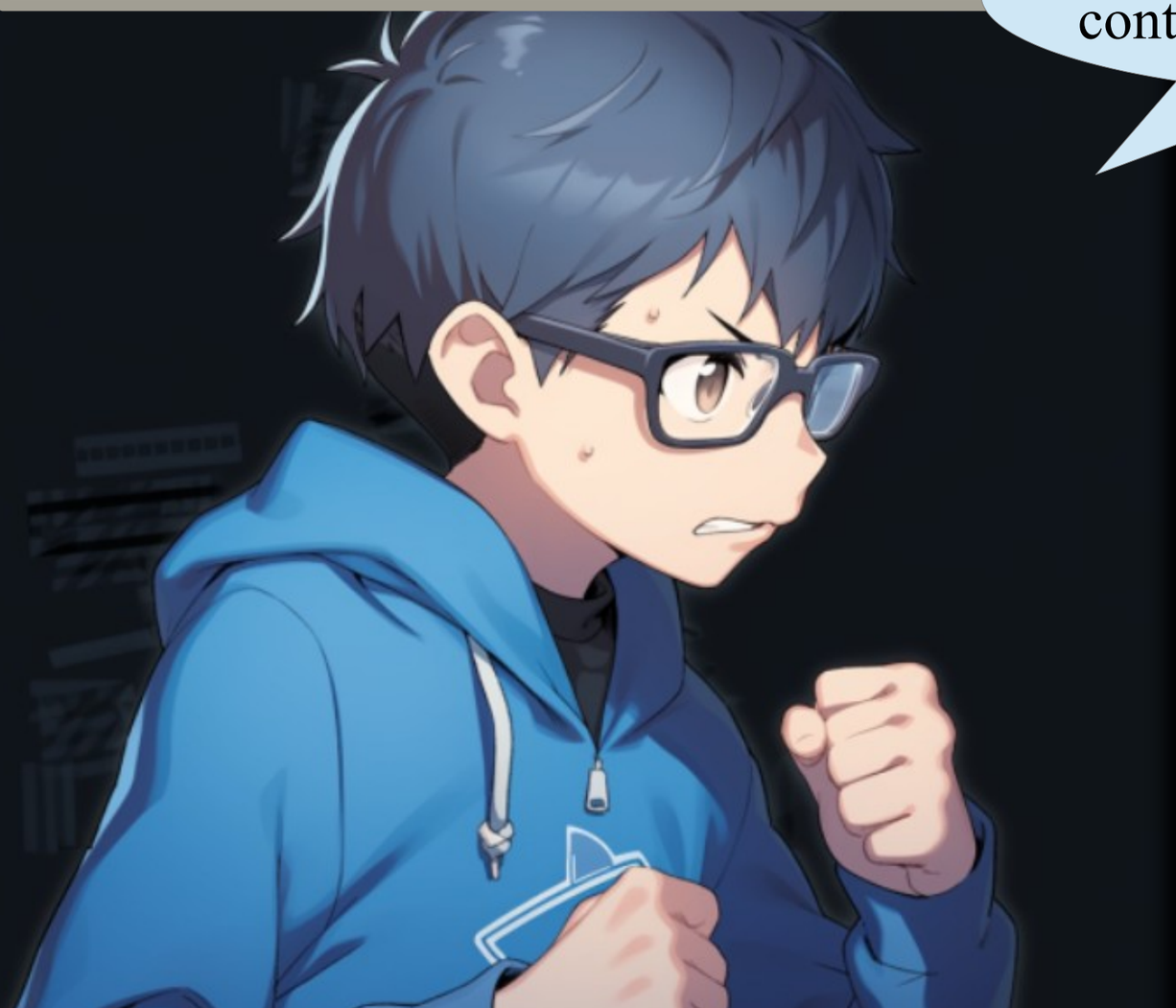


```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
}
```



To create an optional with content we have two options

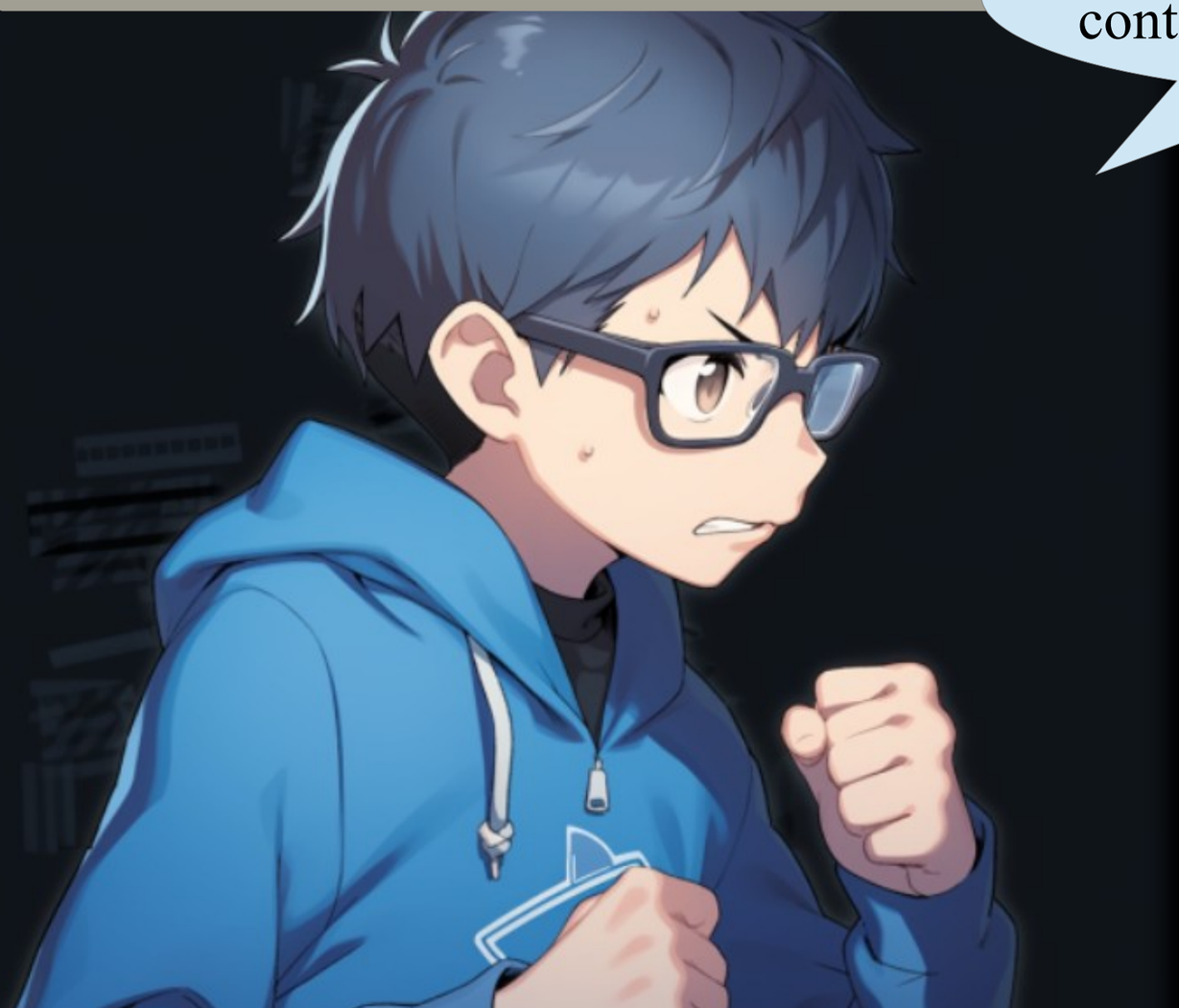
```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
}
```



To create an optional with content we have two options

‘of’ and ‘ofNullable’

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
}
```



To create an optional with content we have two options

‘of’ and ‘ofNullable’

‘ofNullable’ handles null by returning an empty optional

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
}
```

To create an optional with content we have two options

‘of’ and ‘ofNullable’

‘ofNullable’ handles null by returning an empty optional

In both cases, ‘new Some’ is only in the code of Optional.

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
}
```

To create an optional with content we have two options

‘of’ and ‘ofNullable’

‘ofNullable’ handles null by returning an empty optional

In both cases, ‘new Some’ is only in the code of Optional. So the user does not need to ever see Some directly


Is there a value in this box?


Is there a value in this box?
How do I get it out?


Is there a value in this box?
How do I get it out?
This optional is like a bomb;
how can I defuse this bomb?

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...


}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
    T get();
    T orElse(T other);
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
    T get();
    T orElse(T other);
}
```


There is more in
the full Optional


There is more in the full Optional
But here I’m asked to only discuss the essential

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

get is the simplest and
the most dangerous

```java
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

get is the simplest and the most dangerous

Note how Some.get() is implicitly defined by the autogenerated getter


‘get’ is the most dangerous


‘get’ is the most dangerous
If there is no element… kaboom!


‘get’ is the most dangerous
If there is no element… kaboom!

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

orElse is safer

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{...
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

orElse is safer

It asks us what to return in case the element is not there

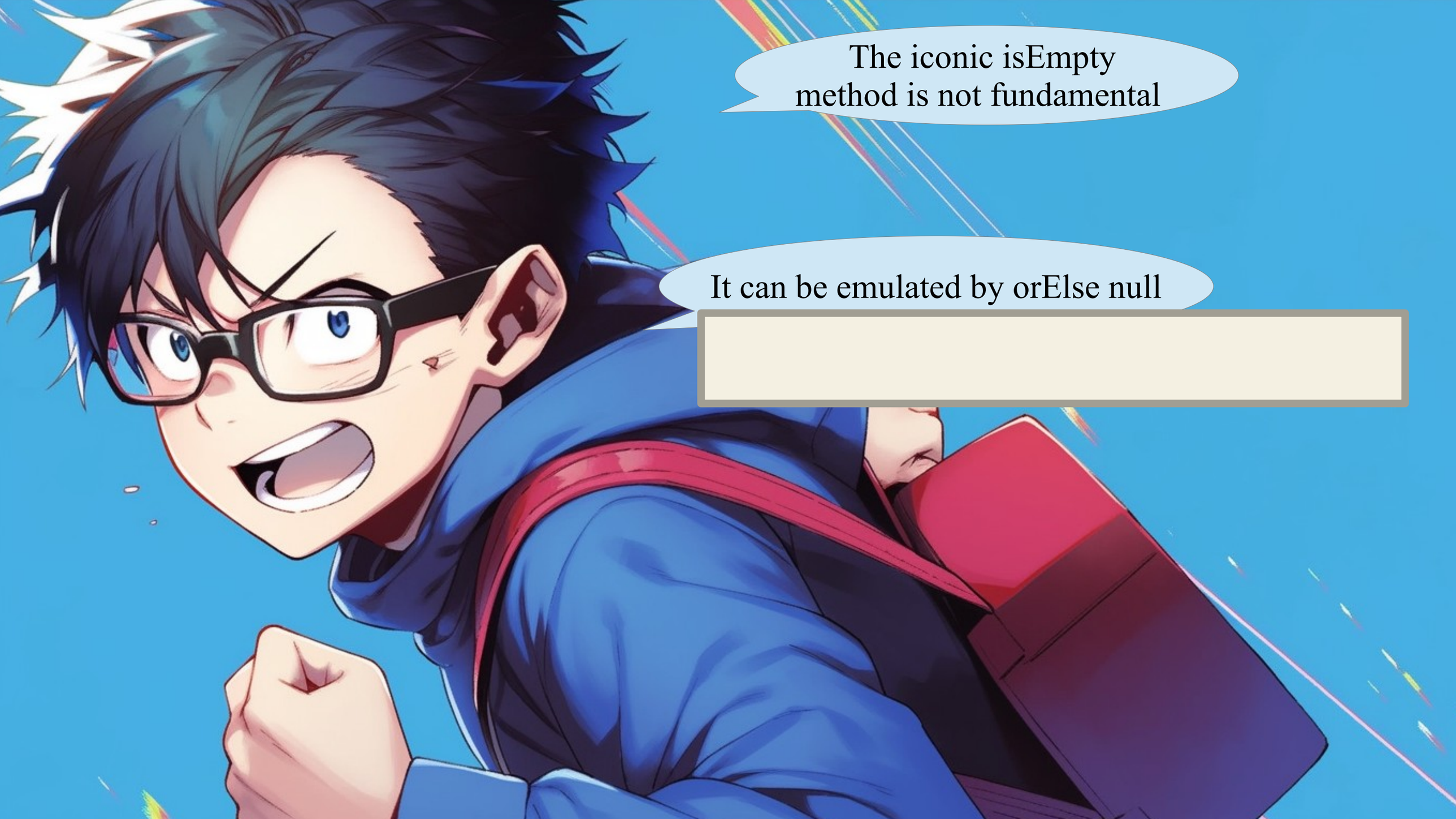

Some.orElse ignores
the parameter and just return
the content of the ‘get’ field

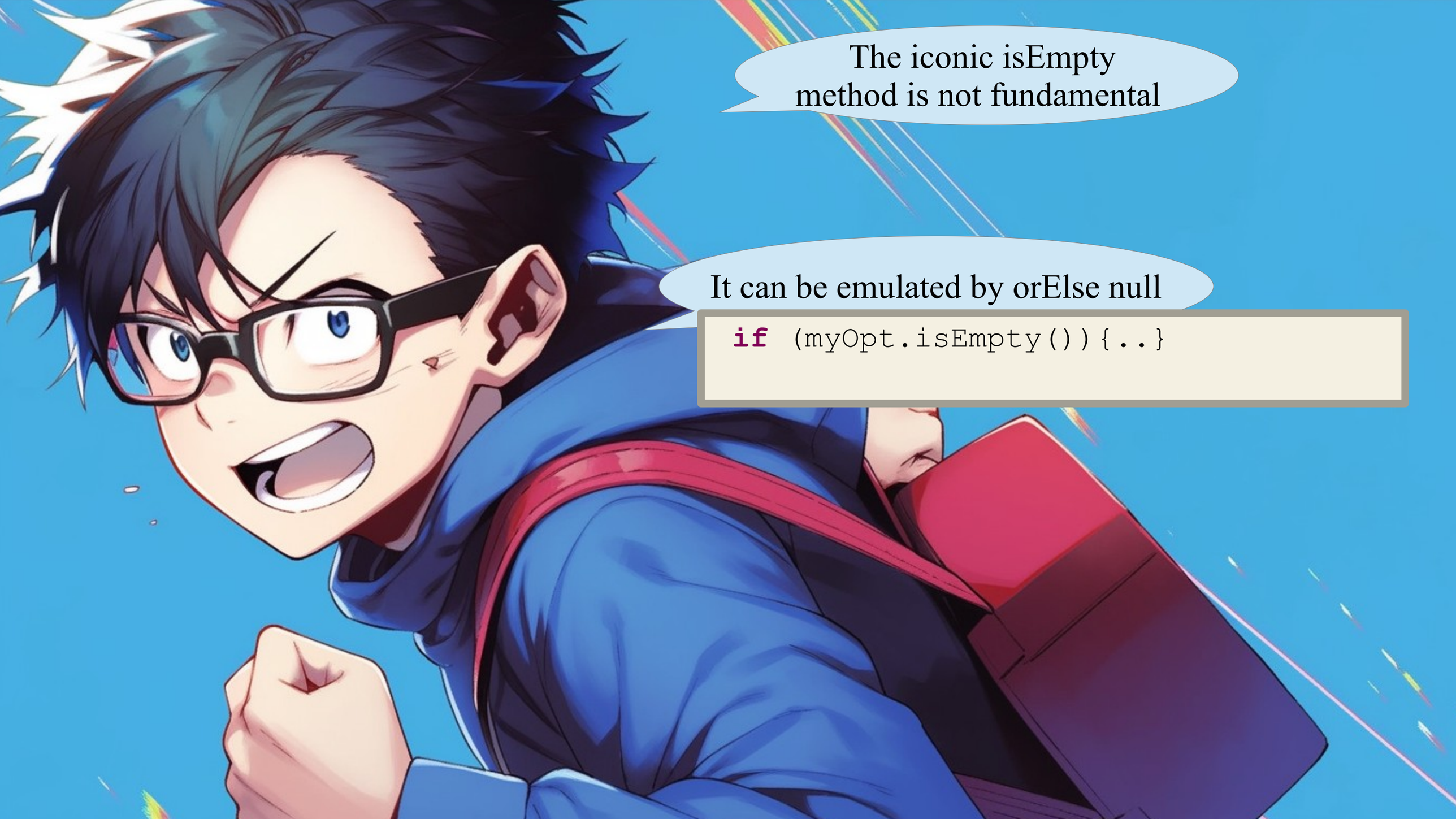

Some.orElse ignores the parameter and just return the content of the ‘get’ field
Empty.orElse just returns the parameter value

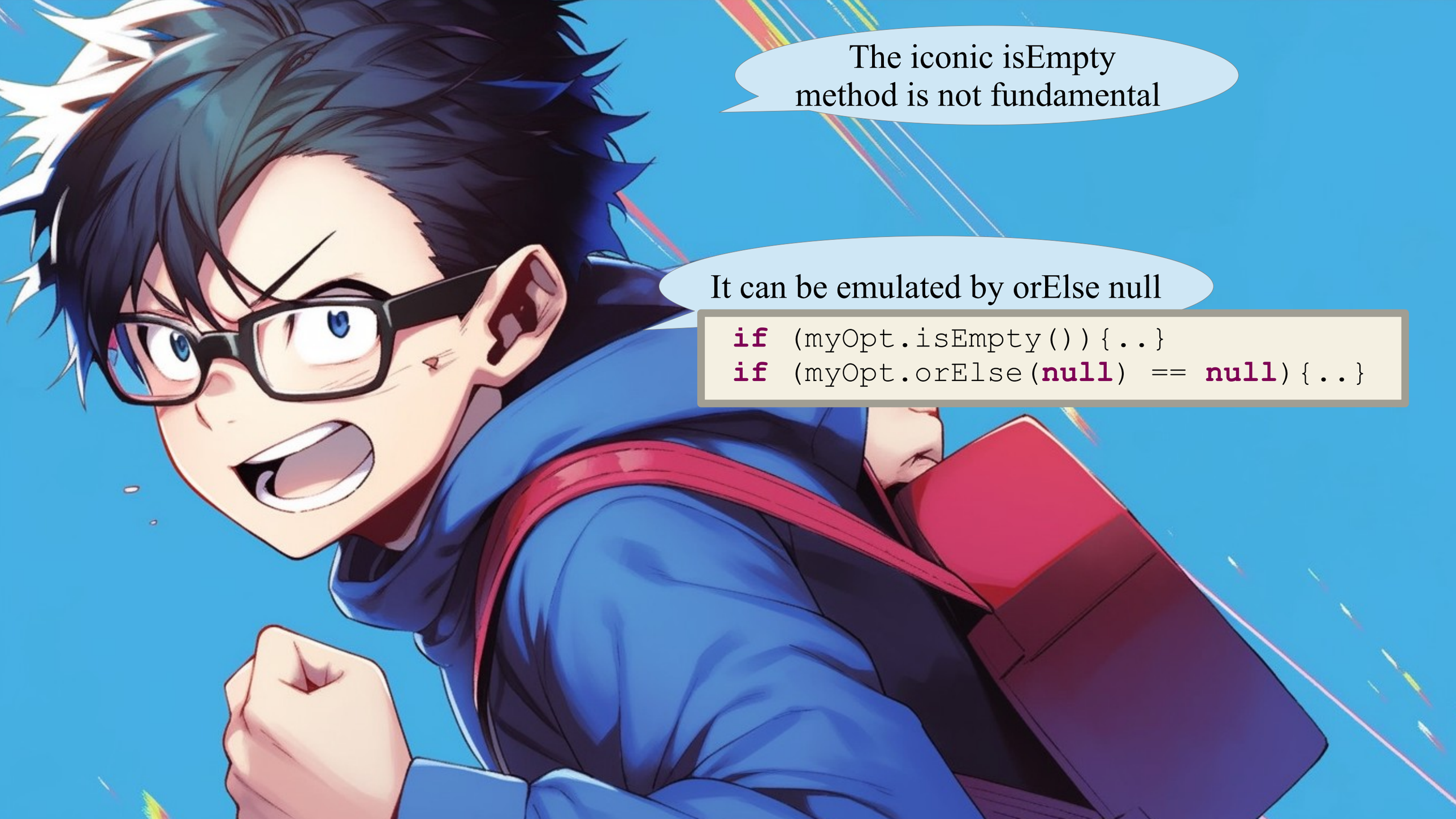

Some.orElse ignores the parameter and just return the content of the ‘get’ field
Empty.orElse just returns the parameter value
In this way, we have a result in both cases

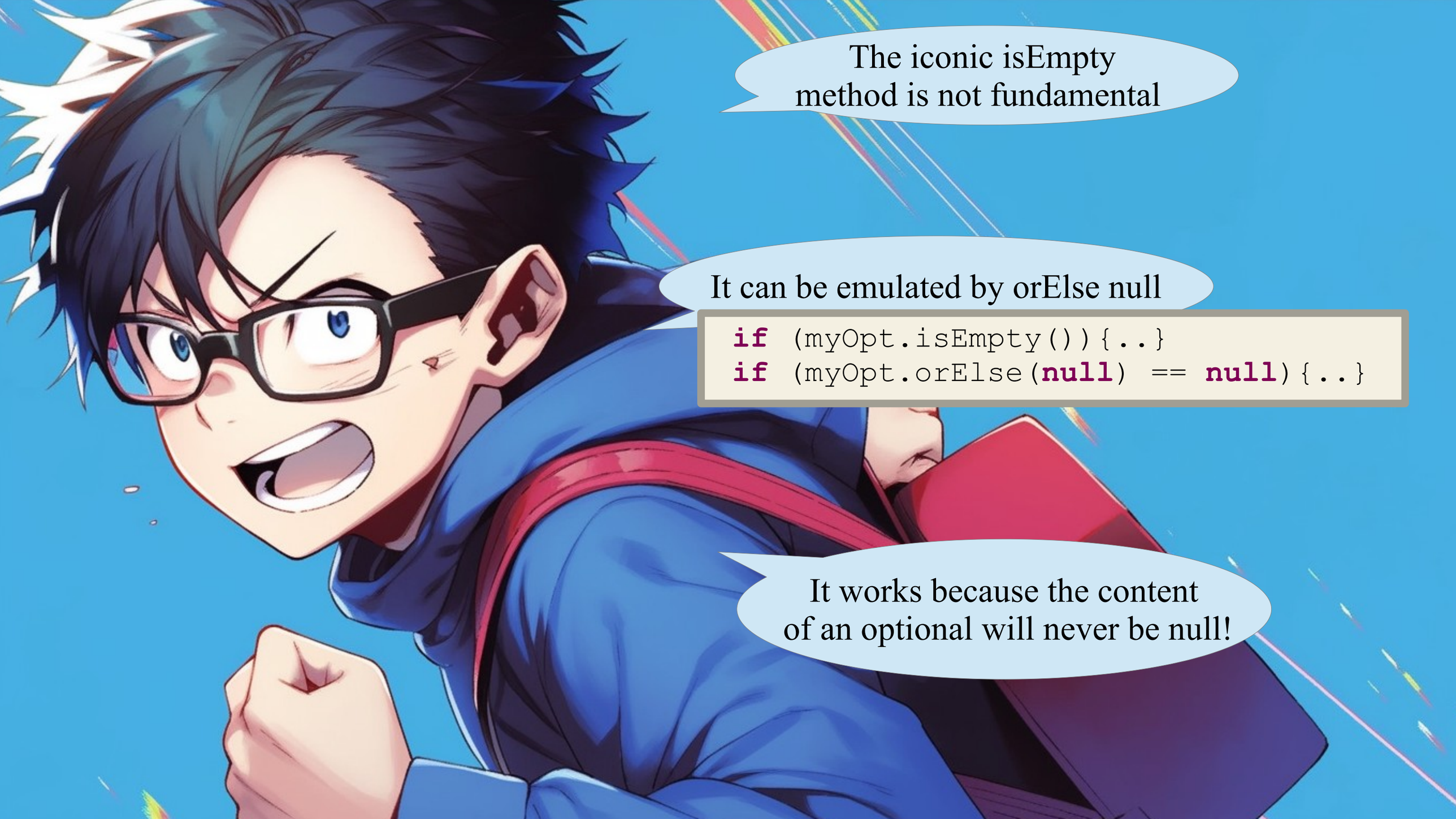

The iconic isEmpty
method is not fundamental


The iconic isEmpty method is not fundamental
It can be emulated by orElse null


The iconic isEmpty method is not fundamental
It can be emulated by orElse null


The iconic isEmpty method is not fundamental
It can be emulated by orElse null
if (myOpt.isEmpty()){..}


The iconic isEmpty method is not fundamental
It can be emulated by orElse null
if (myOpt.isEmpty()){..}
if (myOpt.orElse(null) == null){..}

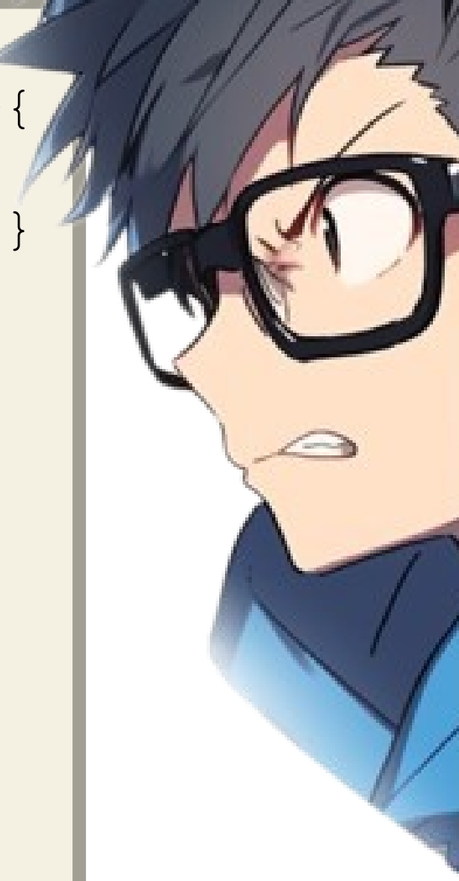

The iconic isEmpty method is not fundamental
It can be emulated by orElse null
if (myOpt.isEmpty()){..}
if (myOpt.orElse(null) == null){..}
It works because the content of an optional will never be null!

