



tooCold




or



tooHot

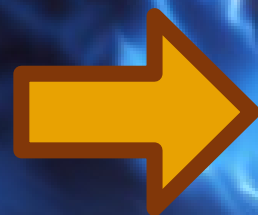


sadClimate



The result is true
if either is true

tooCold



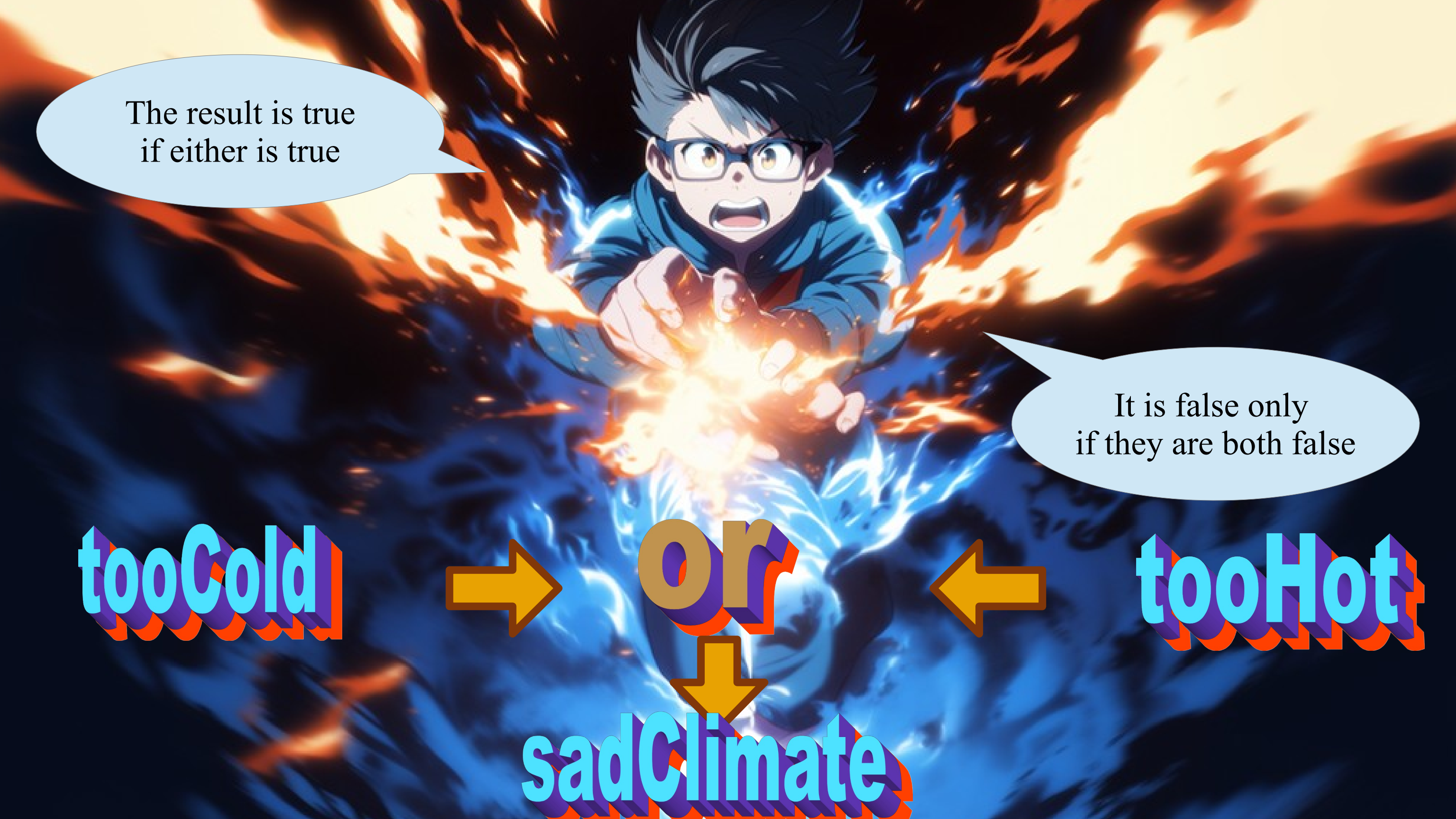
or



tooHot



sadClimate



The result is true
if either is true

It is false only
if they are both false

tooCold



or



tooHot

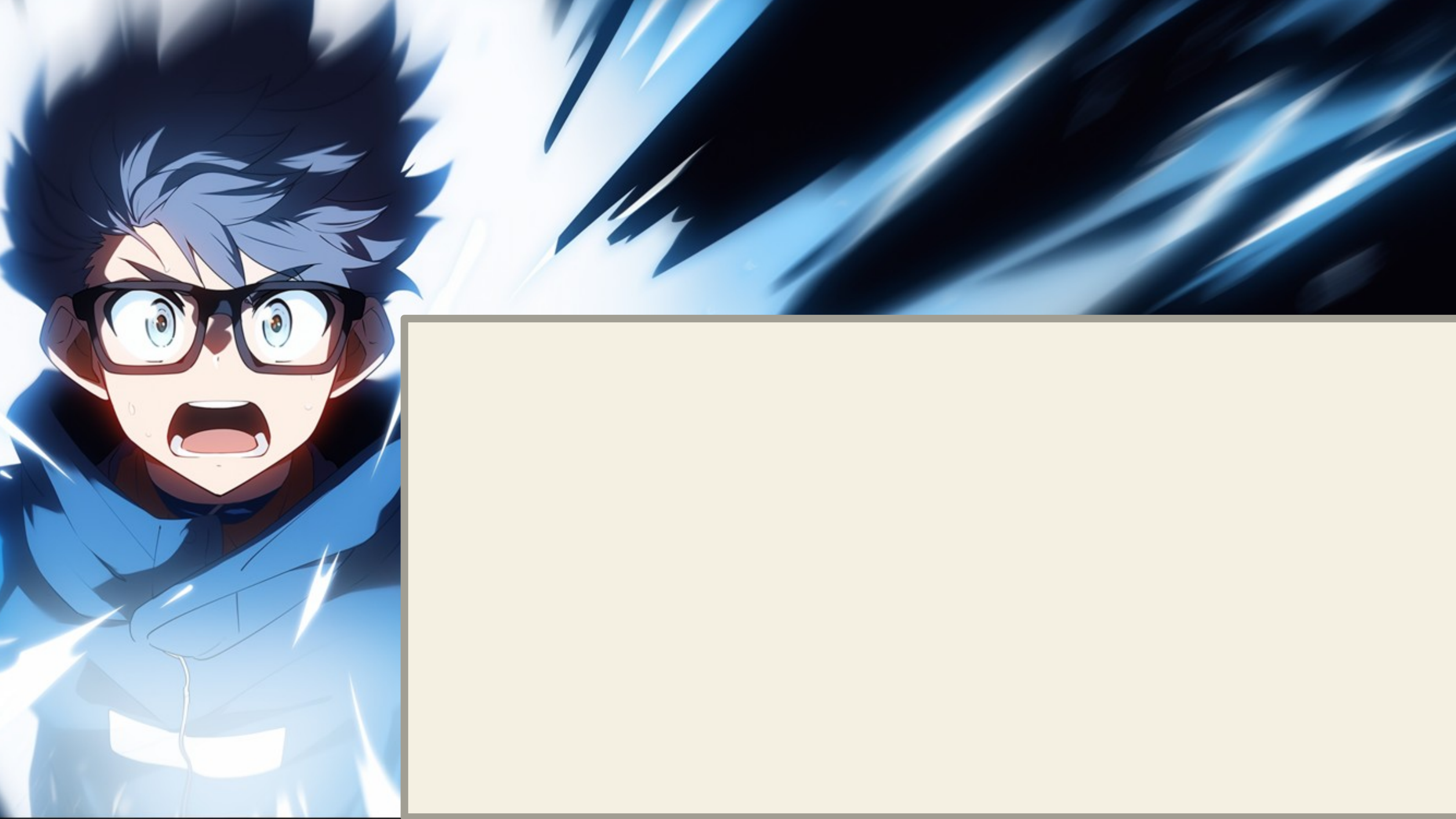


sadClimate




Would this work
also for Optionals?







```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```



The boolean 'or'
goes from boolean and boolean
into boolean


```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```




```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```