```
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{




}
```

```
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{









}

final class Empty<T> implements Optional<T>{


}

record Some<T>(T get) implements Optional<T>{

}
```

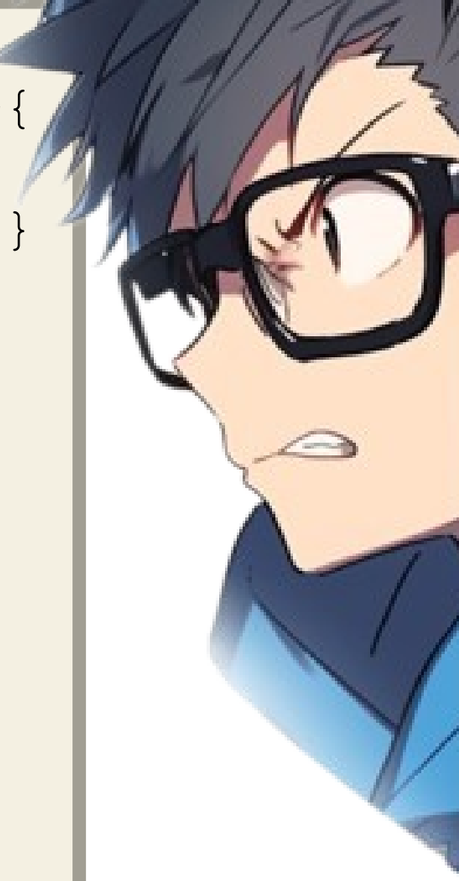
```
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }









}

final class Empty<T> implements Optional<T>{



}

record Some<T>(T get) implements Optional<T>{

}
```

```
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }









}

final class Empty<T> implements Optional<T>{



}

record Some<T>(T get) implements Optional<T>{

}
```



```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }


}

final class Empty<T> implements Optional<T>{


}

record Some<T>(T get) implements Optional<T>{

}
```

```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();

}

final class Empty<T> implements Optional<T>{



}

record Some<T>(T get) implements Optional<T>{

}
```



```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{



}

record Some<T>(T get) implements Optional<T>{

}
```

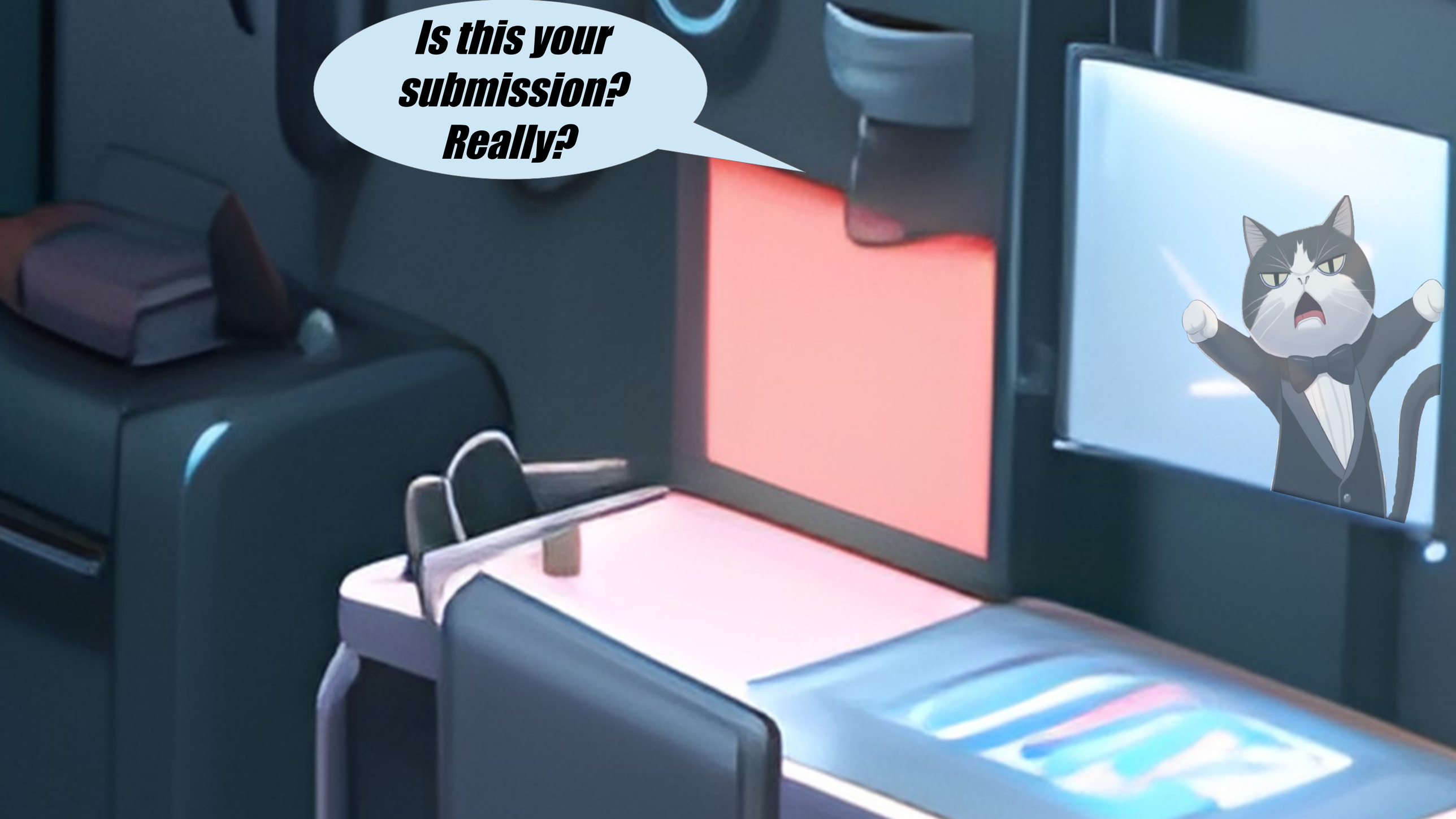
```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }

}

record Some<T>(T get) implements Optional<T>{

}
```

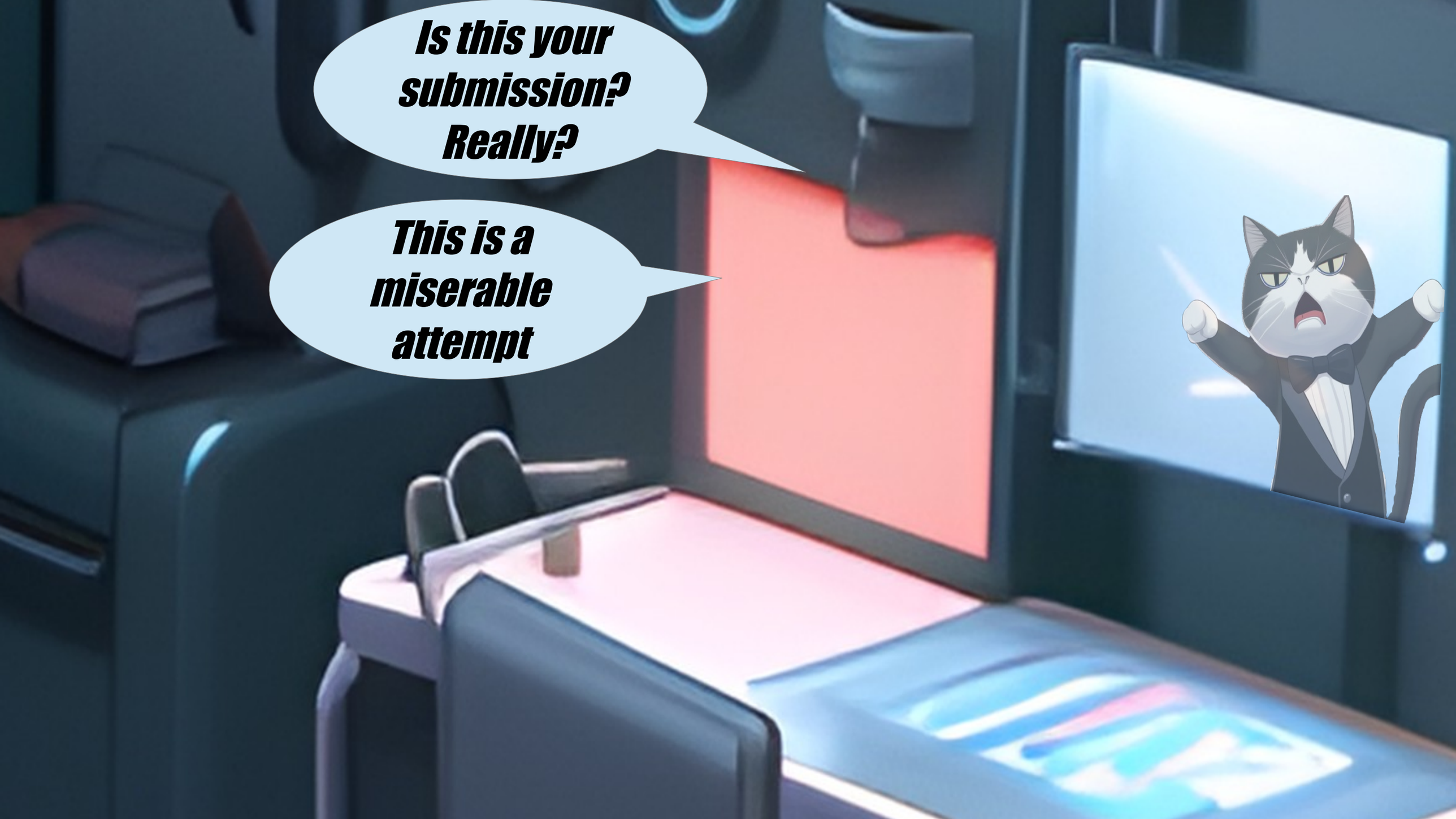
```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{

}
```

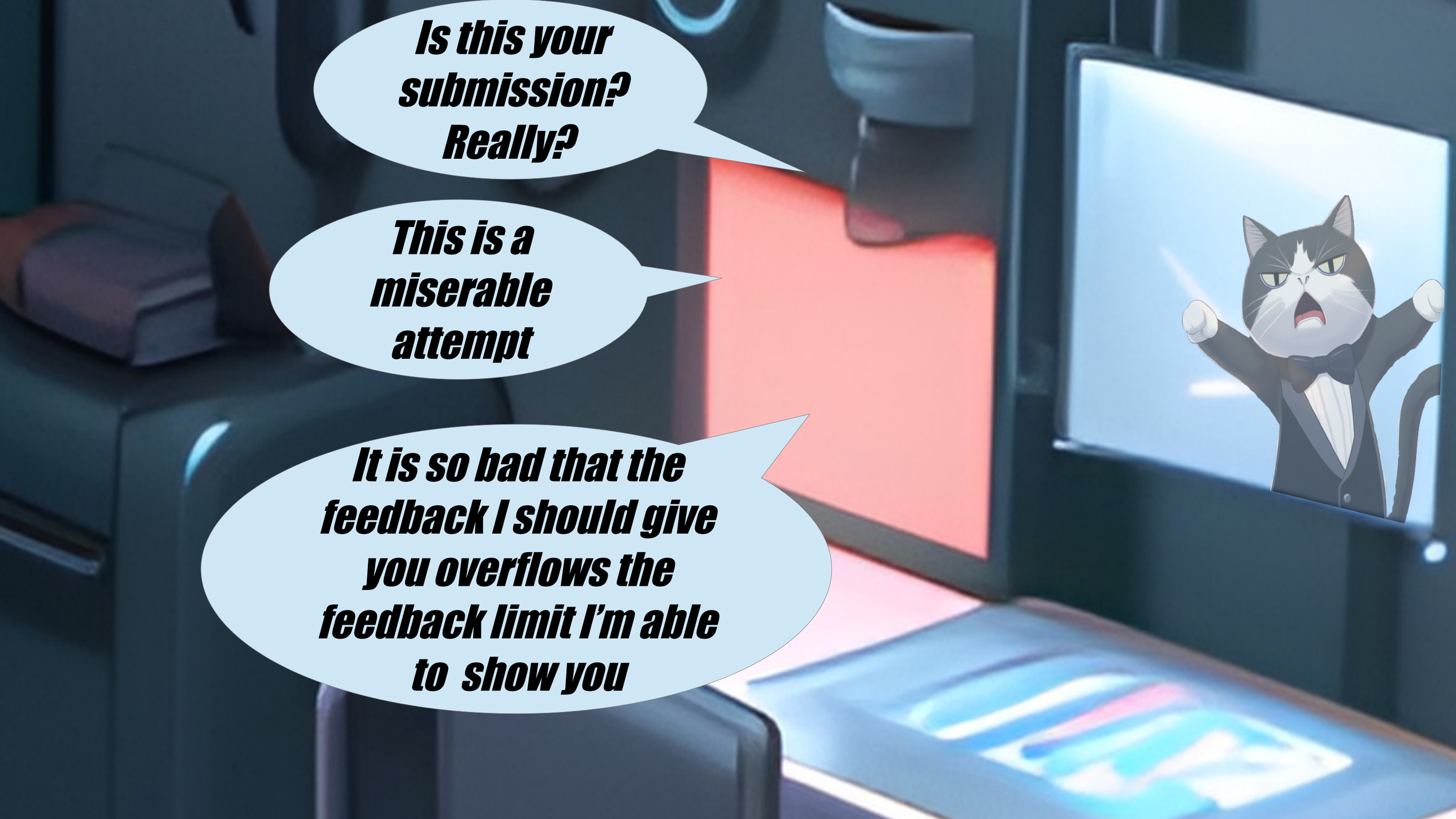
```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

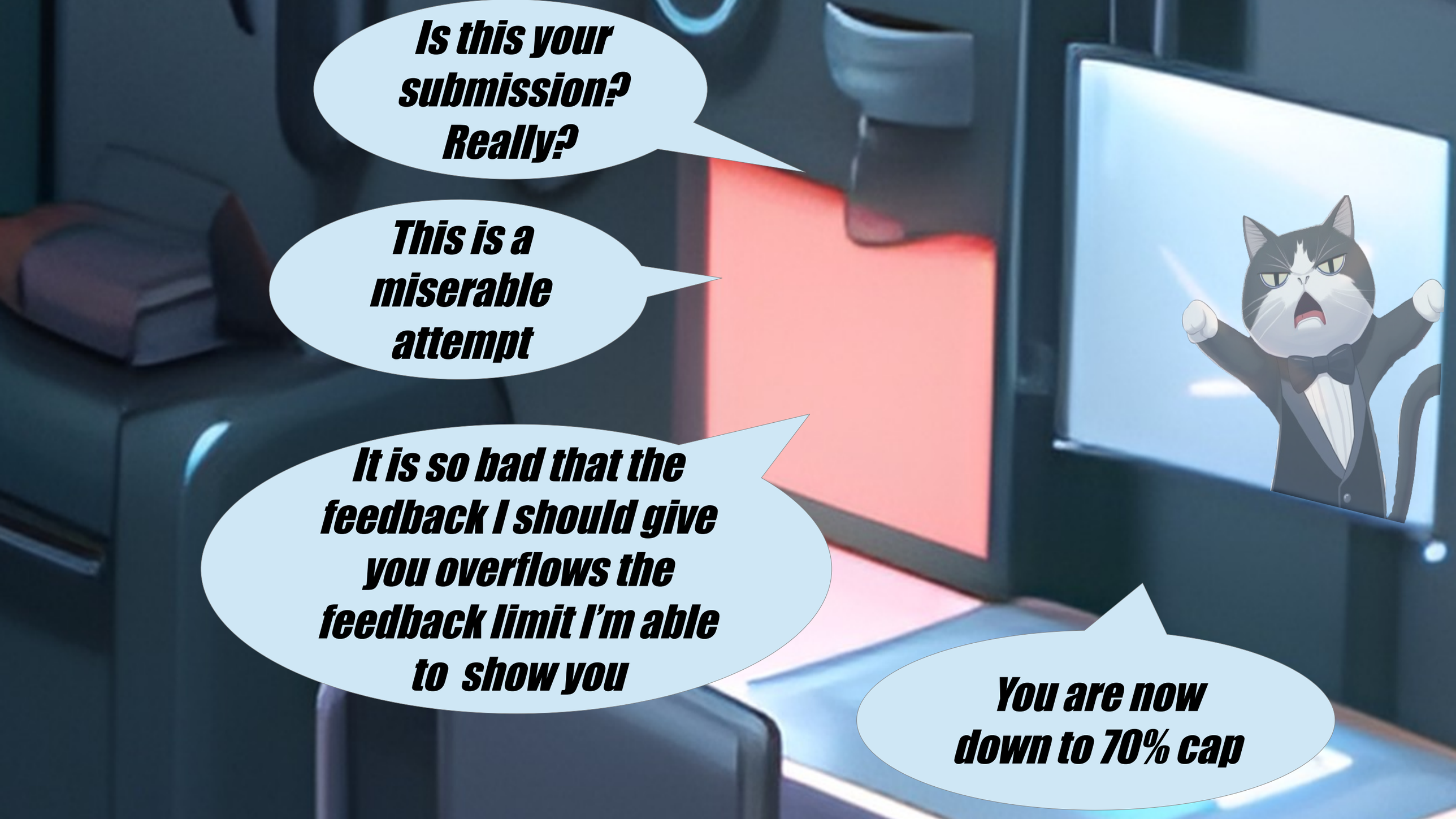
```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <E> Optional<E> empty(){ return new Empty<>(); }
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

Here it is.
I added ‘Serializable’ as asked,
and I’m sorted!


Is this your submission? Really?


Is this your submission? Really?
This is a miserable attempt

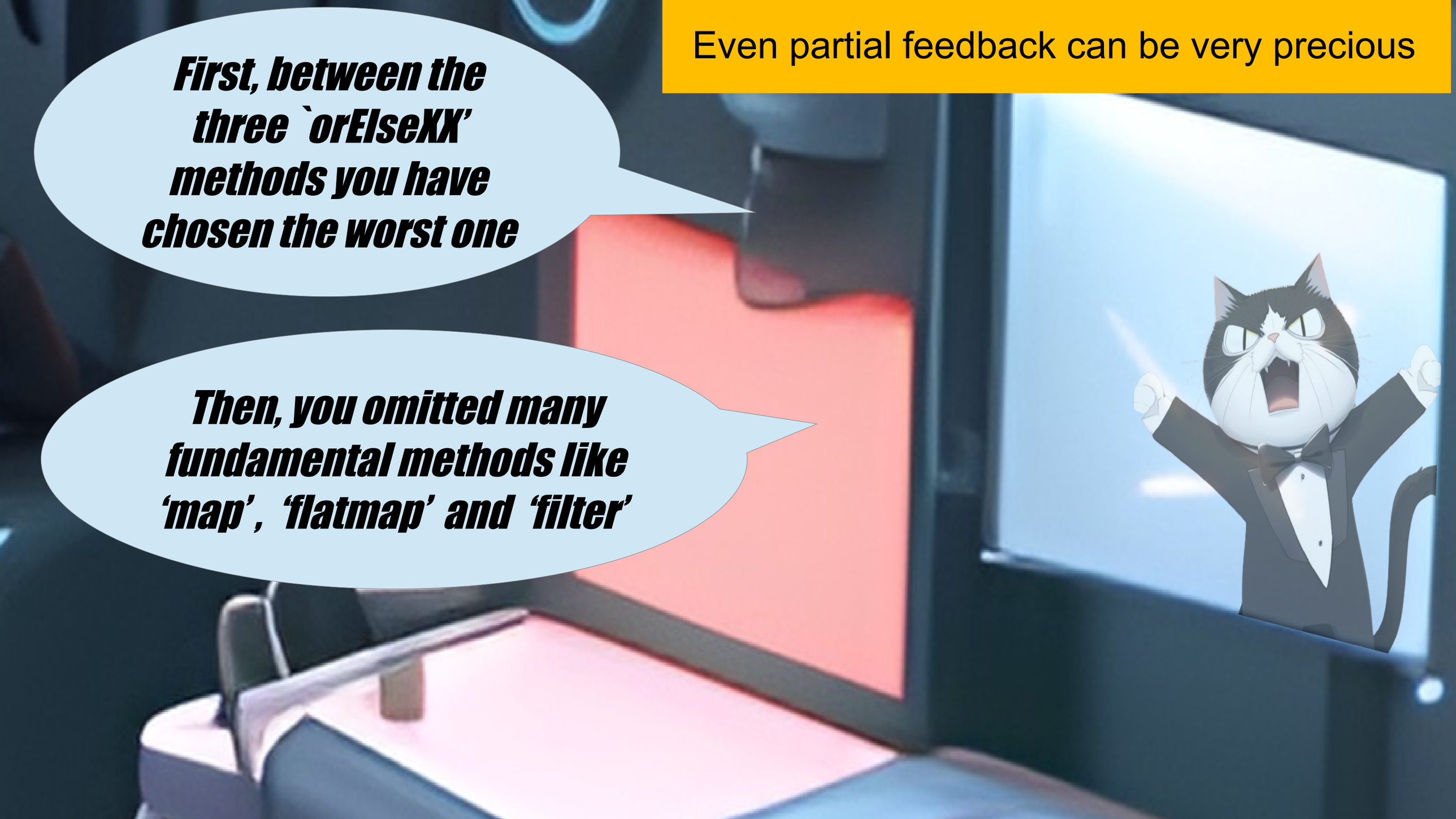

Is this your submission? Really?
This is a miserable attempt
It is so bad that the feedback I should give you overflows the feedback limit I'm able to show you


Is this your submission? Really?
This is a miserable attempt
It is so bad that the feedback I should give you overflows the feedback limit I'm able to show you
You are now down to 70% cap

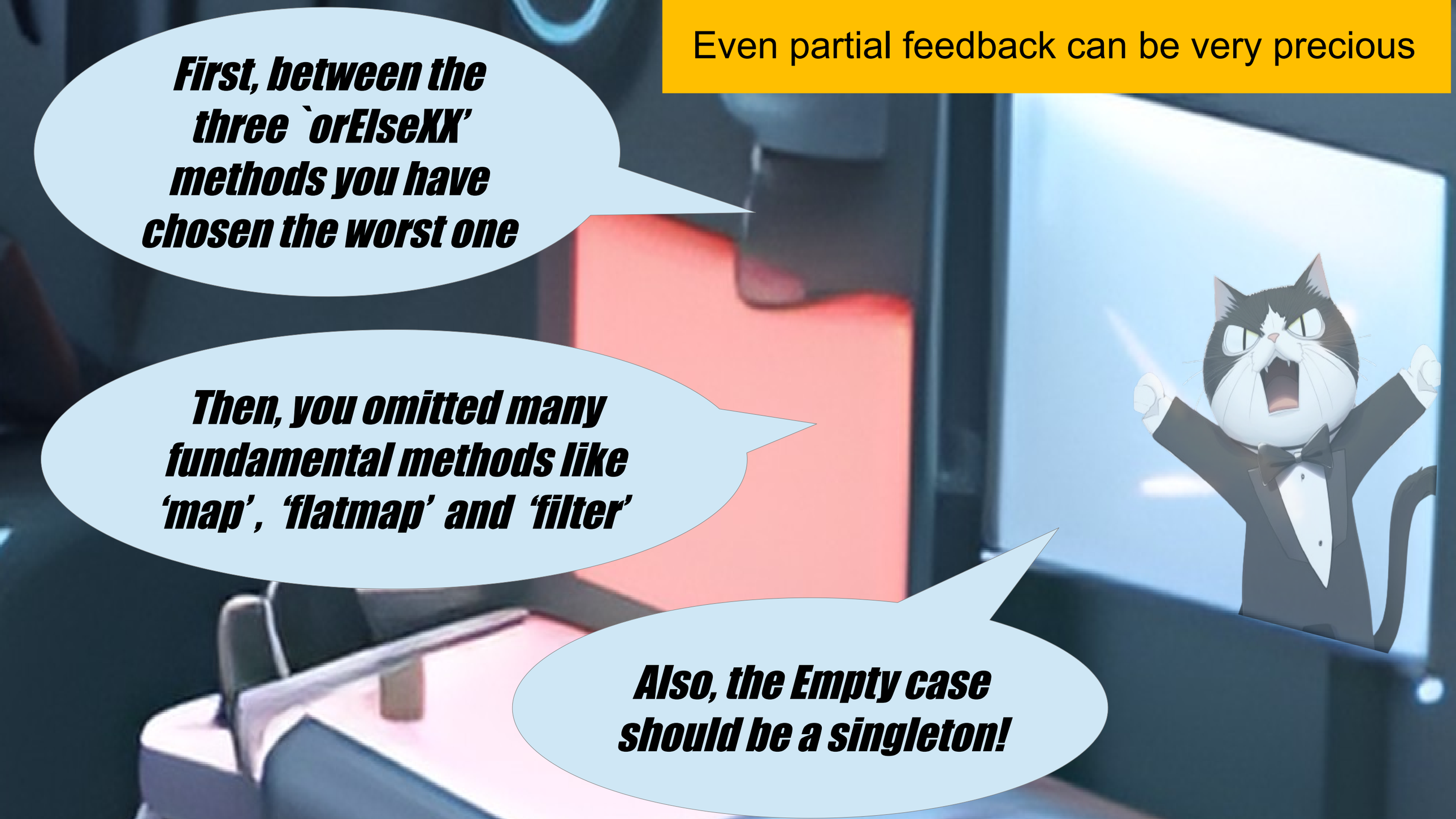
Even partial feedback can be very precious

Even partial feedback can be very precious

Even partial feedback can be very precious



Even partial feedback can be very precious



```java
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

```
public sealed interface Optional<T>
                    extends Serializable permits Empty<T>, Some<T>{
  static <T> Optional<T> of(T value){ return new Some<T>(value); }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  T get();
  T orElse(T other);
}

final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElse(T other){ return other; }
}

record Some<T>(T get) implements Optional<T>{
  public T orElse(T other){ return get; }
}
```

I have no idea how to handle this the feedback


It is time to power up again

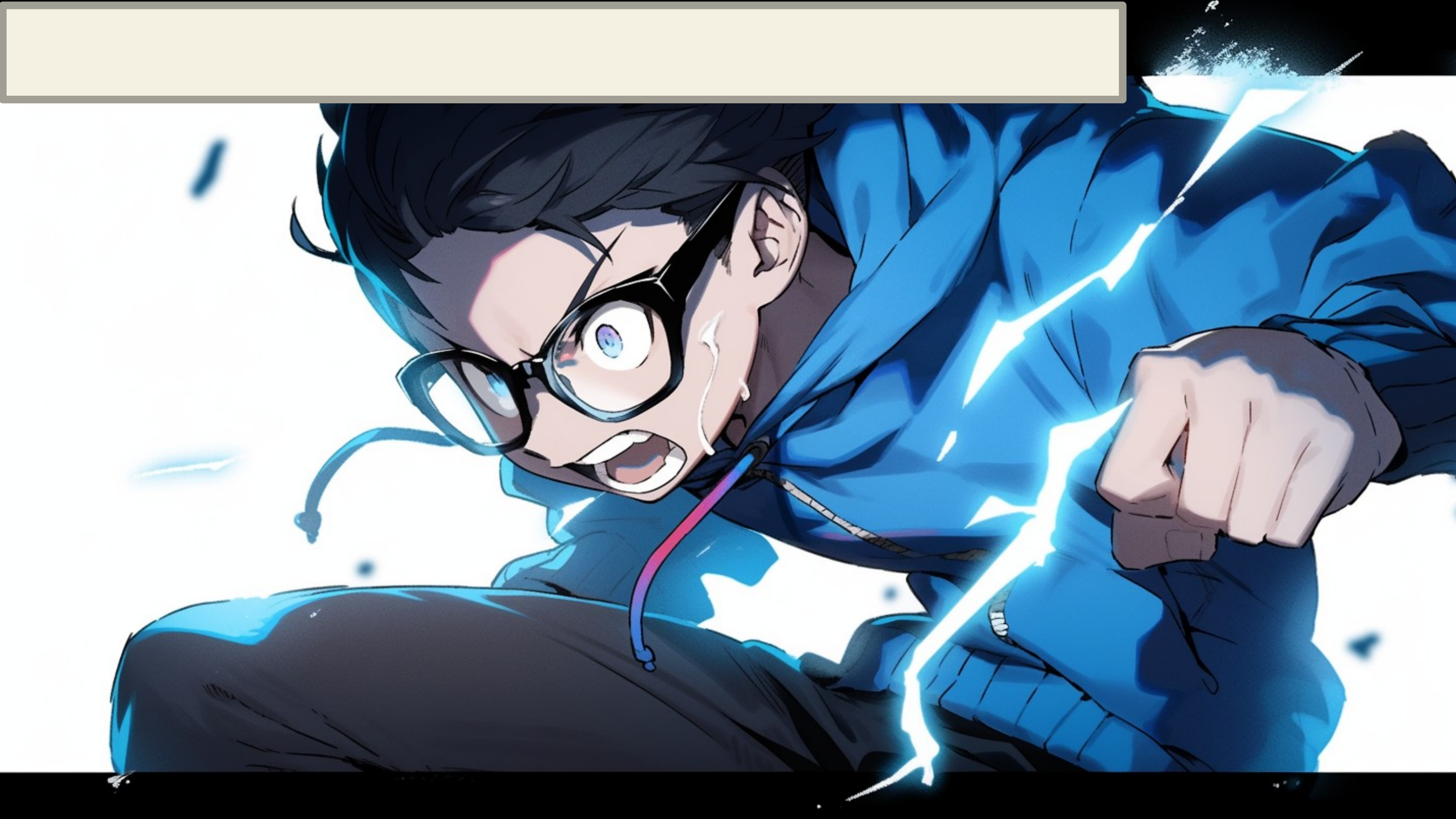

It is time to power up again
Angelic explanations!