```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```

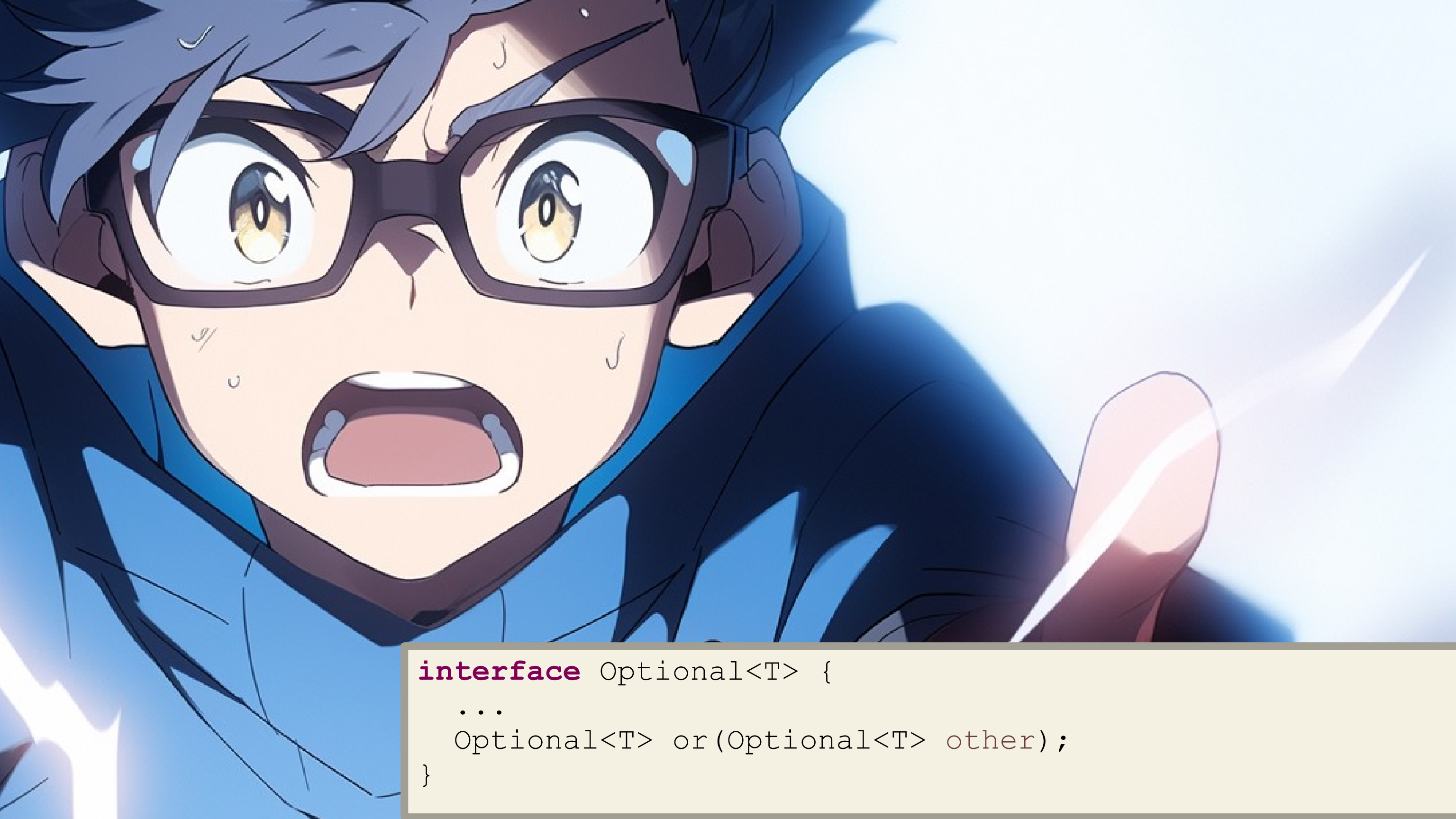



The Optional 'or' would go from
Optional and Optional into Optional


```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```





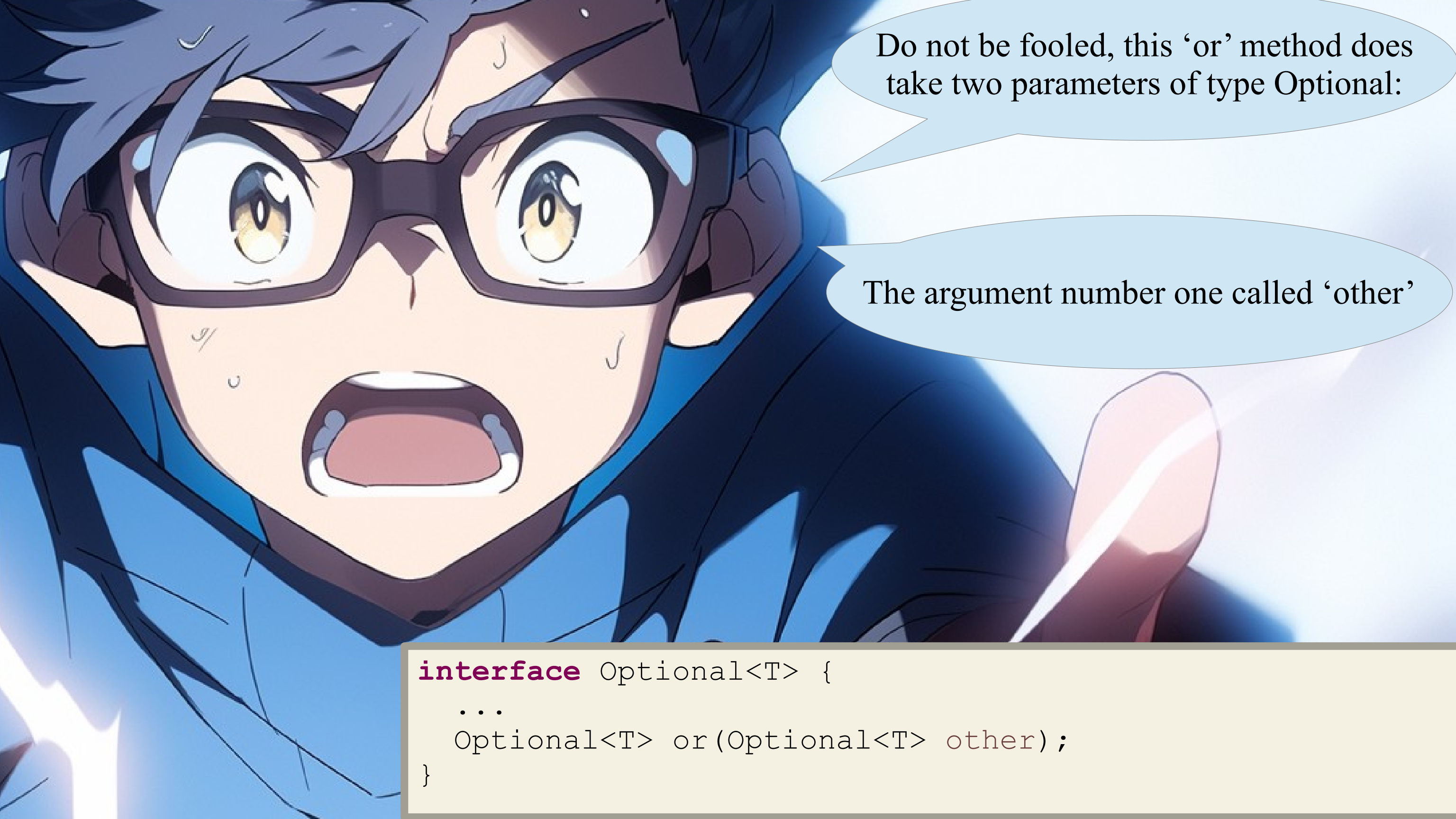


```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```

Do not be fooled, this 'or' method does take two parameters of type Optional:

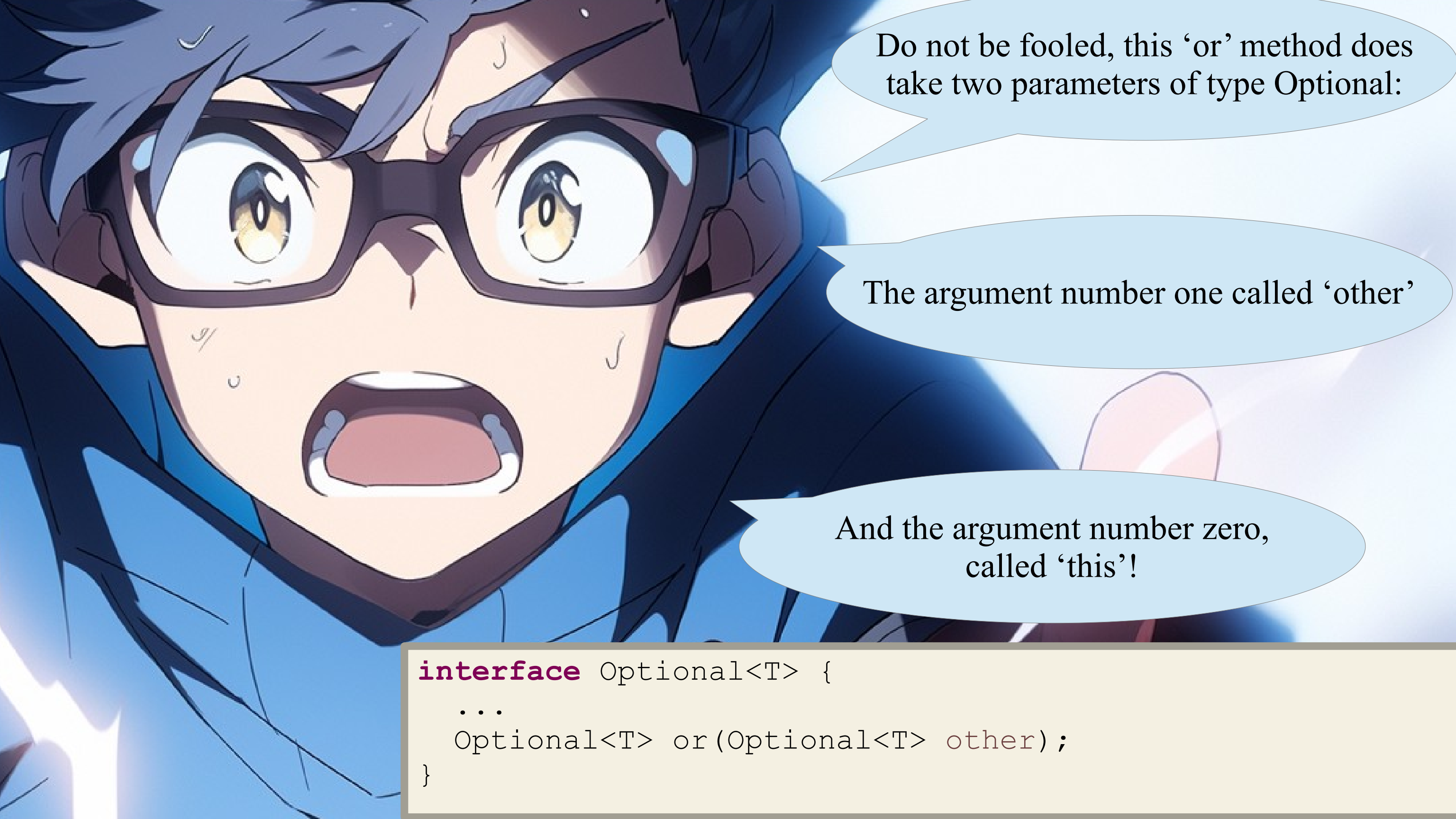
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```



Do not be fooled, this 'or' method does take two parameters of type Optional:

The argument number one called 'other'