There are 3 kinds of
‘orElse’ methods


There are 3 kinds of ‘orElse’ methods
‘orElse’, ‘orElseGet’, ‘orElseThrow’

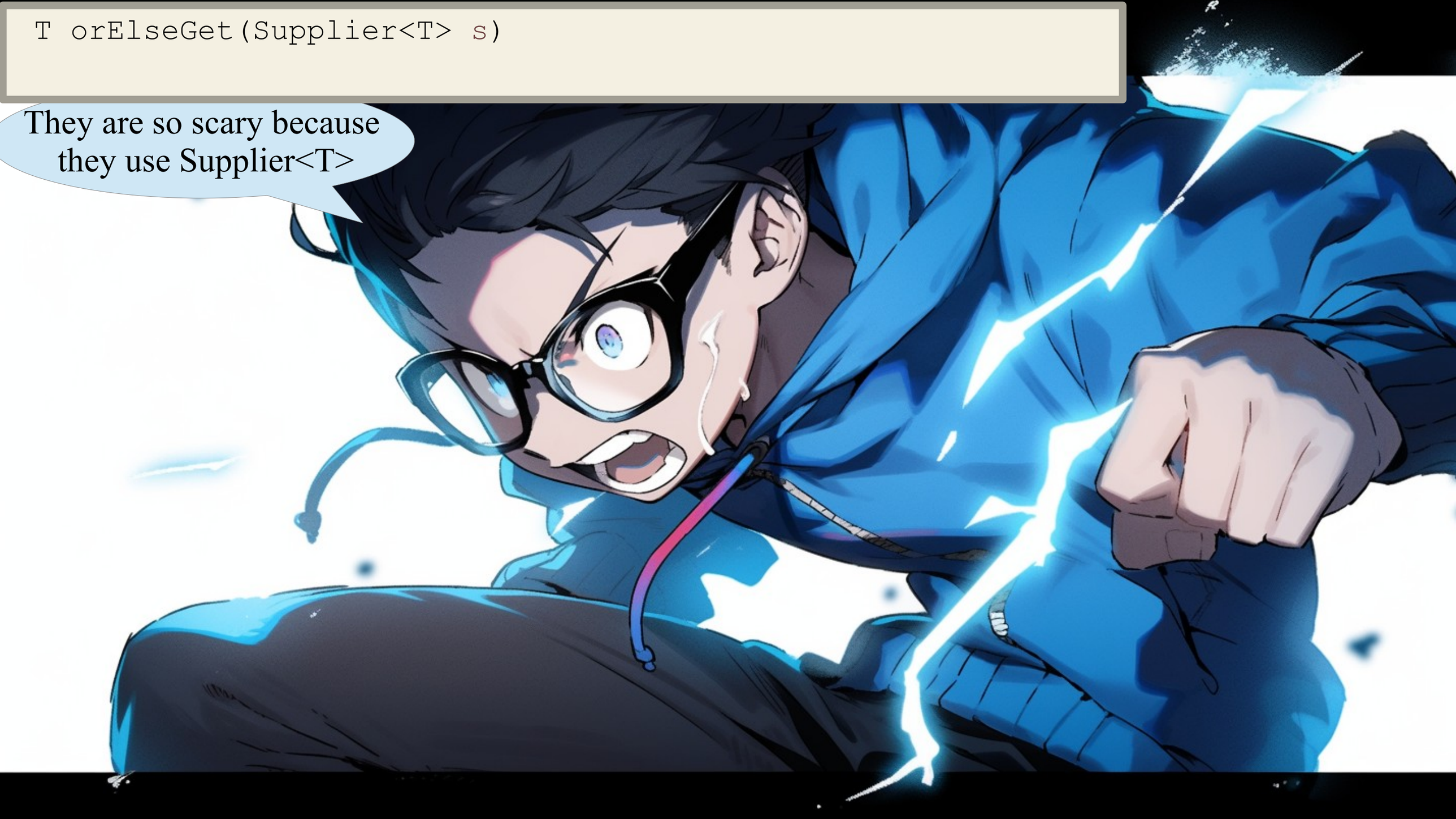

There are 3 kinds of
‘orElse’ methods
‘orElse’,
‘orElseGet’,
‘orElseThrow’
I used ‘orElse’ because
I never understood the others


I guess it is do or
die time again!

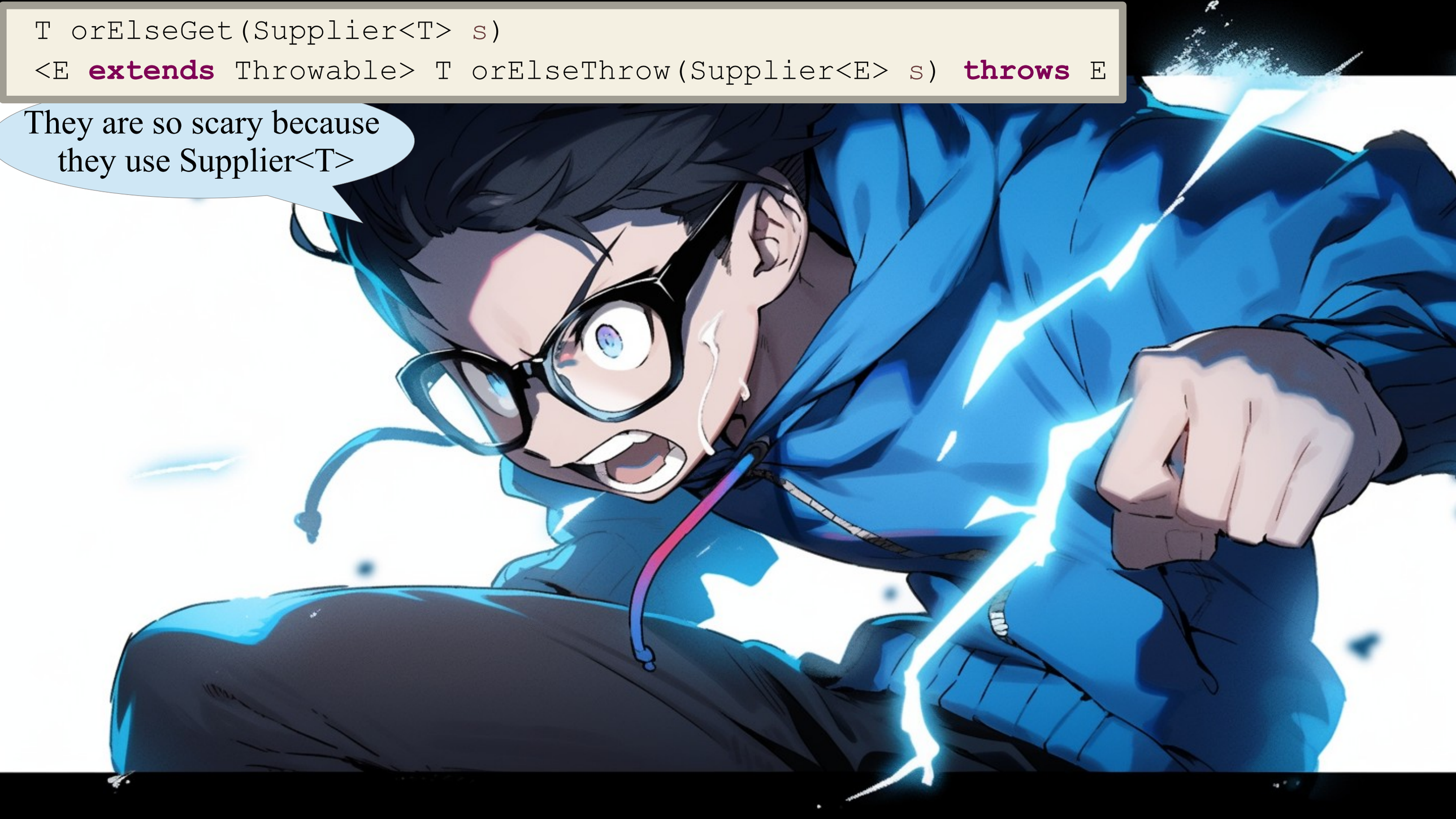



They are so scary because they use Supplier<T>

```
T orElseGet(Supplier<T> s)
```



```
T orElseGet(Supplier<T> s)
<E extends Throwable> T orElseThrow(Supplier<E> s) throws E
```



```
interface Supplier<T>{ T get(); }
```

```
interface Supplier<T>{ T get(); }
```


Why would I ever want it?


Why would I ever want it?
What is wrong with just ‘orElse’?!


I did it.
I broke trough!


I did it.
I broke trough!
Now, what lays
on the other side
of my fears?

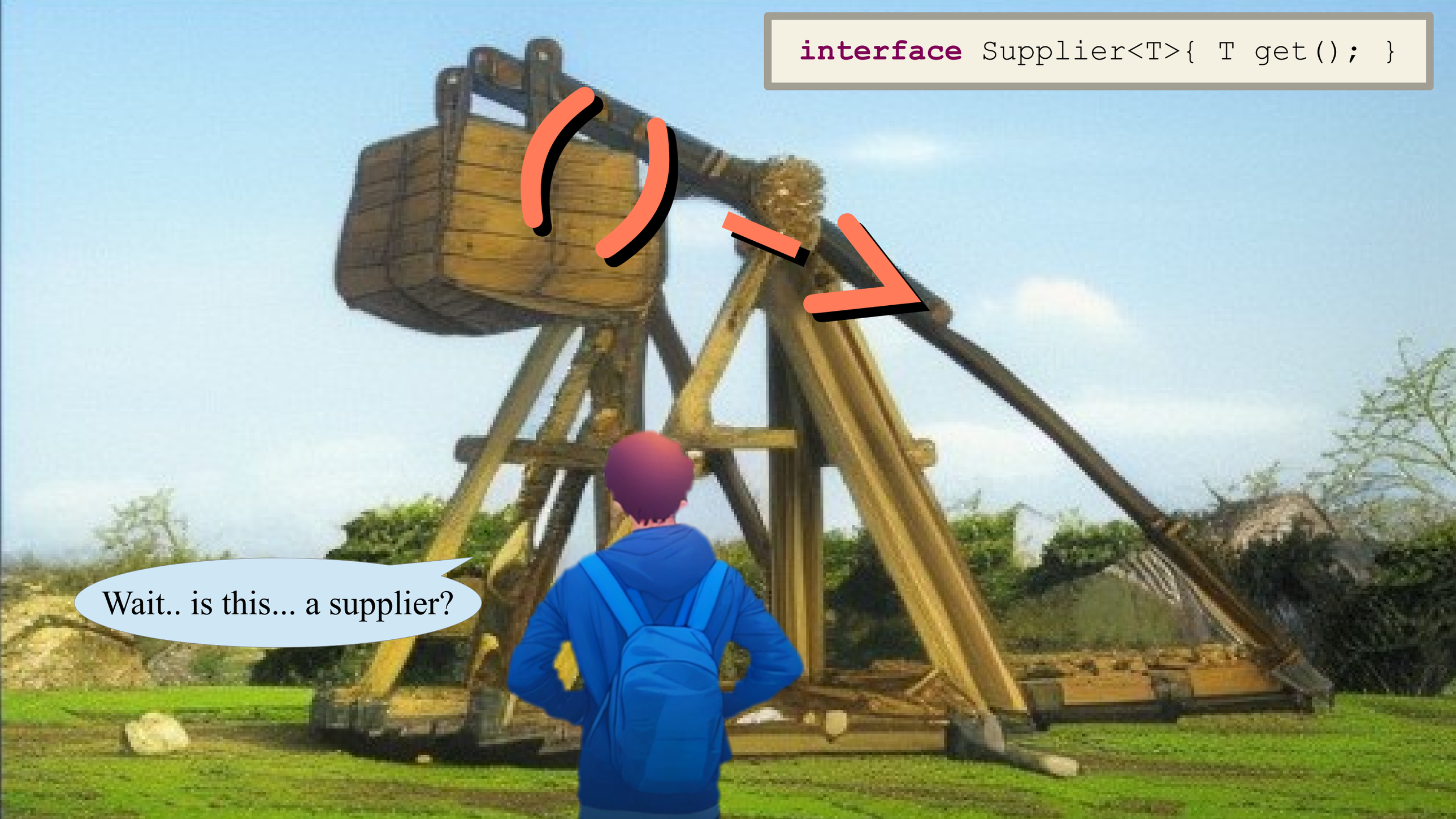

Well… this is…
unexpected

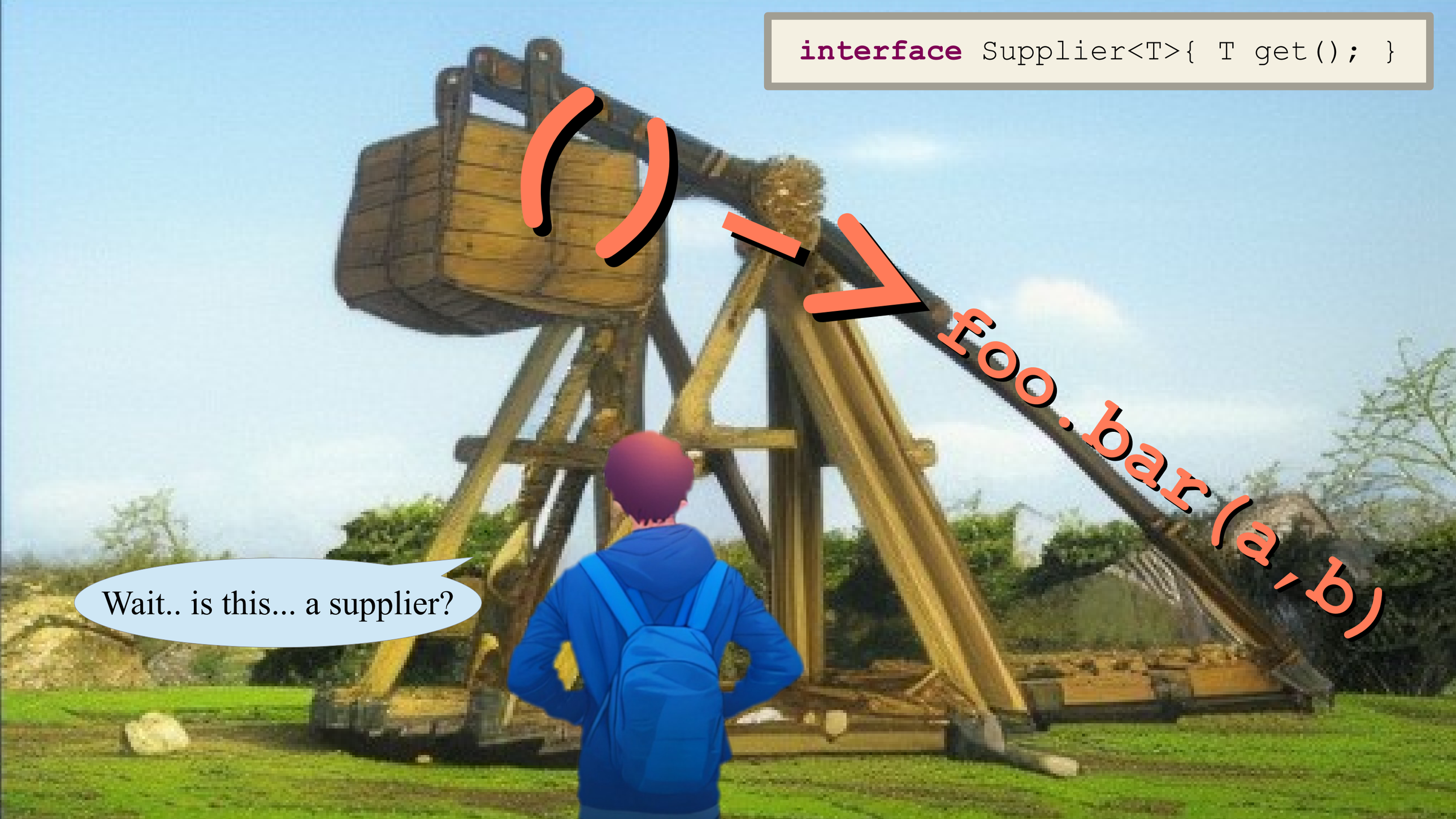

Why a trebuchet here?

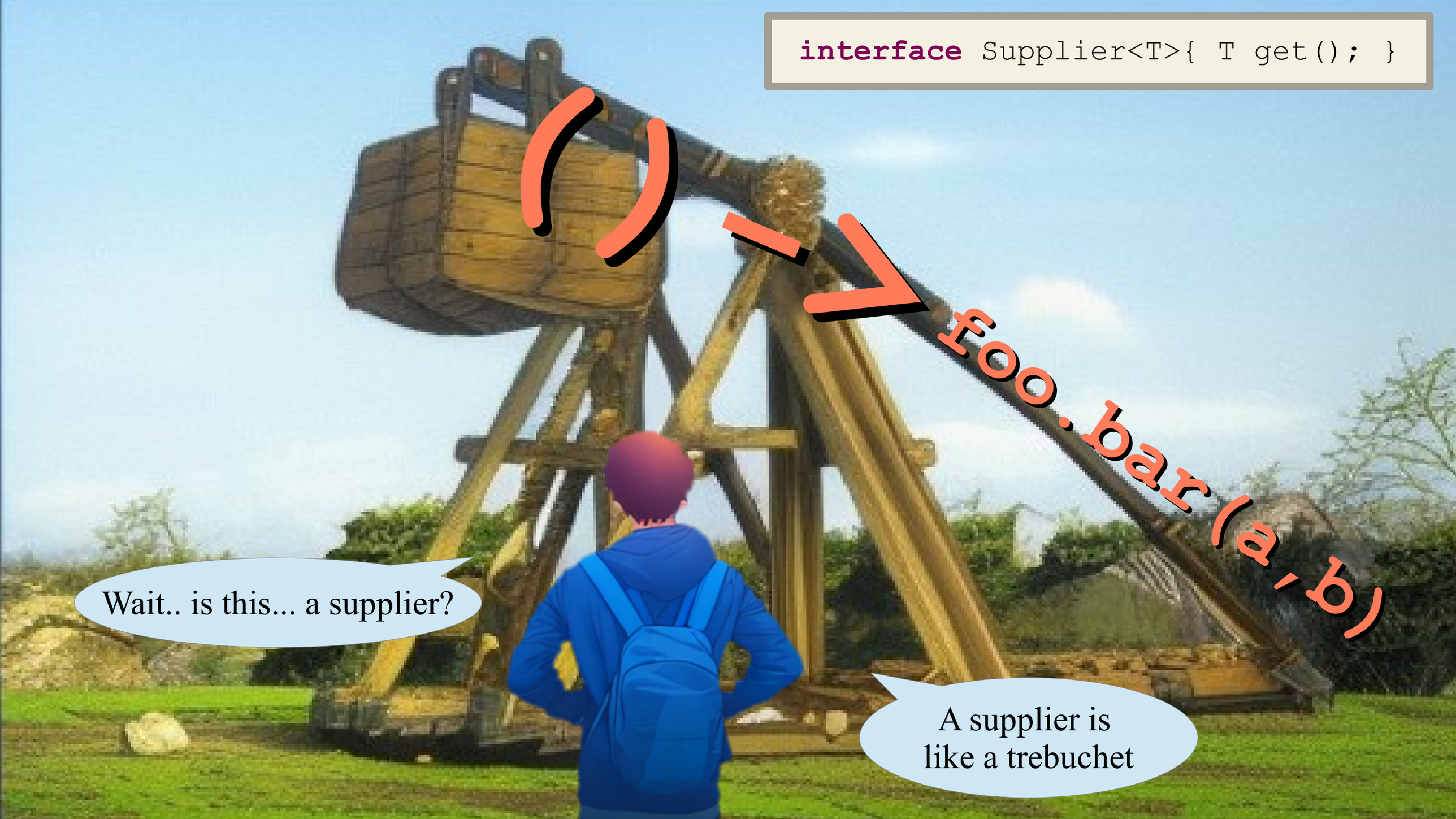

Why a trebuchet here?
What am I going to do with it


interface Supplier<T>{ T get(); }


interface Supplier<T>{ T get(); }
Wait.. is this... a supplier?


interface Supplier<T>{ T get(); }
()->
Wait.. is this... a supplier?


interface Supplier<T>{ T get(); }
()->foo.bar(a,b)
Wait.. is this... a supplier?


interface Supplier<T>{ T get(); }
()->foo.bar(a,b)
Wait.. is this... a supplier?
A supplier is like a trebuchet


We can load a trebuchet with a payload.


We can load a trebuchet with a payload.
When we release the weight, we launch the payload.


We can load a trebuchet with a payload.
When we release the weight, we launch the payload.
Same for suppliers:


We can load a trebuchet with a payload.
When we release the weight, we launch the payload.
Same for suppliers:


```
interface Supplier<T>{ T get(); }
```


```
interface Supplier<T>{ T get(); }

Supplier<String> s= ()-> tree.toString();
```


```
interface Supplier<T>{ T get(); }

Supplier<String> s= ()-> tree.toString();
```


```
interface Supplier<T>{ T get(); }

Supplier<String> s= ()-> tree.toString();
```


A supplier works like a lazy computation


A supplier works like a lazy computation
A frozen unit of work that can be triggered only when it is truly needed

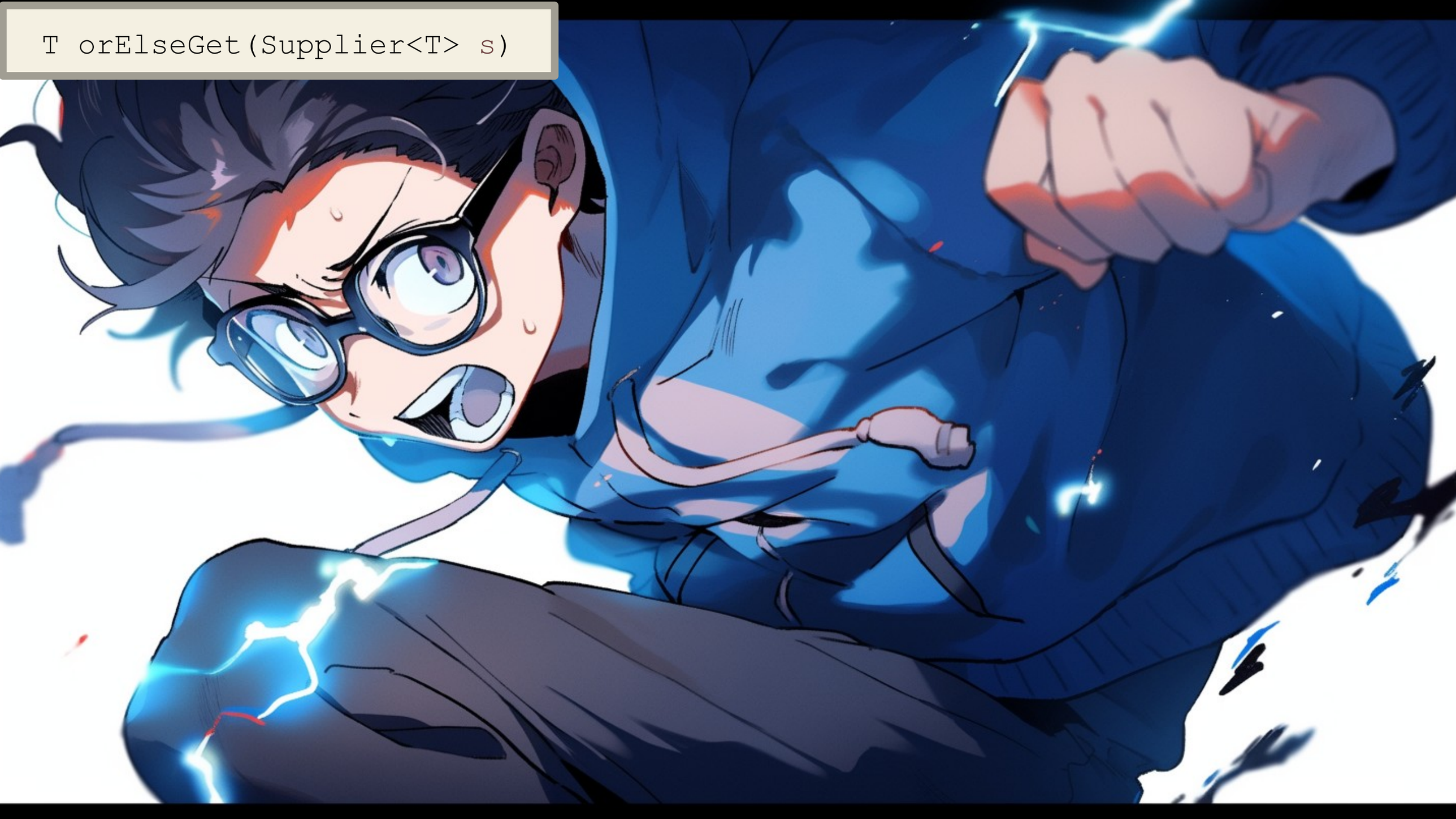

A supplier works like a lazy computation
A frozen unit of work that can be triggered only when it is truly needed
With ‘Optional.orElse’ we need to provide a value for the empty case


A supplier works like a lazy computation
A frozen unit of work that can be triggered only when it is truly needed
With ‘Optional.orElse’ we need to provide a value for the empty case
To provide a value, we need to have a value. To have a value, we need to compute it

```
T orElseGet(Supplier<T> s)
```

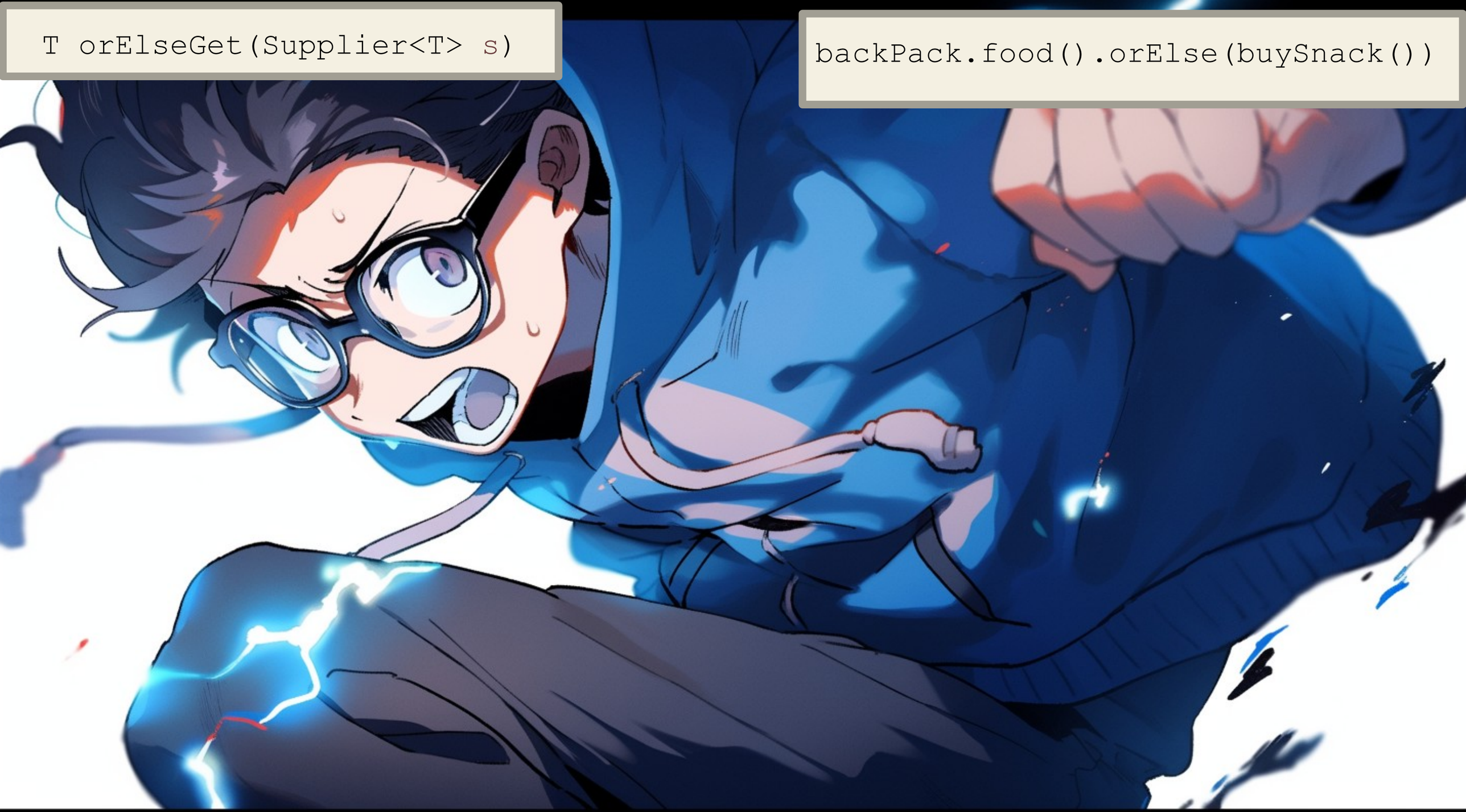
```
T orElseGet(Supplier<T> s)
```

```
T orElseGet(Supplier<T> s)
```



```
T orElseGet(Supplier<T> s)
```

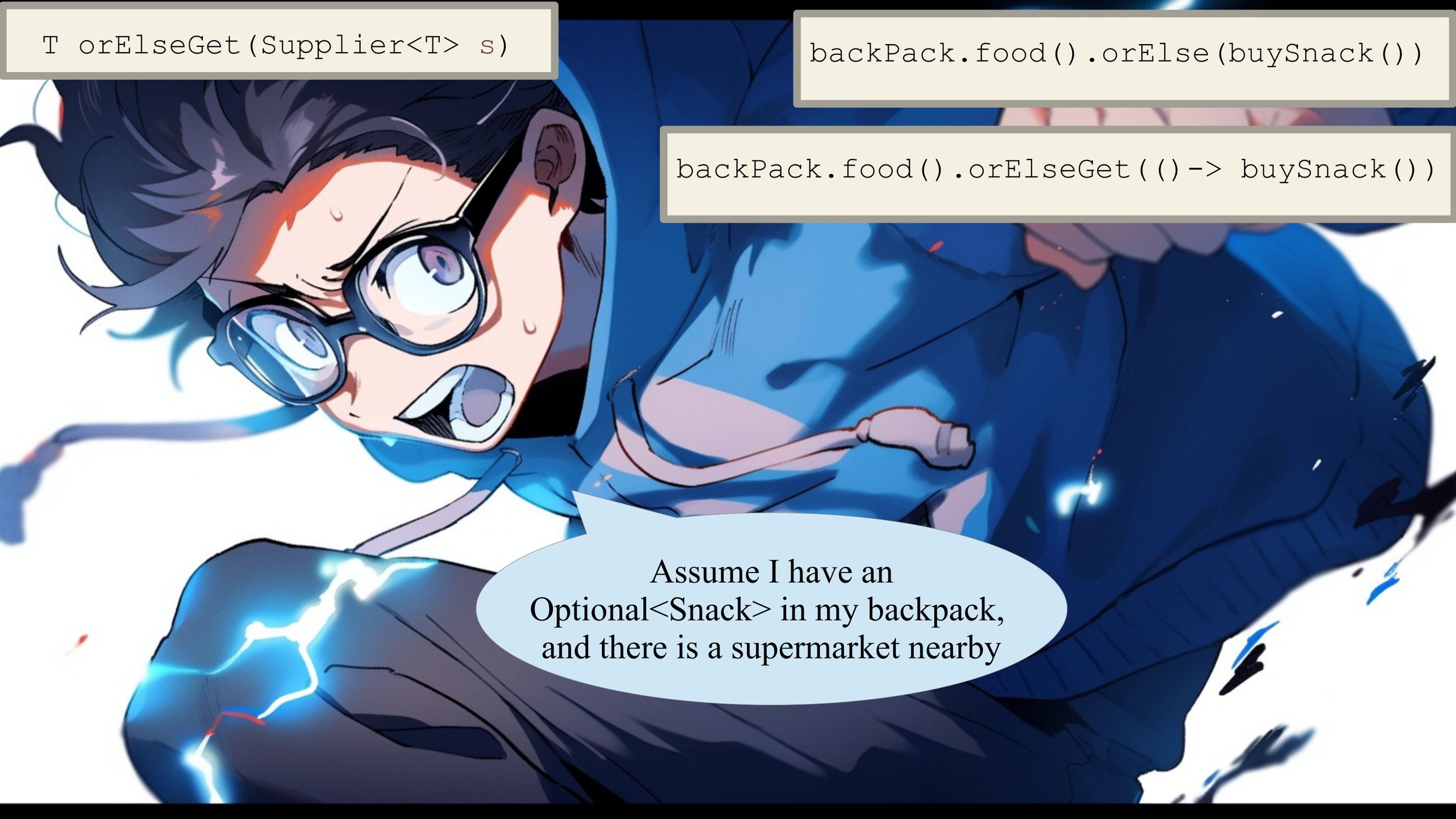
```
backPack.food().orElse(buySnack())
```



```
T orElseGet(Supplier<T> s)
```

```
backPack.food().orElse(buySnack())
```

```
backPack.food().orElseGet(()-> buySnack())
```


T orElseGet(Supplier<T> s)
backPack.food().orElse(buySnack())
backPack.food().orElseGet(()-> buySnack())
Assume I have an
Optional<Snack> in my backpack,
and there is a supermarket nearby

```
T orElseGet(Supplier<T> s)
```

```
backPack.food().orElse(buySnack())
```

```
backPack.food().orElseGet(()-> buySnack())
```

```
T orElseGet(Supplier<T> s)
```

```
backPack.food().orElse(buySnack())
```

```
backPack.food().orElseGet(()-> buySnack())
```

```
T orElseGet(Supplier<T> s)
```

```
backPack.food().orElse(buySnack())
```

```
backPack.food().orElseGet(()-> buySnack())
```

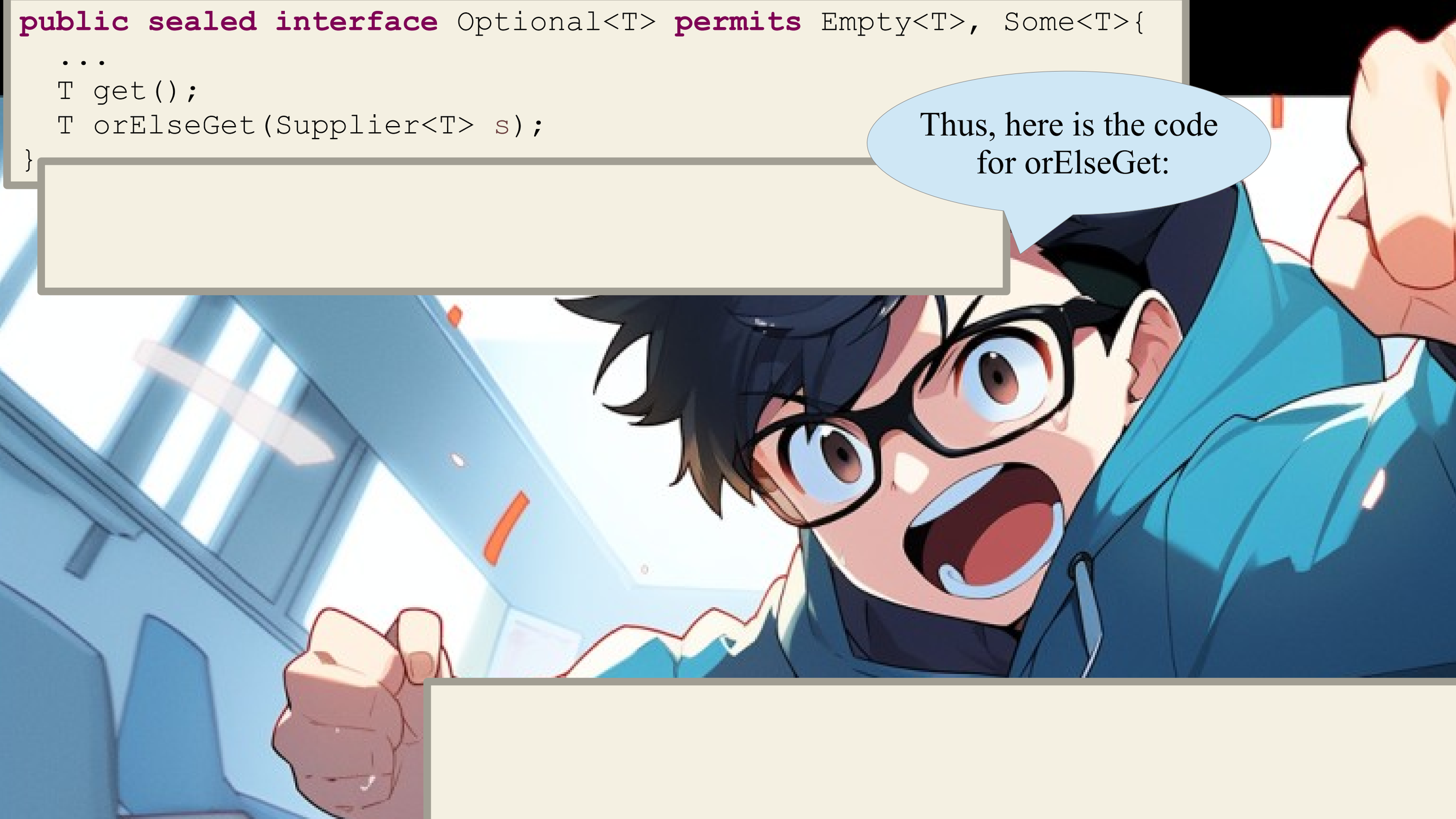
```
T orElseGet(Supplier<T> s)
```

```
backPack.food().orElse(buySnack())
```

```
backPack.food().orElseGet(()-> buySnack())
```