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```

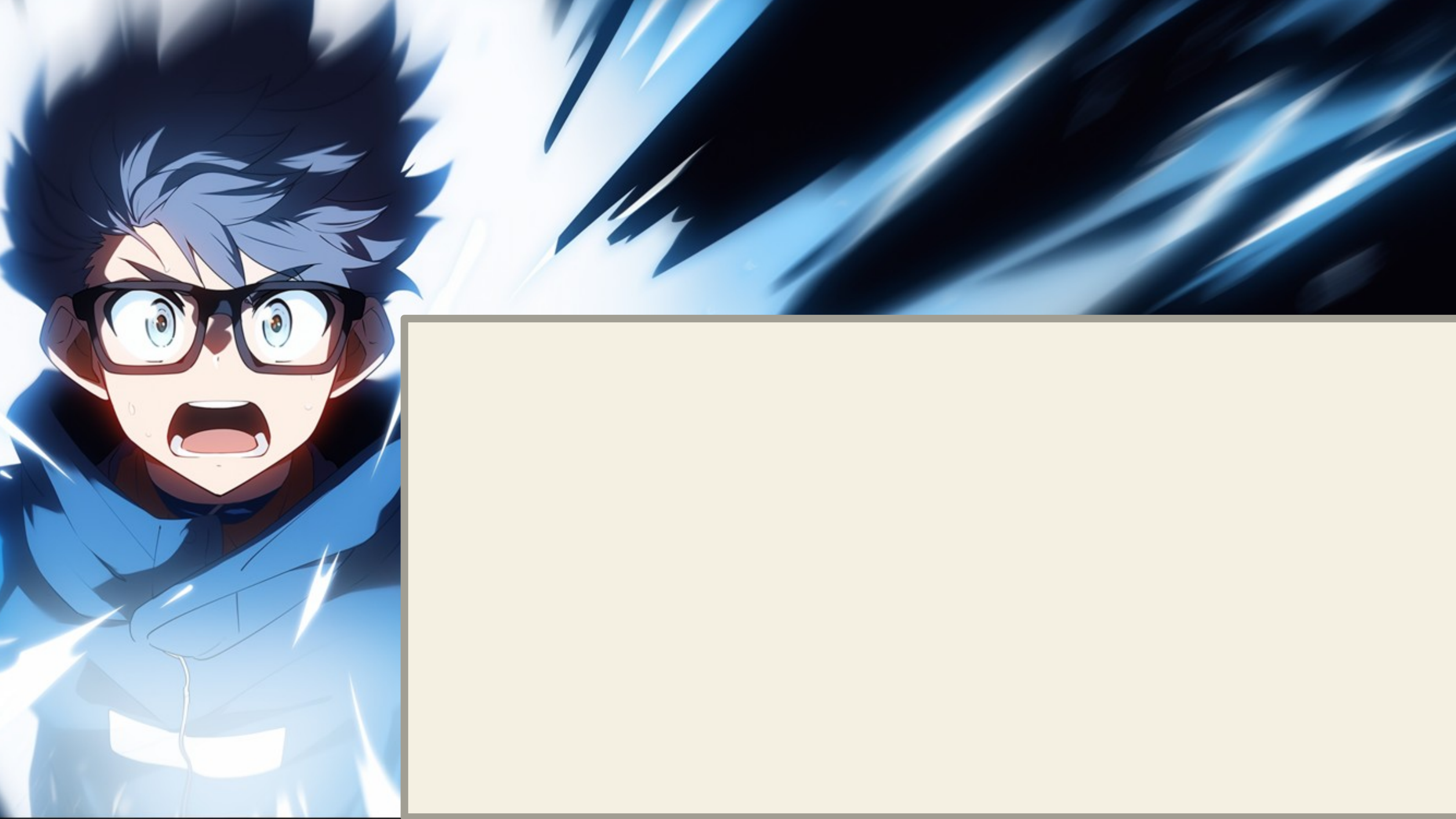



Do not be fooled, this 'or' method does take two parameters of type Optional:

The argument number one called 'other'

And the argument number zero, called 'this'!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```