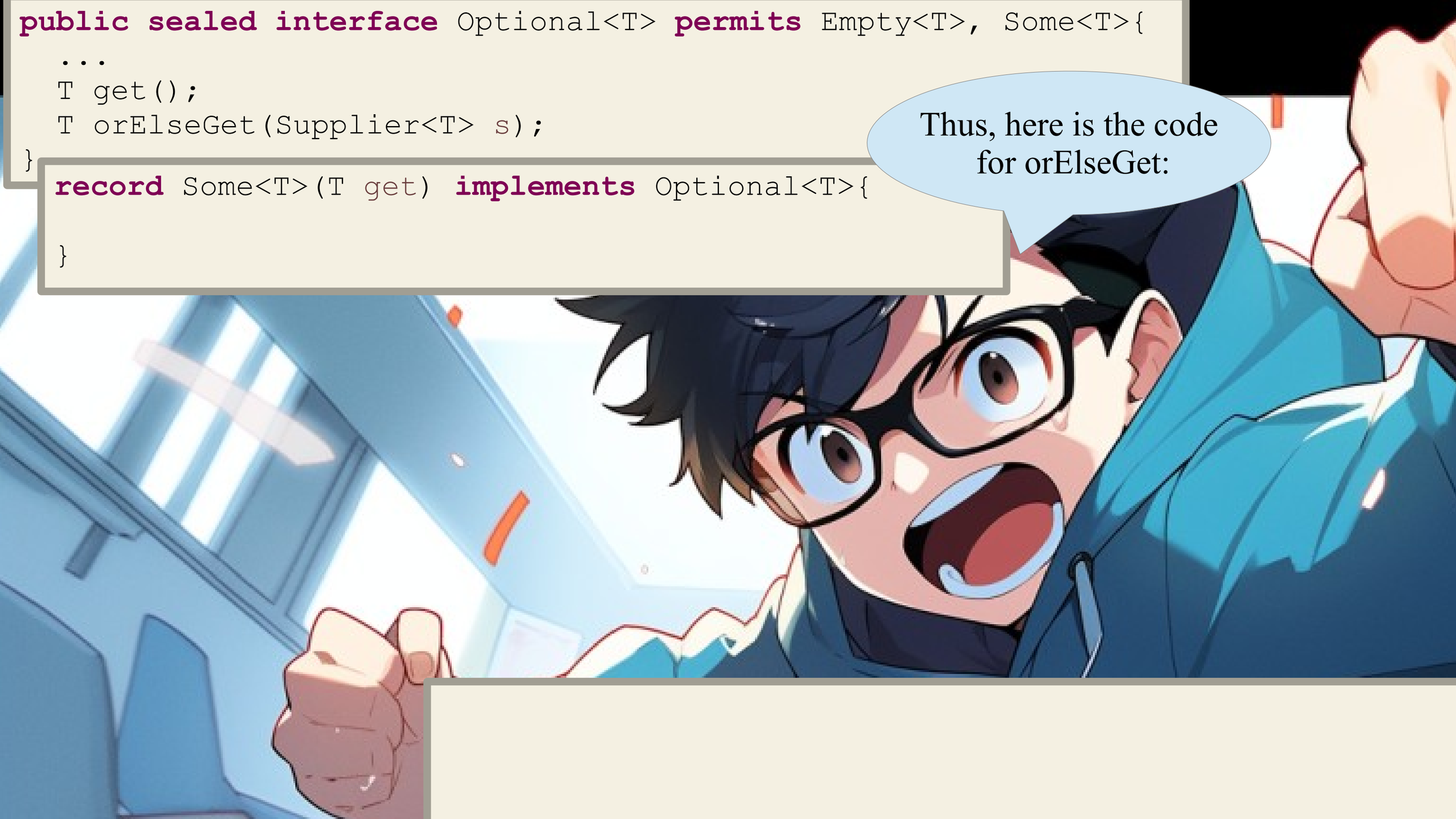

Thus, here is the code for orElseGet:

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   ...
   T get();
   T orElseGet(Supplier<T> s);
}
```



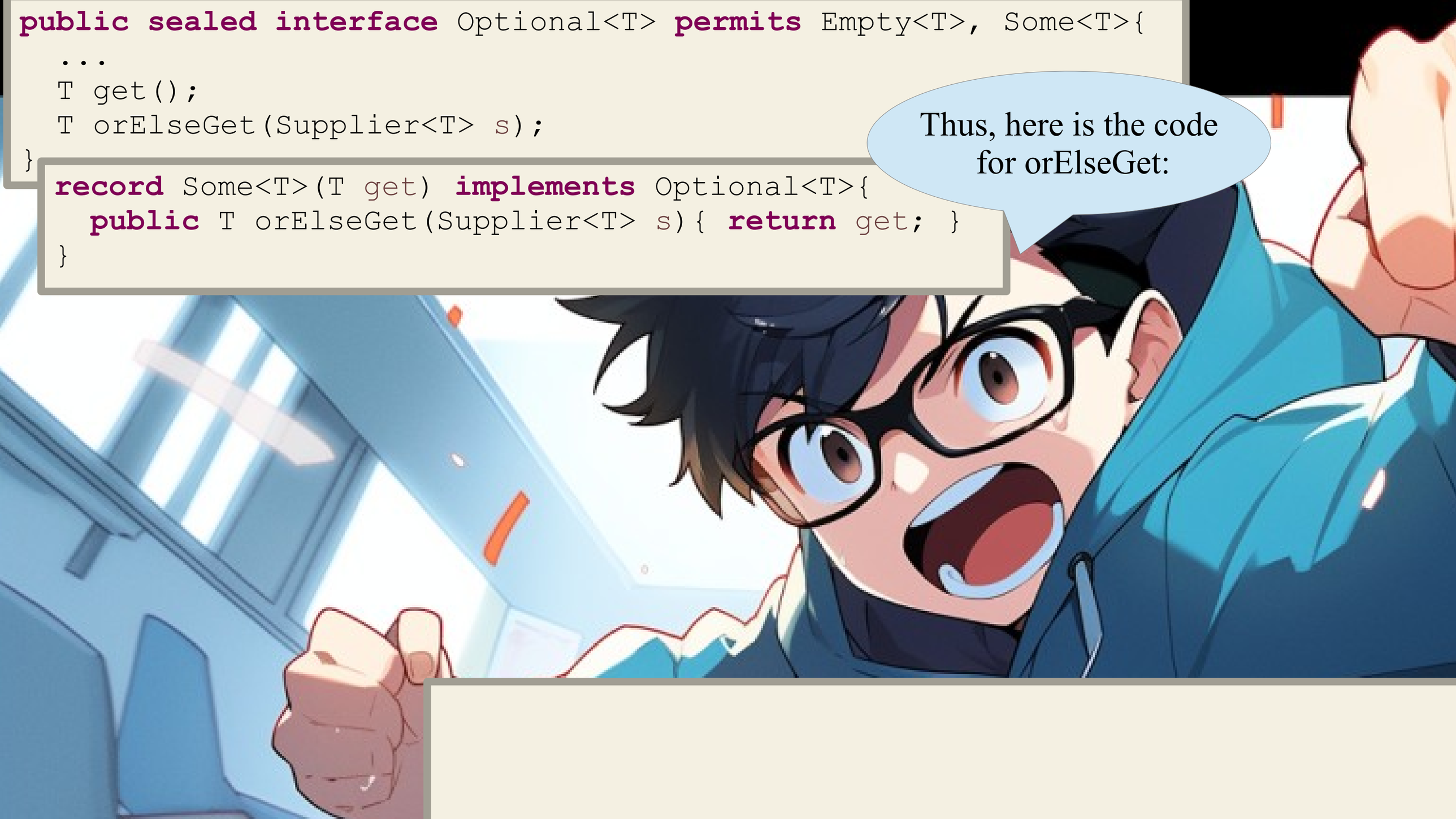
```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   ...
   T get();
   T orElseGet(Supplier<T> s);
}
```

```
record Some<T>(T get) implements Optional<T>{

}
```



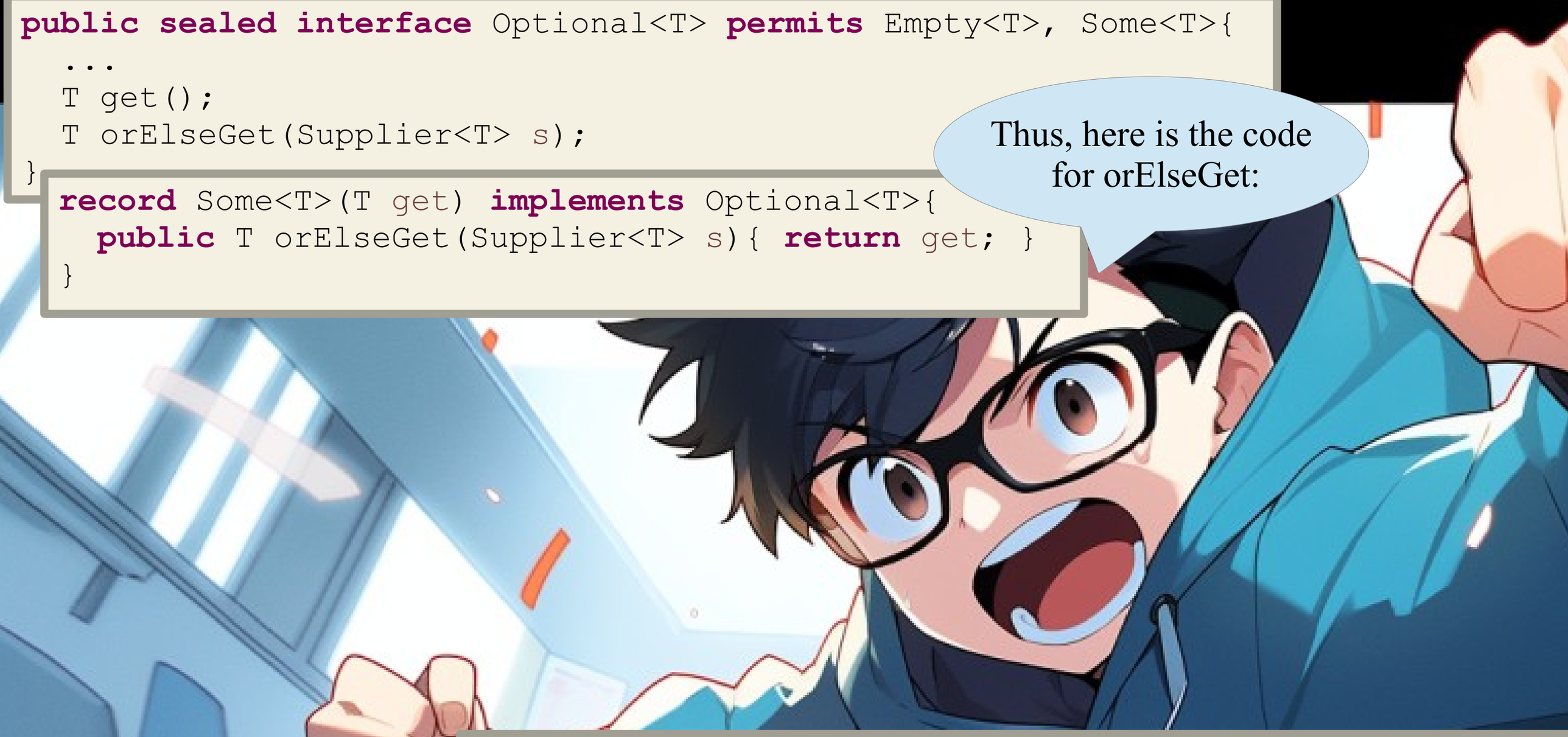
```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   ...
   T get();
   T orElseGet(Supplier<T> s);
}
```

```
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> s){ return get; }
}
```



```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   ...
   T get();
   T orElseGet(Supplier<T> s);
}
```

```
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> s){ return get; }
}
```




```
final class Empty<T> implements Optional<T>{


}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   ...
   T get();
   T orElseGet(Supplier<T> s);
}
```

```
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> s){ return get; }
}
```


```
final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElseGet(Supplier<T> s){ return s.get(); }
}
```

```java
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  ...
  T get();
  T orElseGet(Supplier<T> s);
}
```

```java
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> s){ return get; }
}
```

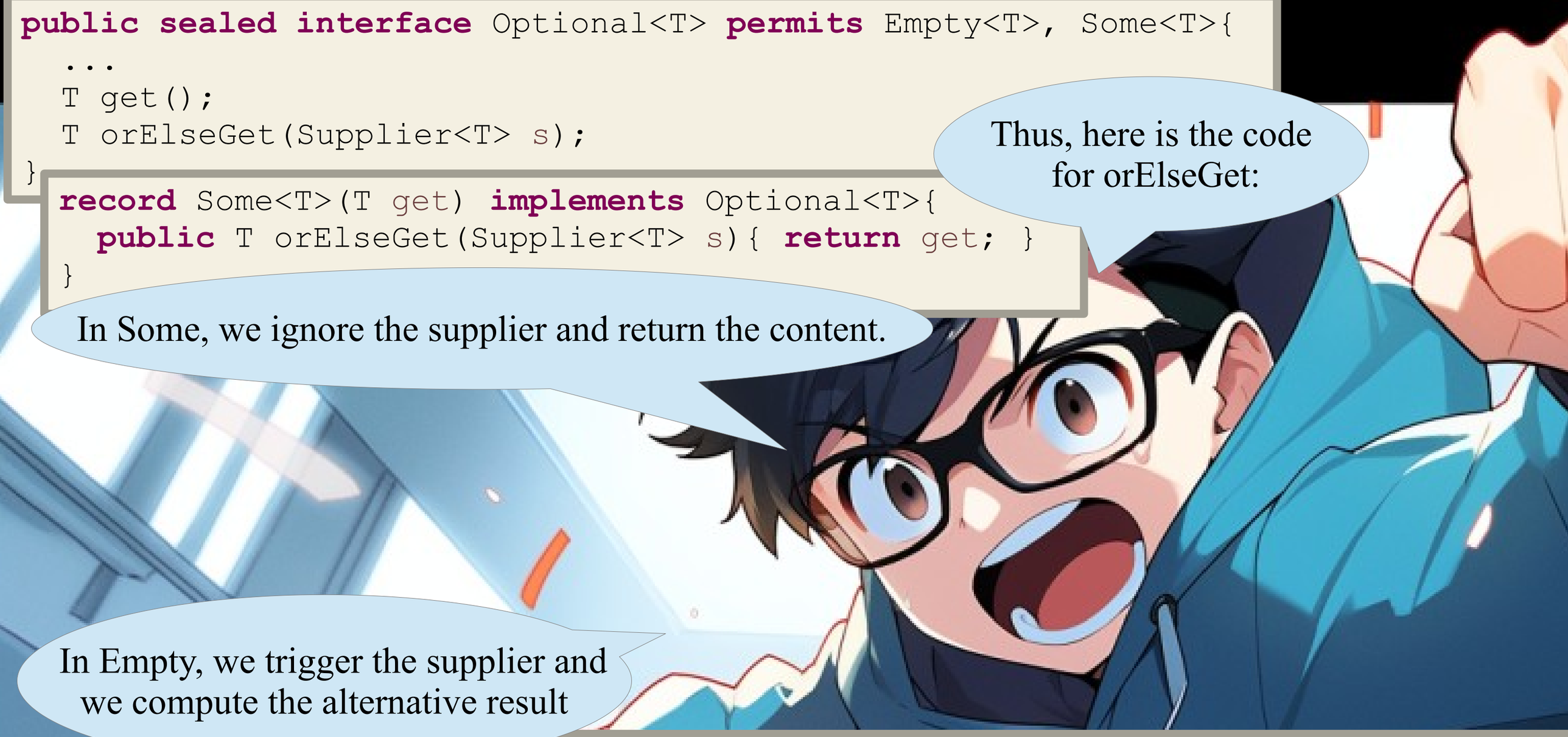
Thus, here is the code for orElseGet:

In Some, we ignore the supplier and return the content.