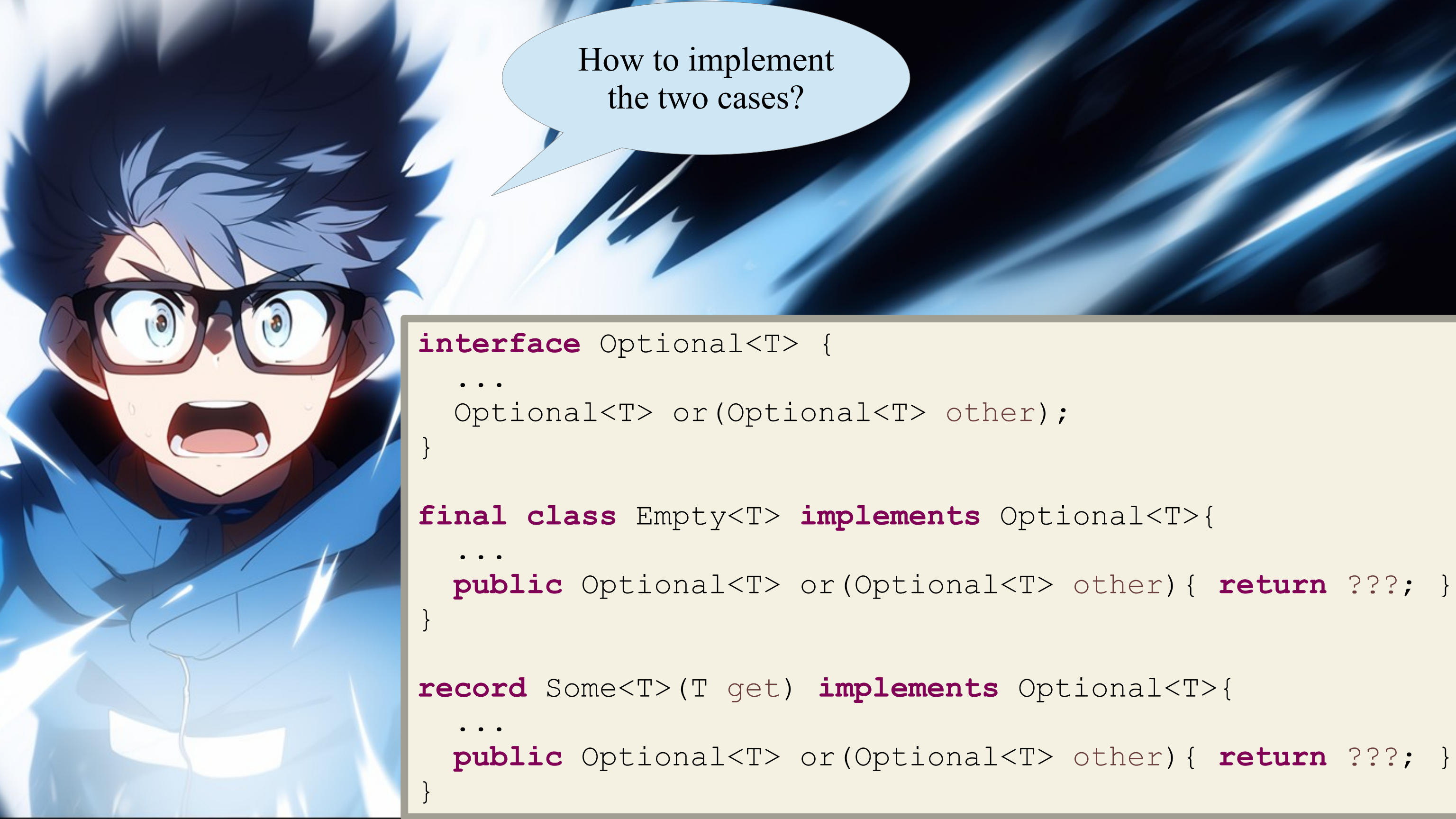
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```



```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return ???; }  
}
```




```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```



How to implement
the two cases?

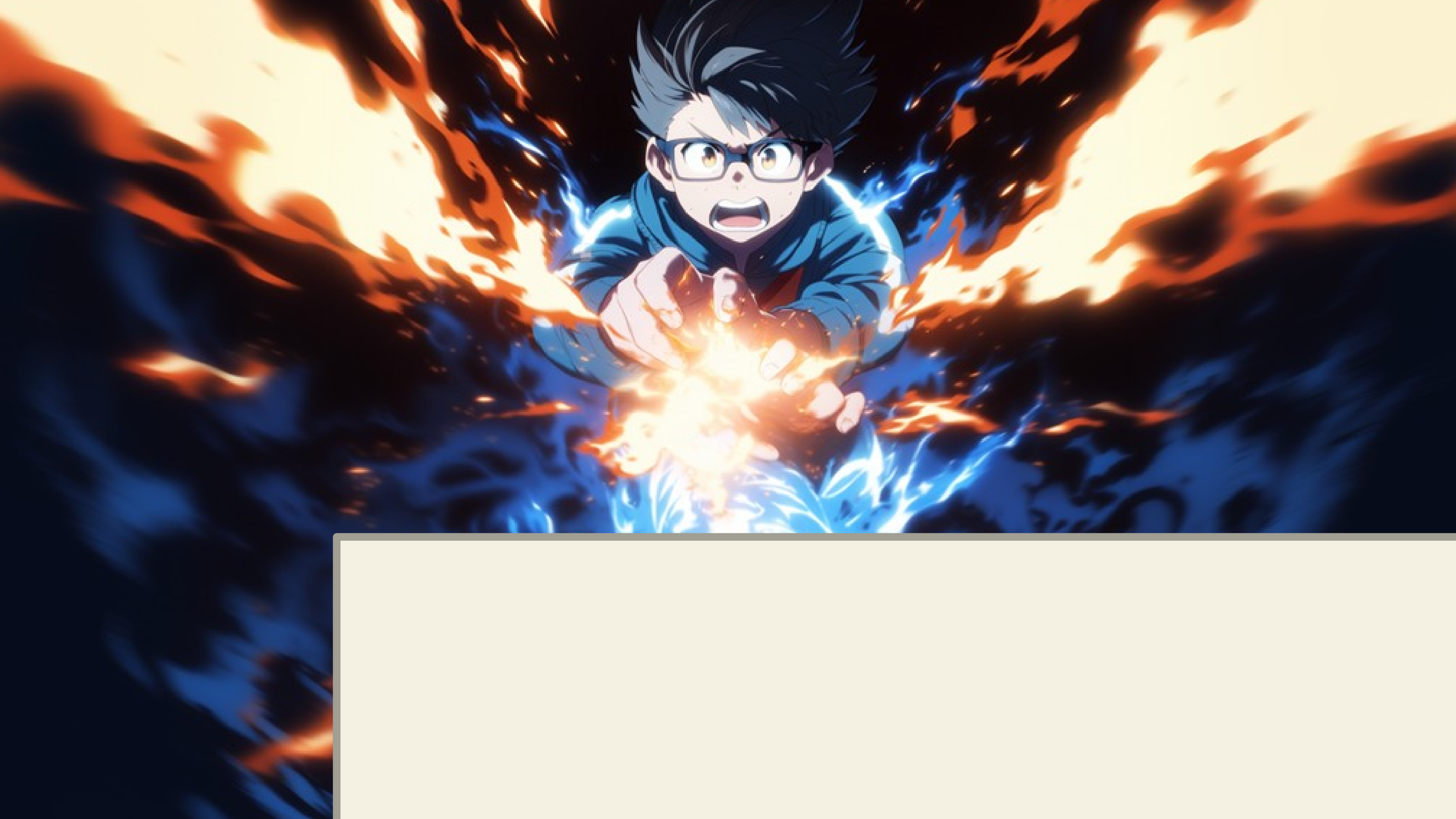
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```




How to implement
the two cases?

The empty case is probably the ‘false’ case,
since it smells of NOPE all around!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return ???; }  
}
```






```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}
```




```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```

“false.or(other)” for booleans

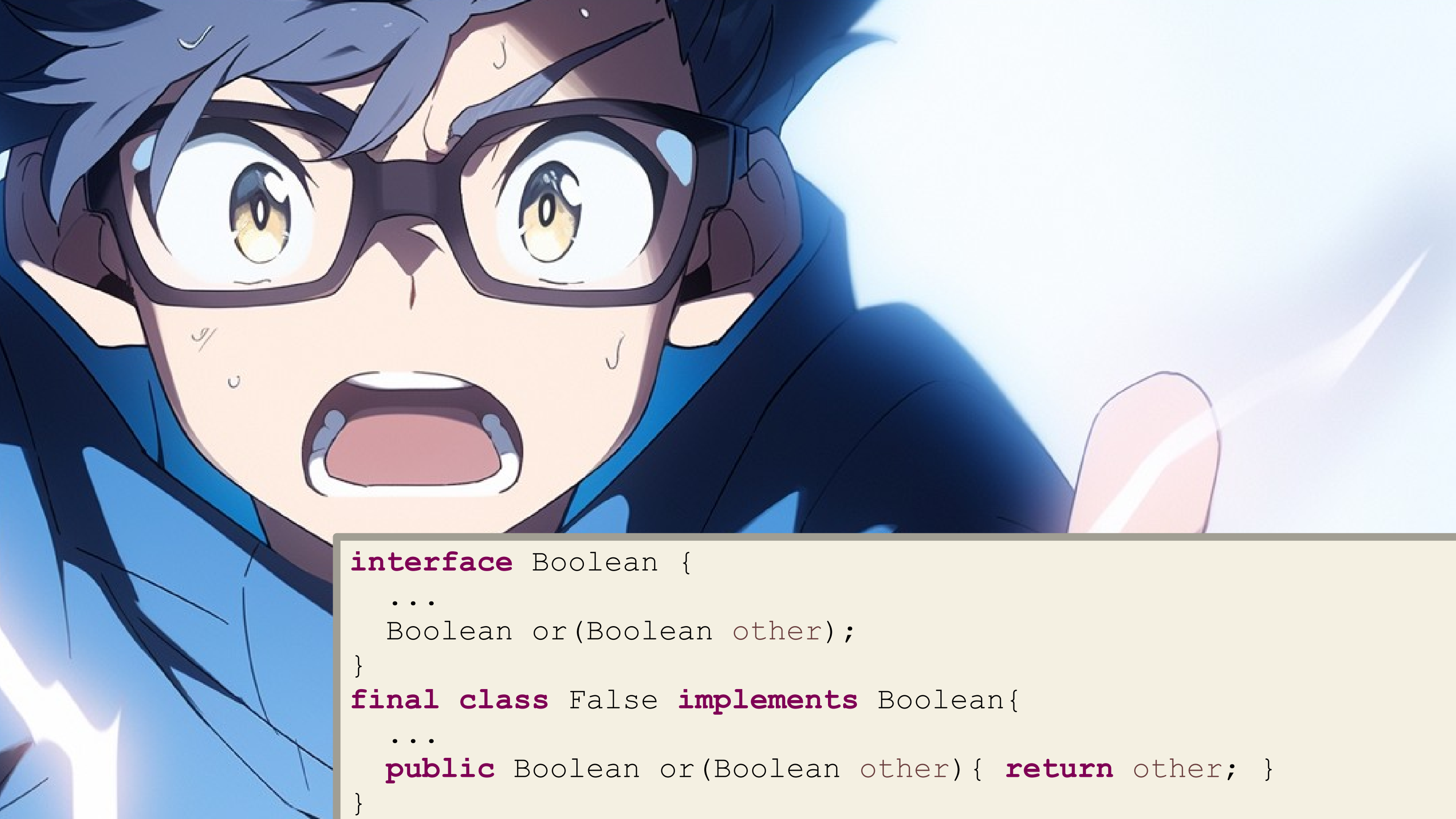
```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```