```java
final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElseGet(Supplier<T> s){ return s.get(); }
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  ...
  T get();
  T orElseGet(Supplier<T> s);
}
```

```
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> s){ return get; }
}
```

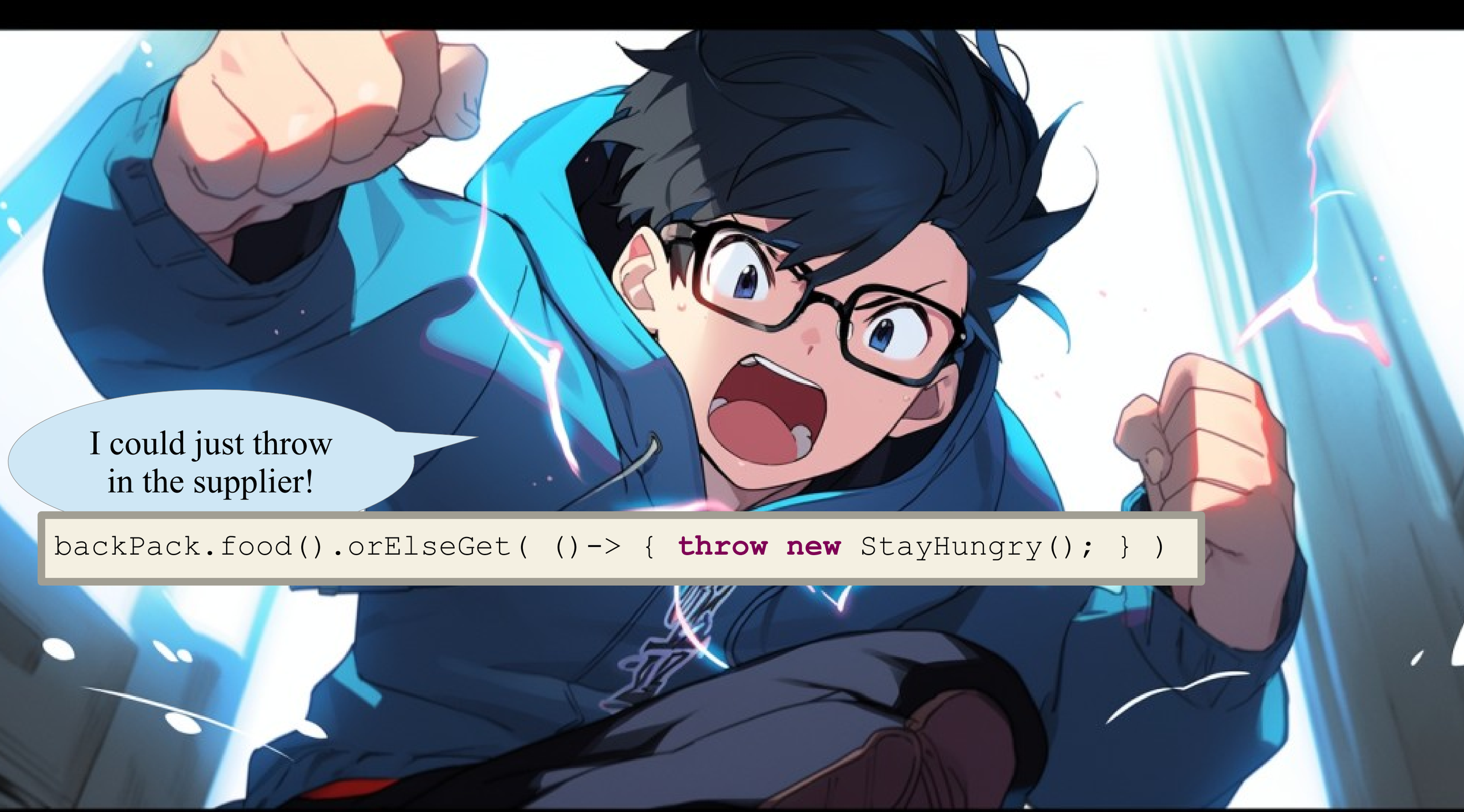


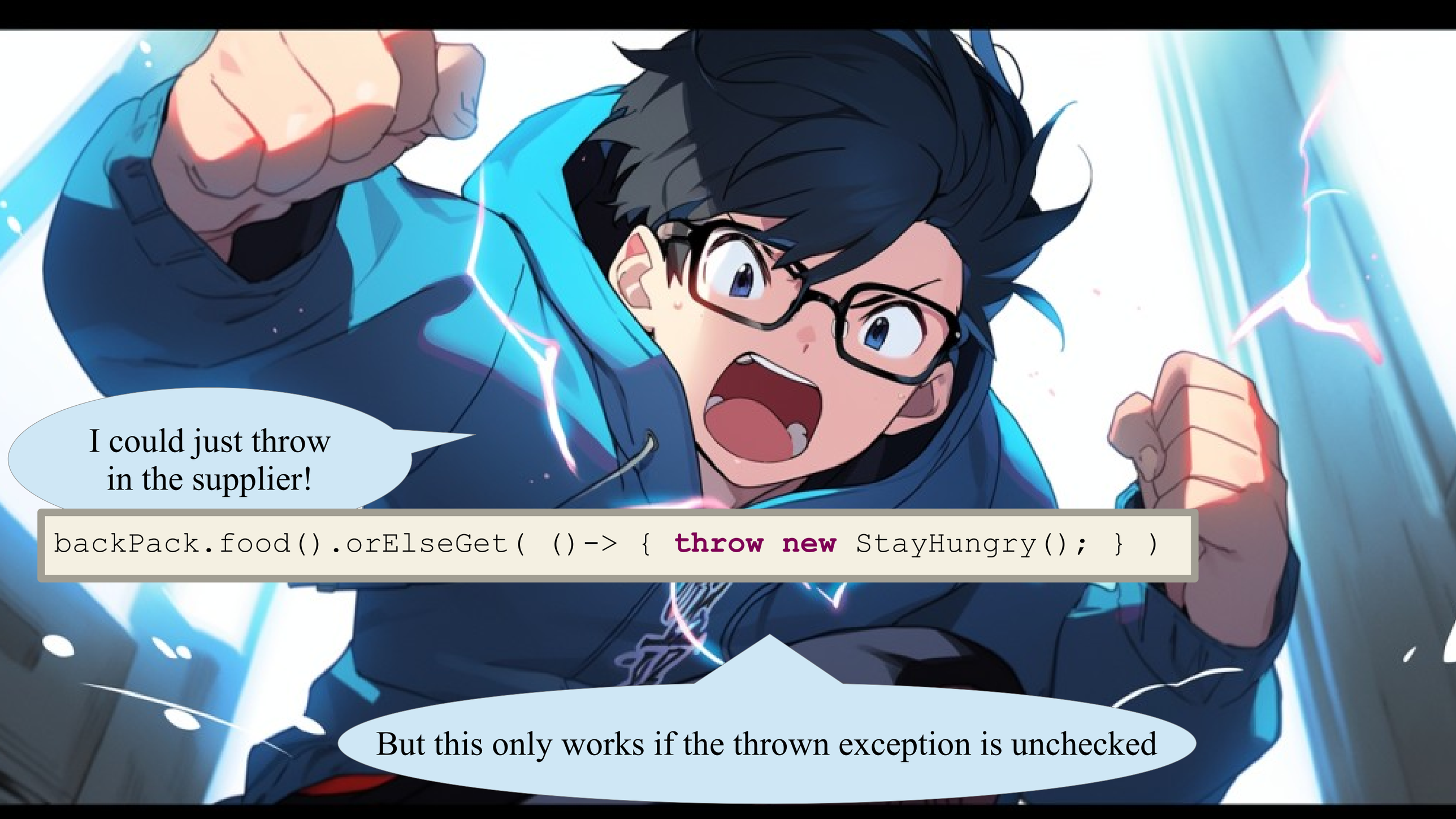

```
final class Empty<T> implements Optional<T>{
  public T get(){ throw new NoSuchElementException(); }
  public T orElseGet(Supplier<T> s){ return s.get(); }
}
```

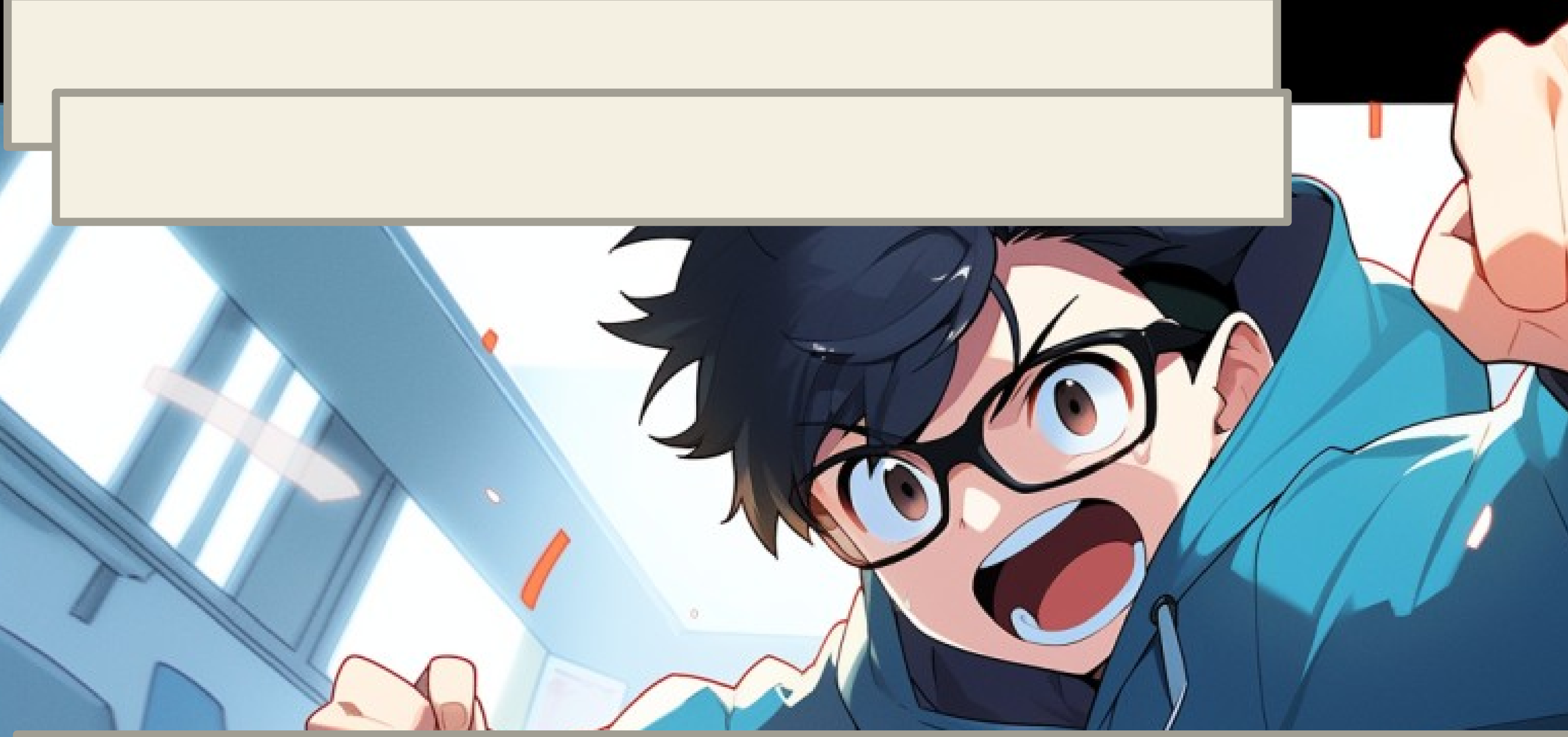

Now,
with the last breath
of power left


Now,
with the last breath
of power left
Can we
derive orElseThrow
from orElseGet?


I could just throw in the supplier!

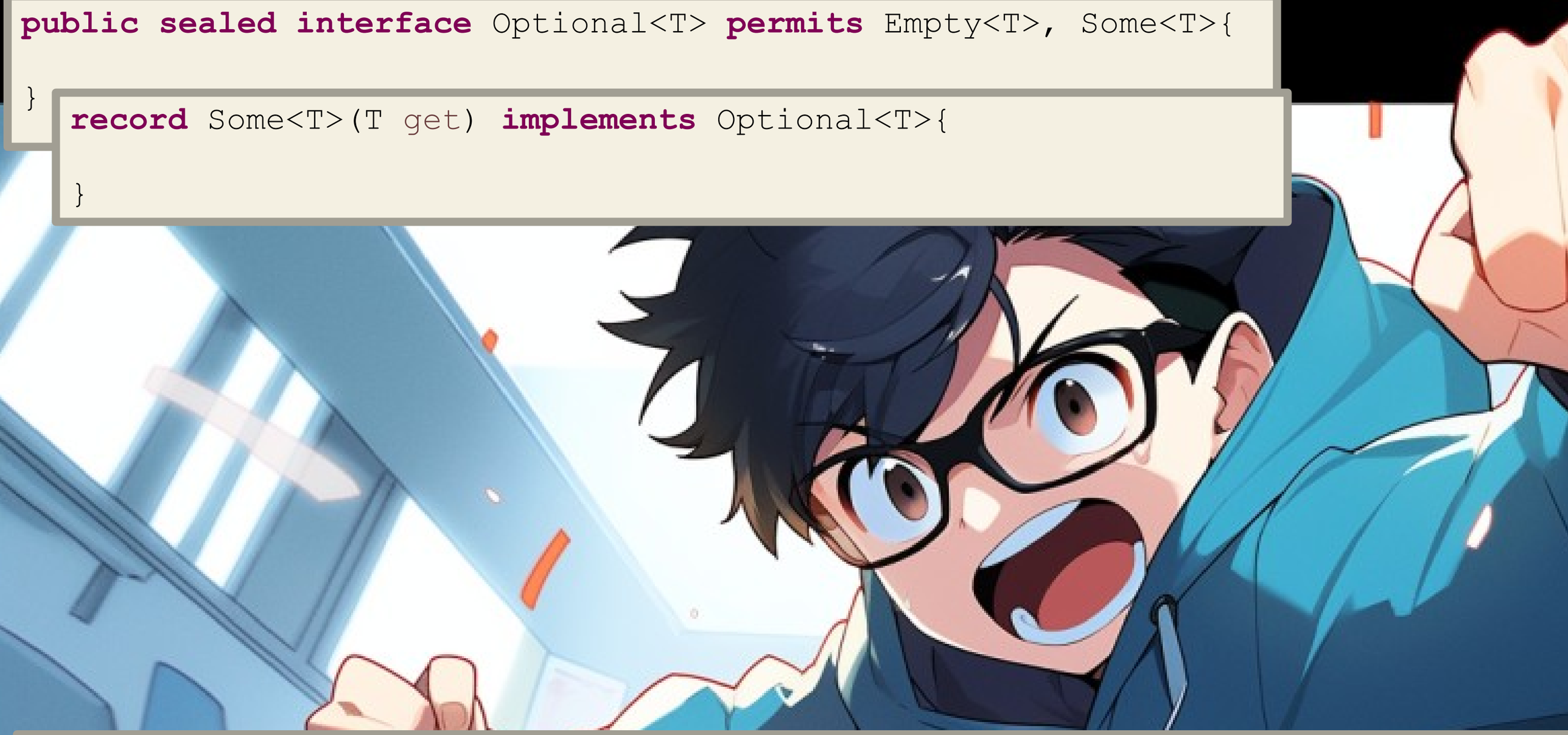

I could just throw in the supplier!
backPack.food().orElseGet(()-> { throw new StayHungry(); })


I could just throw in the supplier!
backPack.food().orElseGet(()-> { throw new StayHungry(); })
But this only works if the thrown exception is unchecked



```java
public sealed interface Optional<T> permits Empty<T>, Some<T>{

}
```

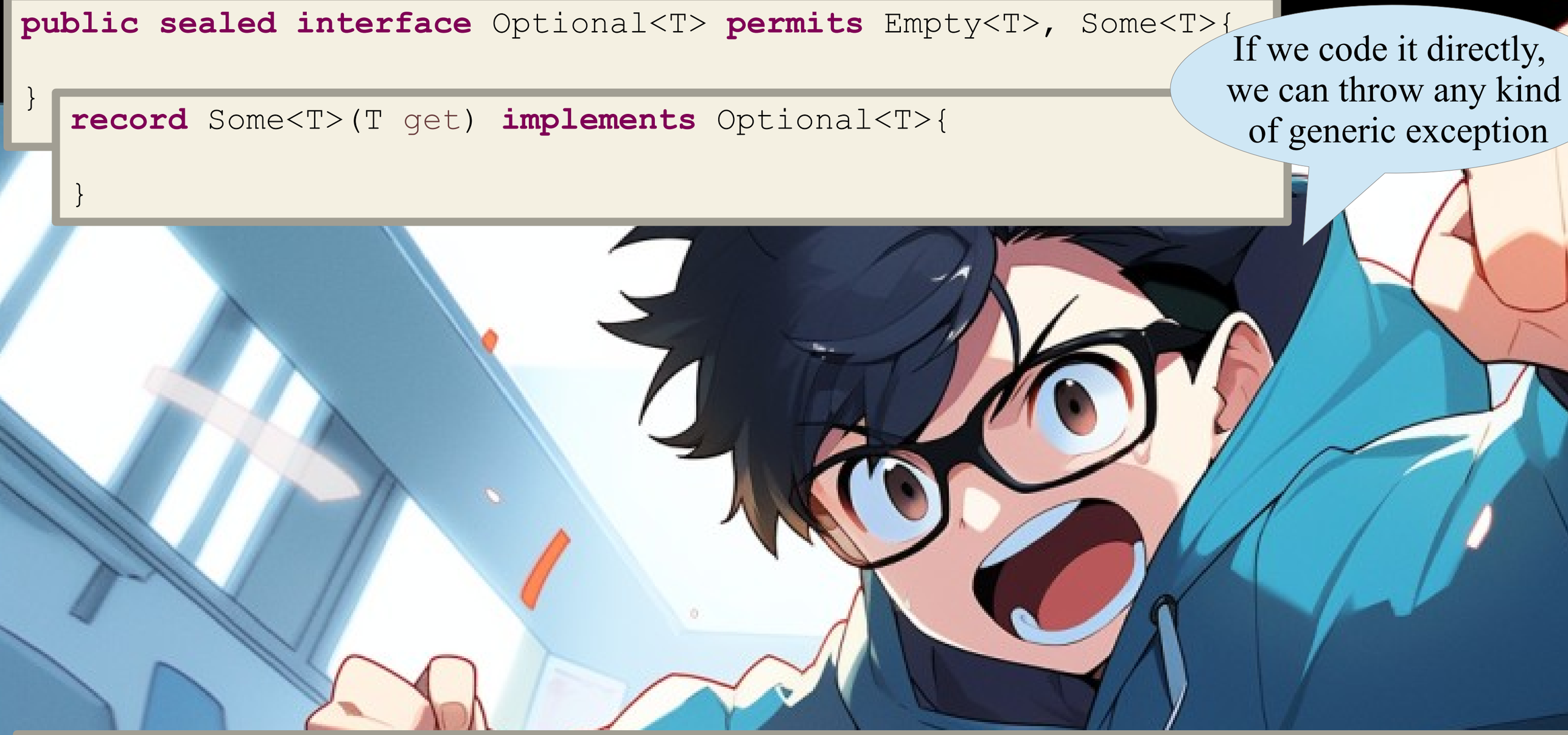
```java
record Some<T>(T get) implements Optional<T>{