“false.or(other)” for booleans

Would look like this

```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```

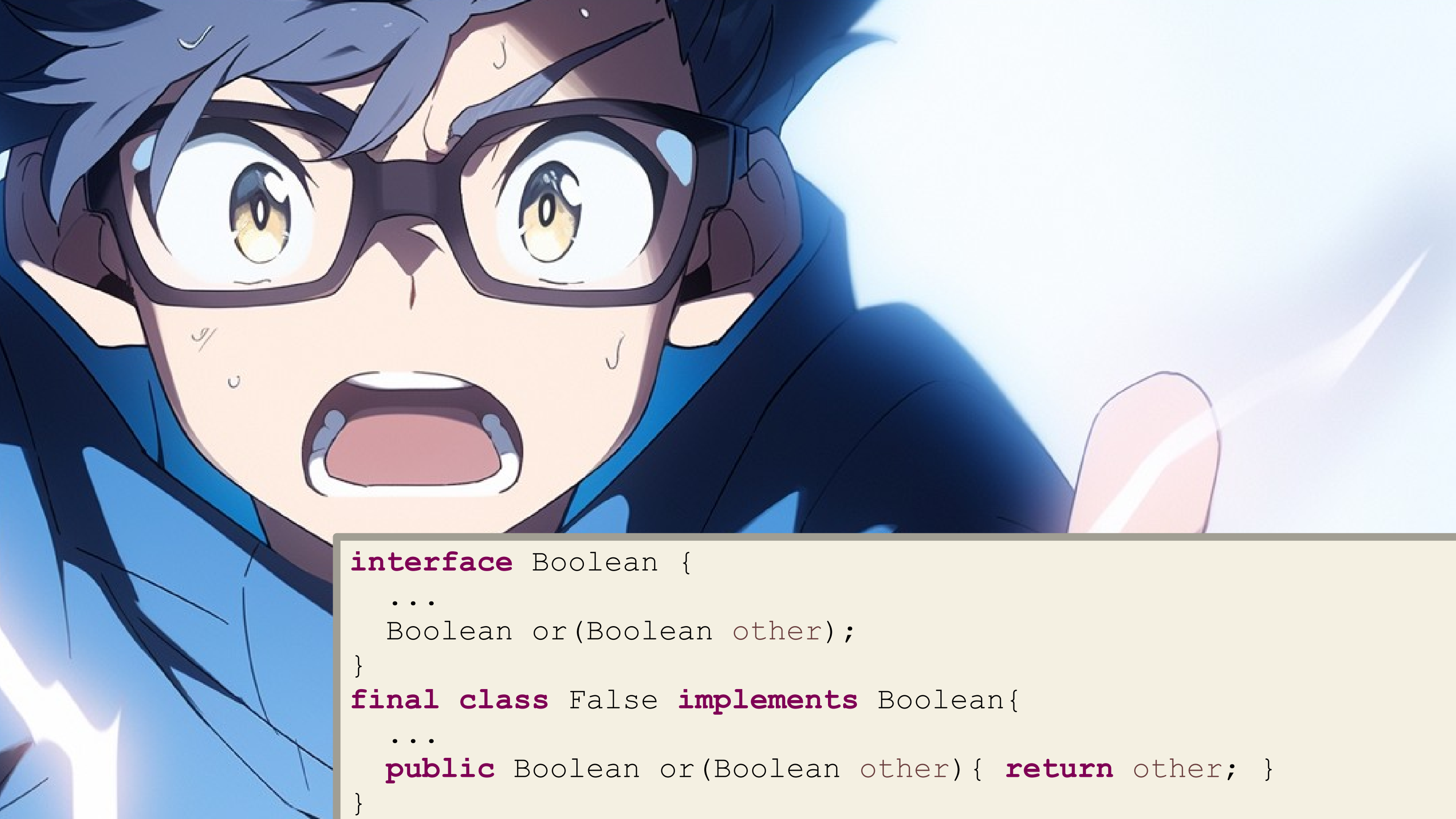



```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```

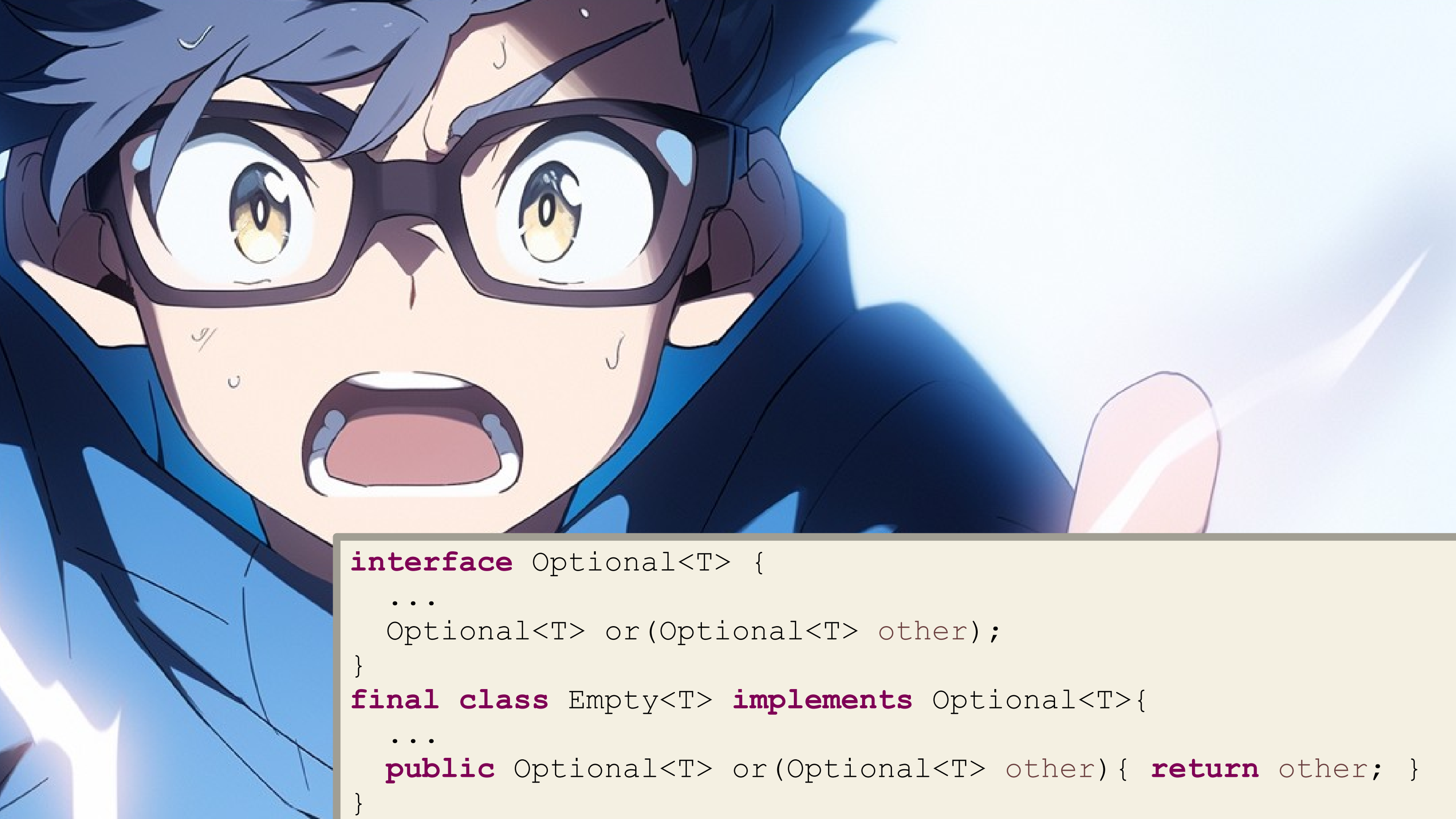


We are 'False', so we are not 'True'.
There is a chance that 'other' is 'True'.
So we return it.

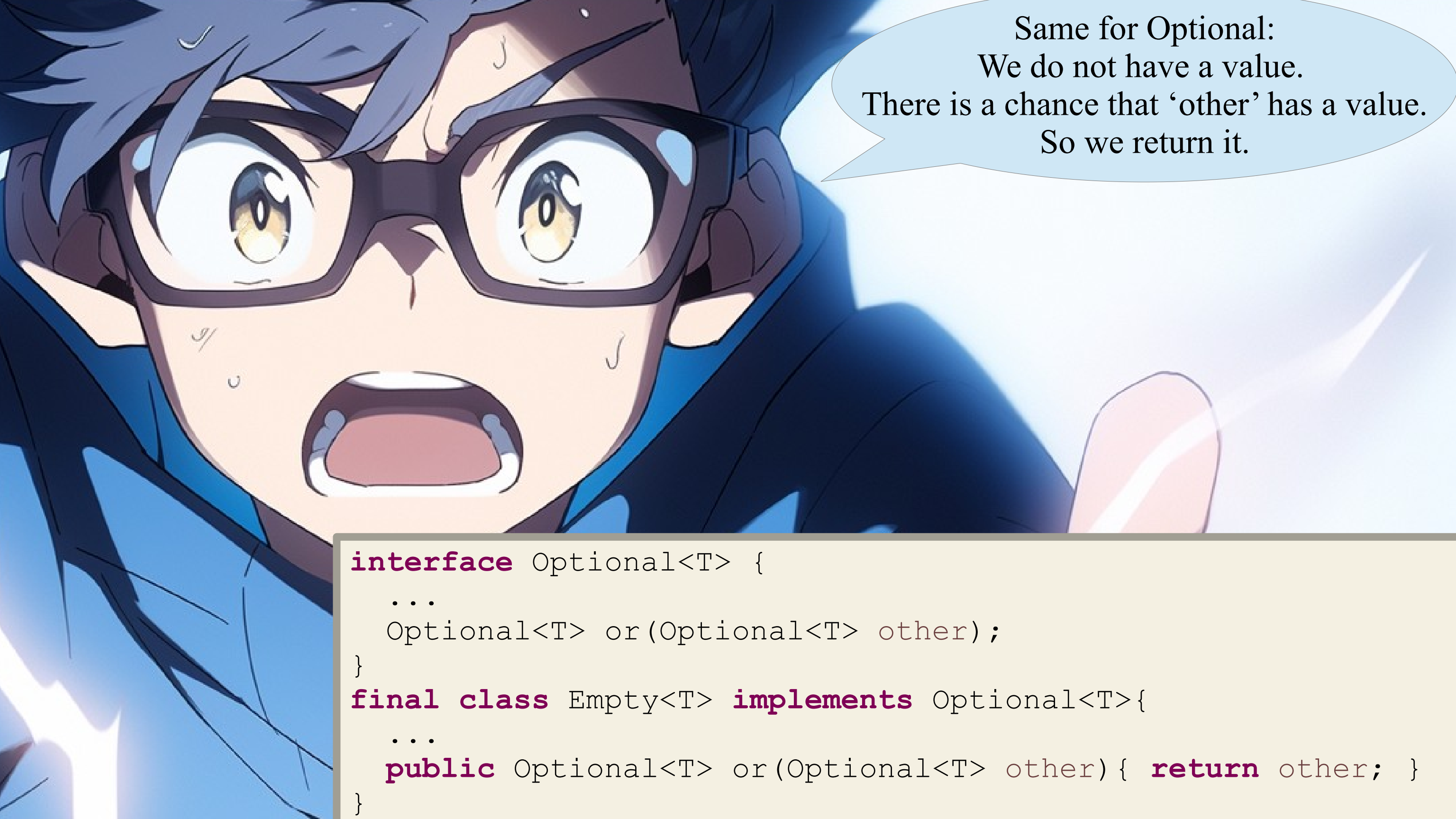
```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```