}
```



```java
final class Empty<T> implements Optional<T>{

}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{

}
```

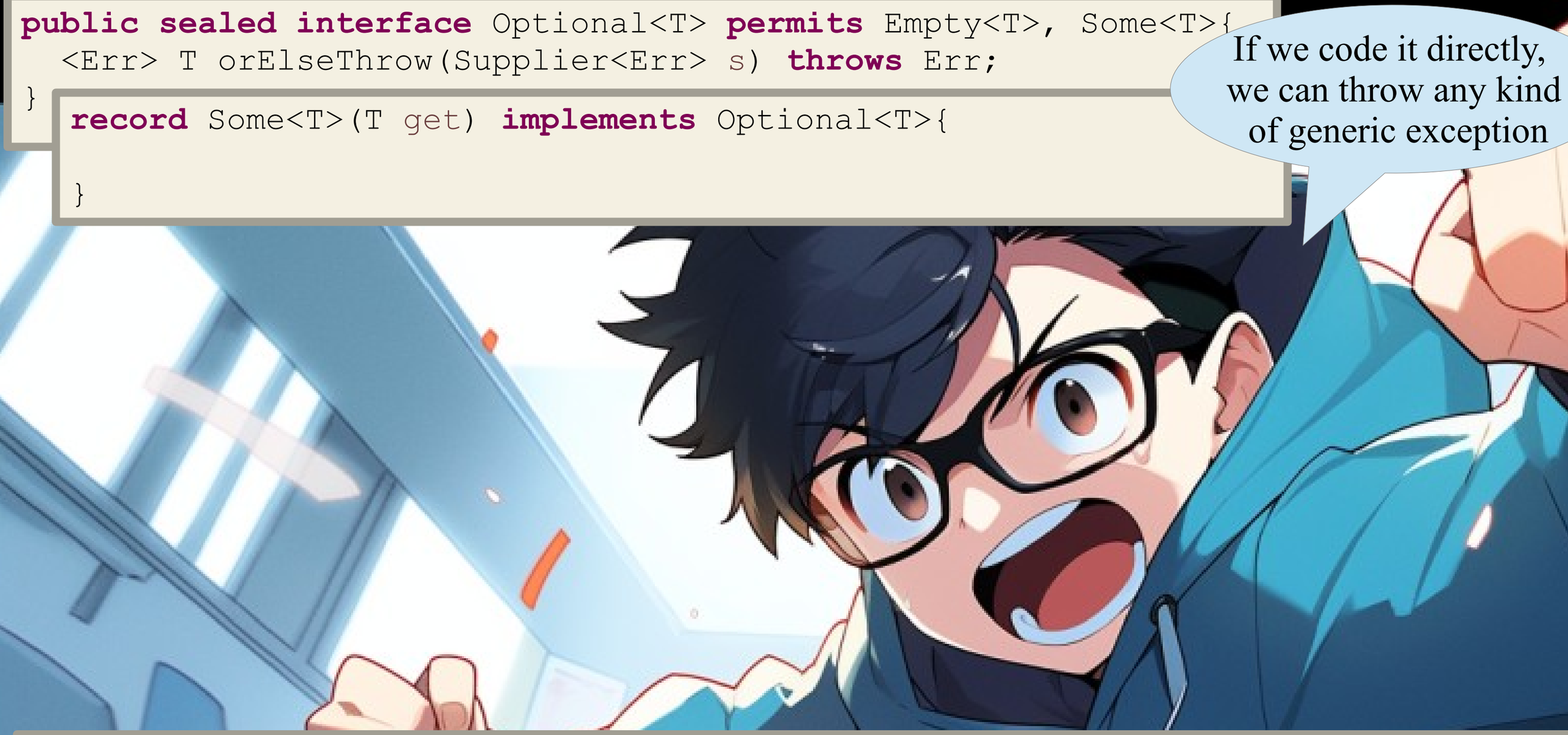
```
record Some<T>(T get) implements Optional<T>{

}
```




```
final class Empty<T> implements Optional<T>{

}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
   <Err> T orElseThrow(Supplier<Err> s) throws Err;
}
```

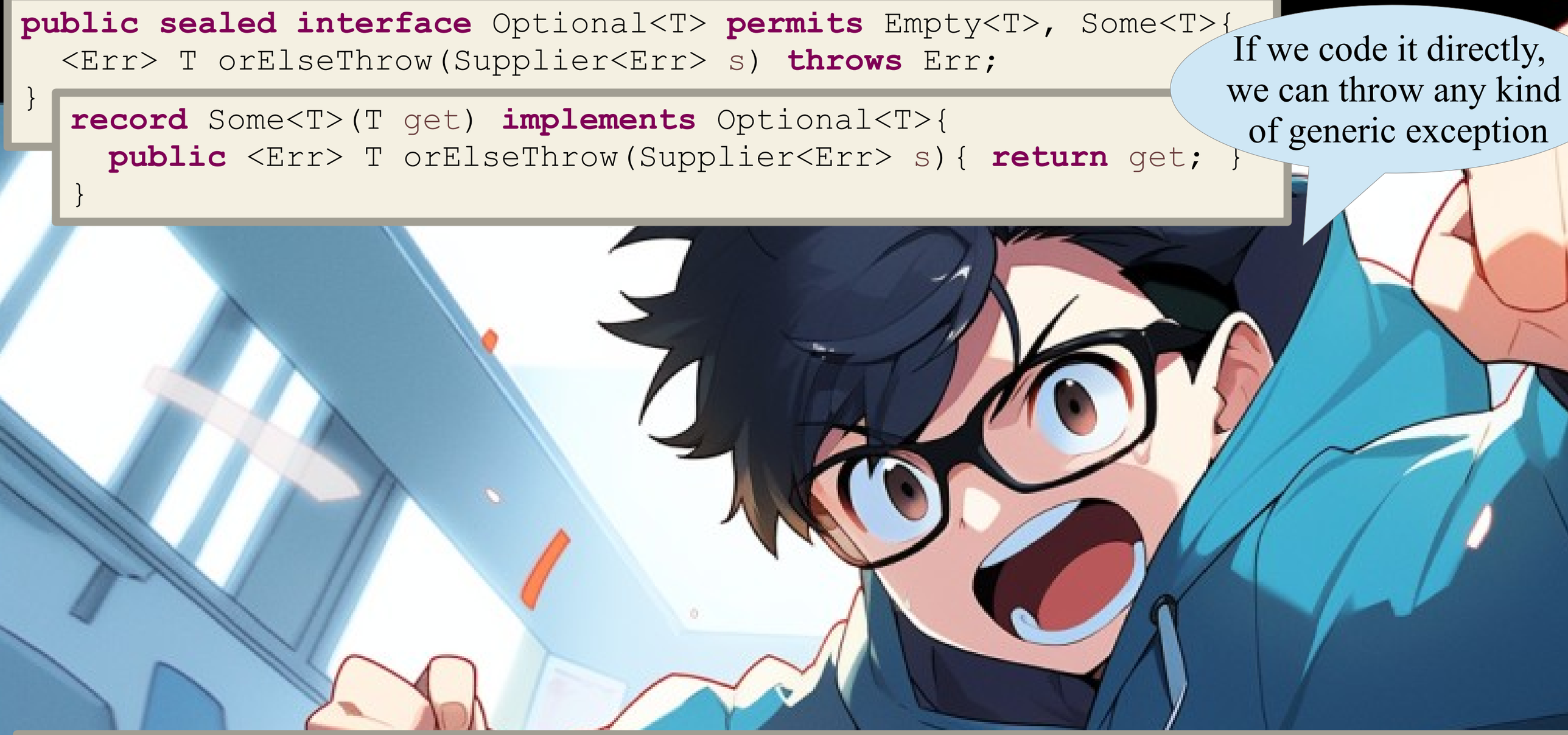
```
record Some<T>(T get) implements Optional<T>{

}
```




```
final class Empty<T> implements Optional<T>{

}
```

```java
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  <Err> T orElseThrow(Supplier<Err> s) throws Err;
}
```

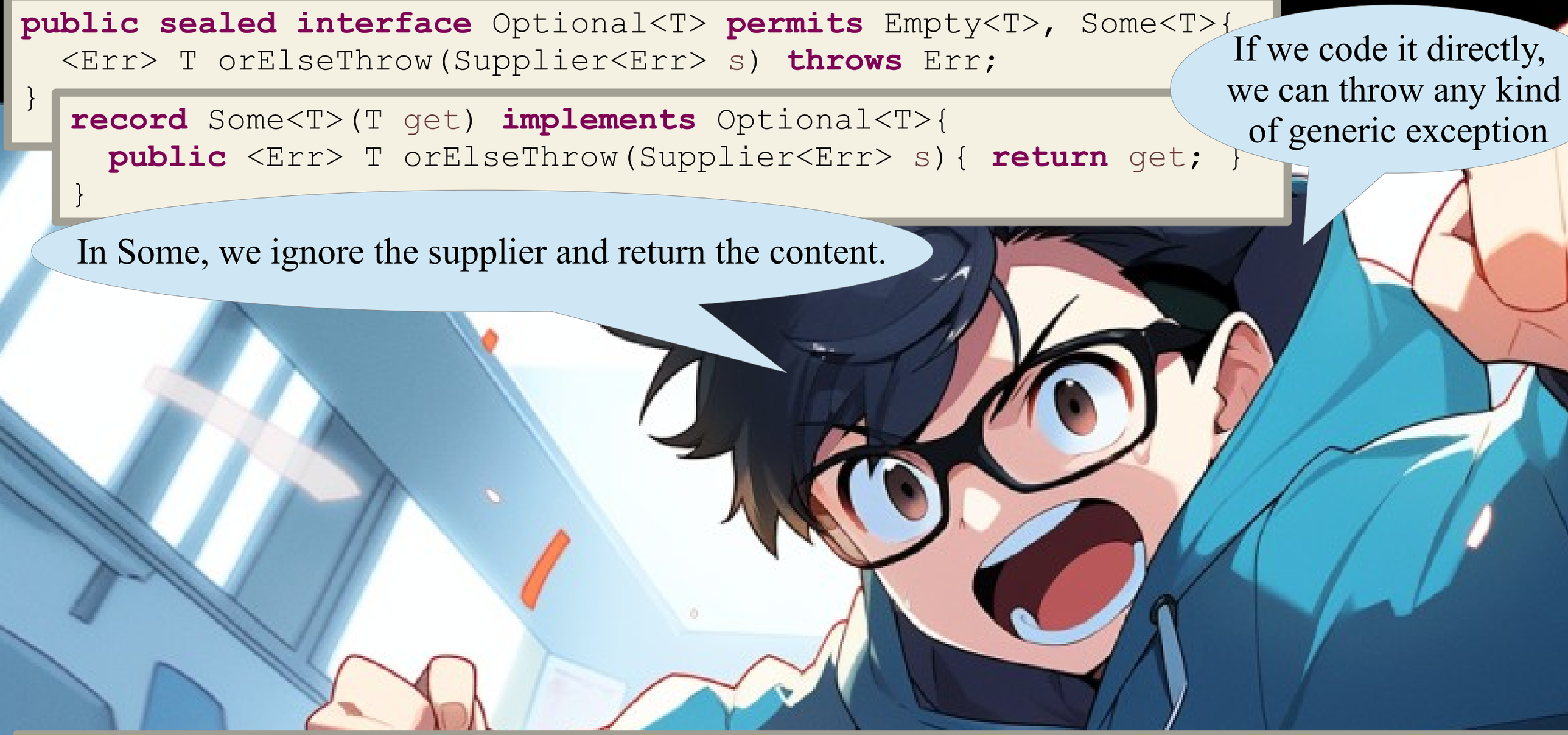
```java
record Some<T>(T get) implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s){ return get; }
}
```




```java
final class Empty<T> implements Optional<T>{

}
```

```java
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  <Err> T orElseThrow(Supplier<Err> s) throws Err;
}
```

```java
record Some<T>(T get) implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s){ return get; }
}
```



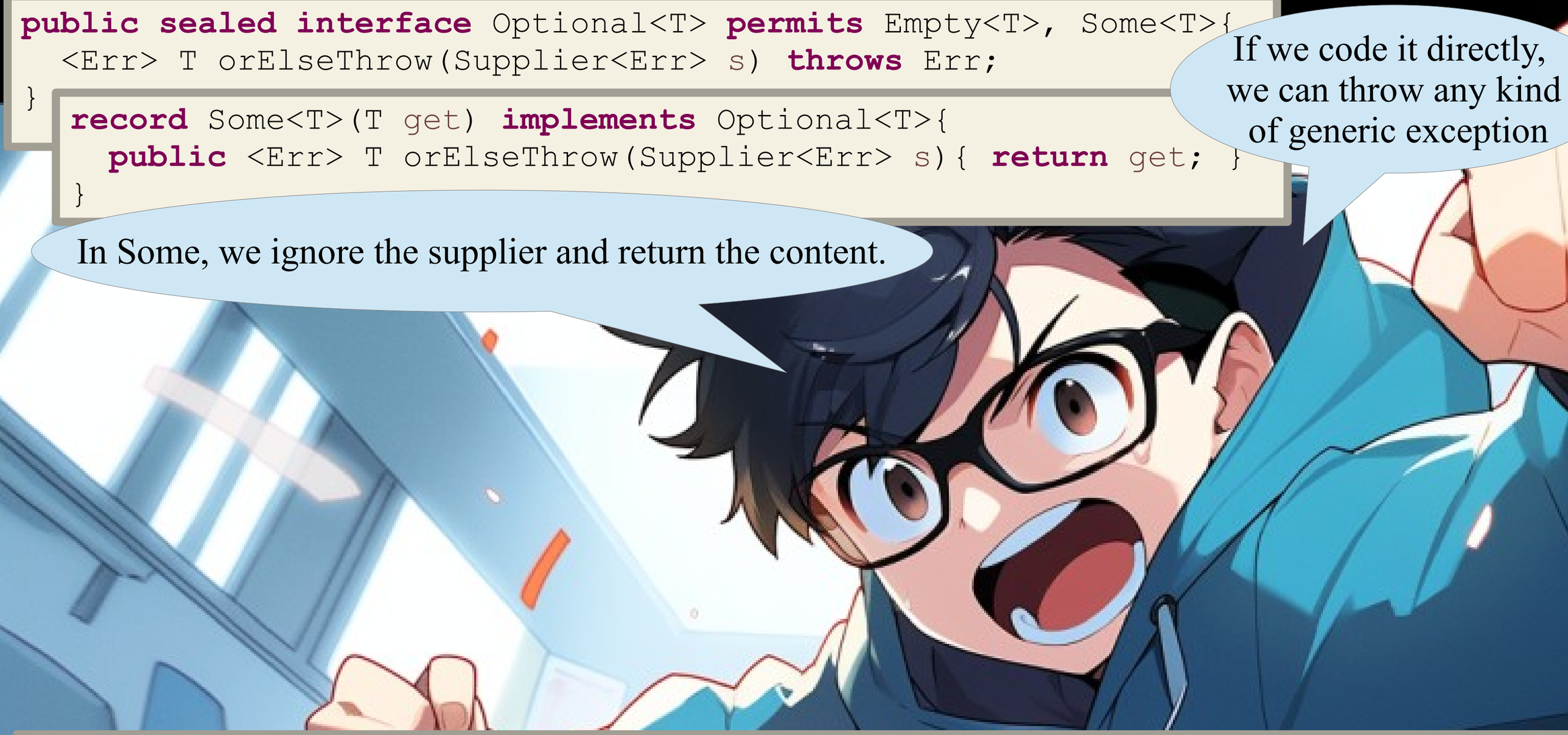

```java
final class Empty<T> implements Optional<T>{

}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  <Err> T orElseThrow(Supplier<Err> s) throws Err;
}
```

```
record Some<T>(T get) implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s){ return get; }
}
```

If we code it directly, we can throw any kind of generic exception

In Some, we ignore the supplier and return the content.



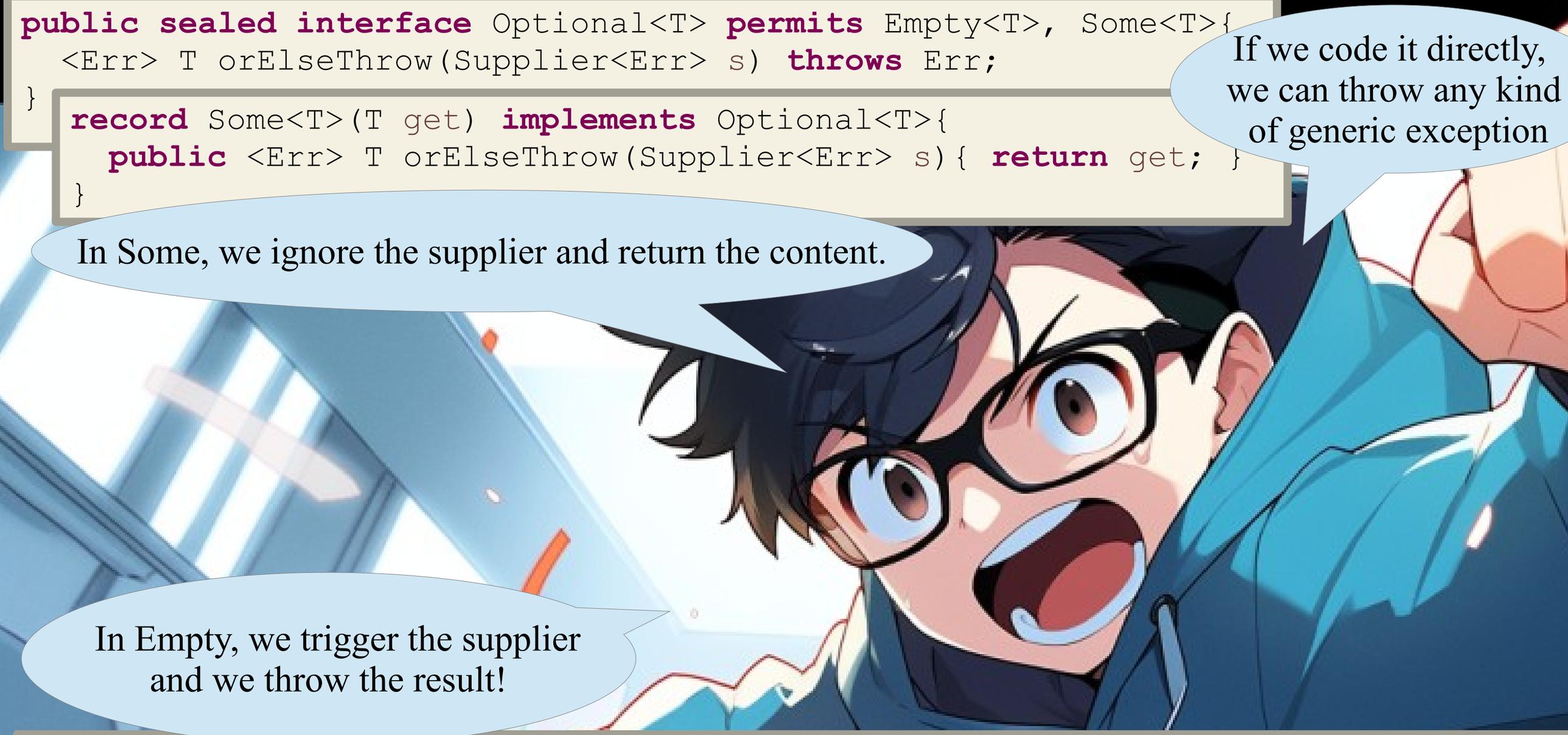
```
final class Empty<T> implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s) throws Err { throw s.get(); }
}
```

```
public sealed interface Optional<T> permits Empty<T>, Some<T>{
  <Err> T orElseThrow(Supplier<Err> s) throws Err;
}
```

```
record Some<T>(T get) implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s){ return get; }
}
```

If we code it directly, we can throw any kind of generic exception

In Some, we ignore the supplier and return the content.



In Empty, we trigger the supplier and we throw the result!

```
final class Empty<T> implements Optional<T>{
  public <Err> T orElseThrow(Supplier<Err> s) throws Err { throw s.get(); }
}
```