```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class False implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return other; }  
}
```



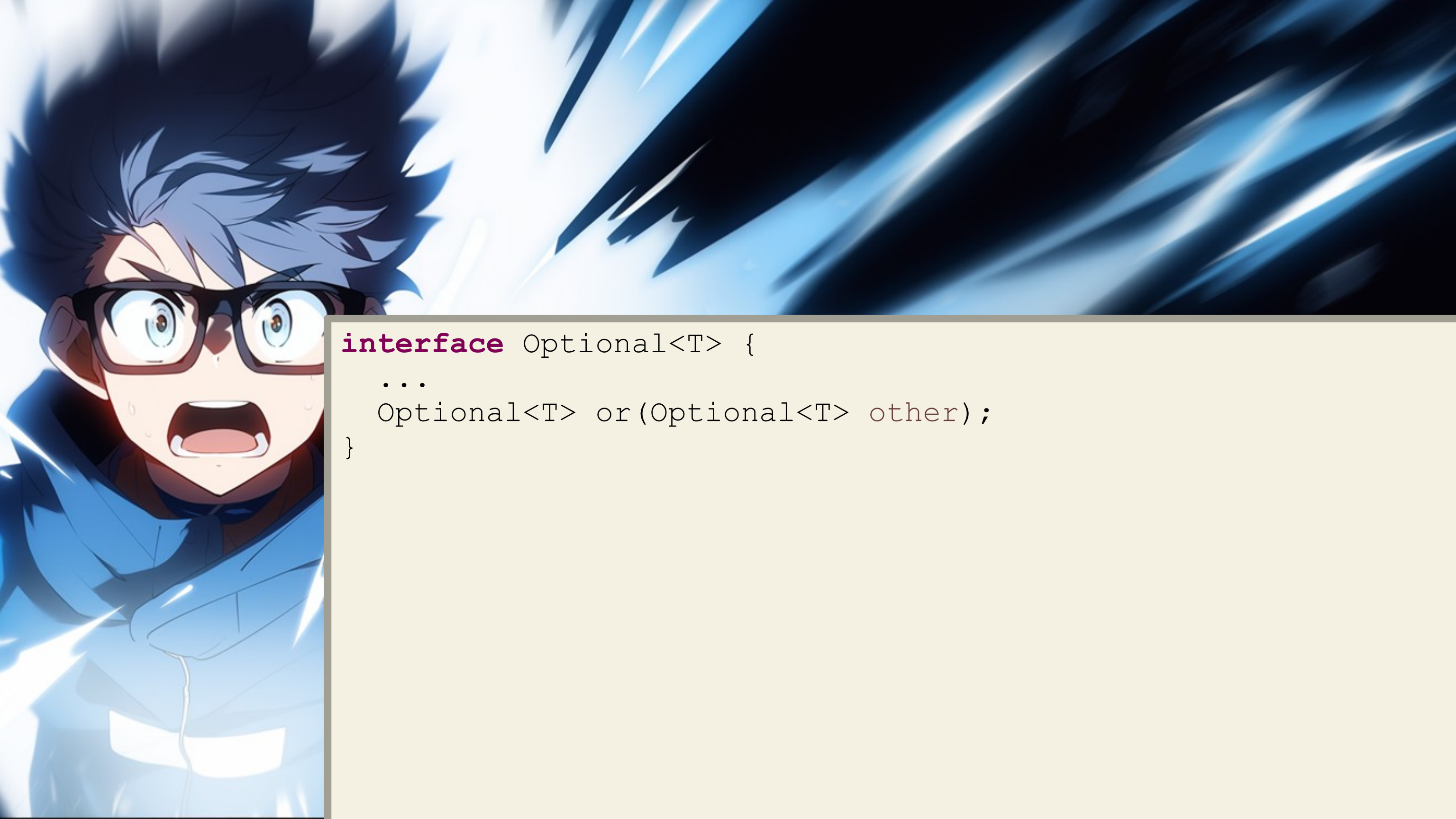
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return other; }  
}
```

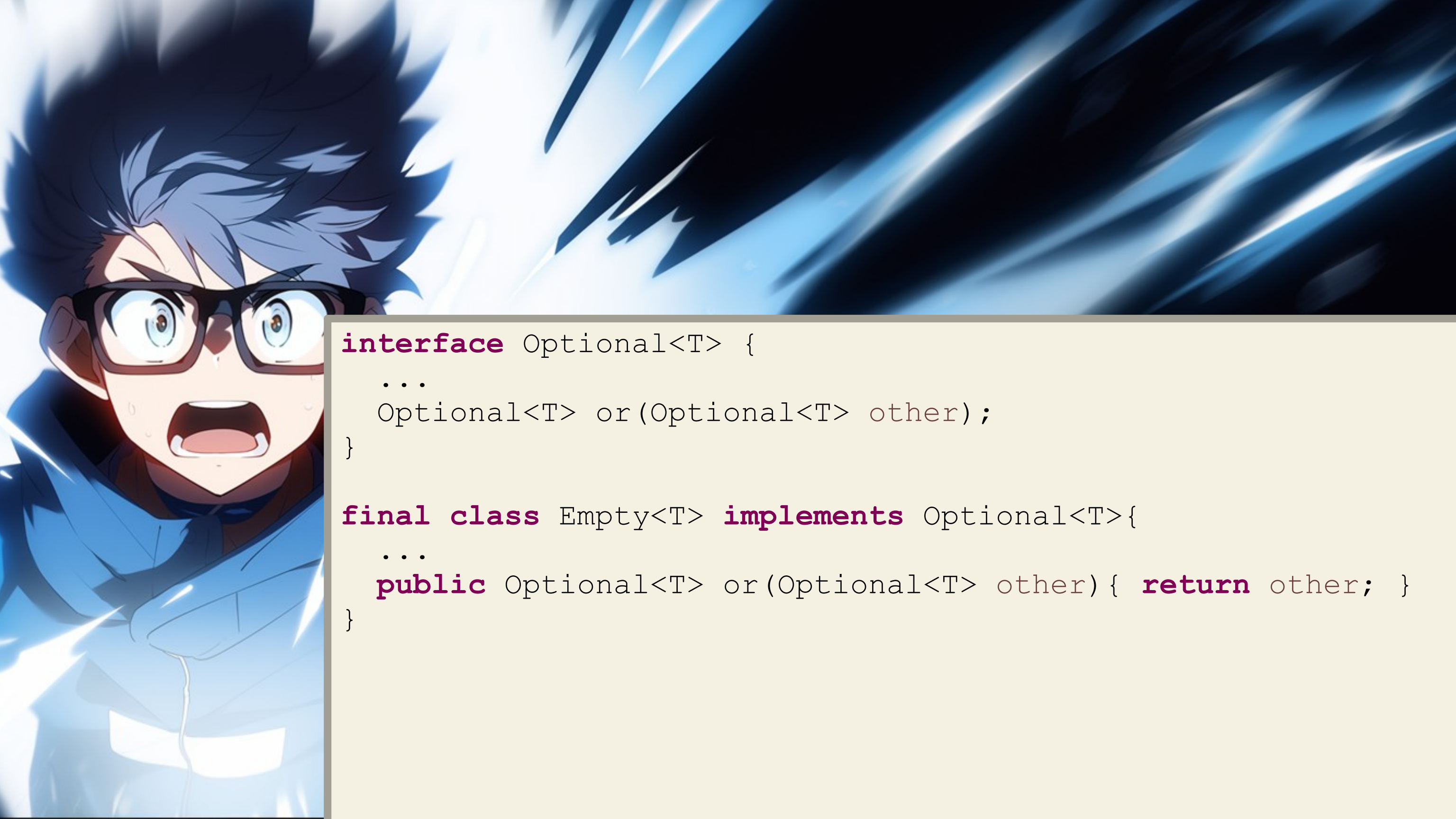
Same for Optional:
We do not have a value.
There is a chance that 'other' has a value.
So we return it.

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return other; }  
}
```

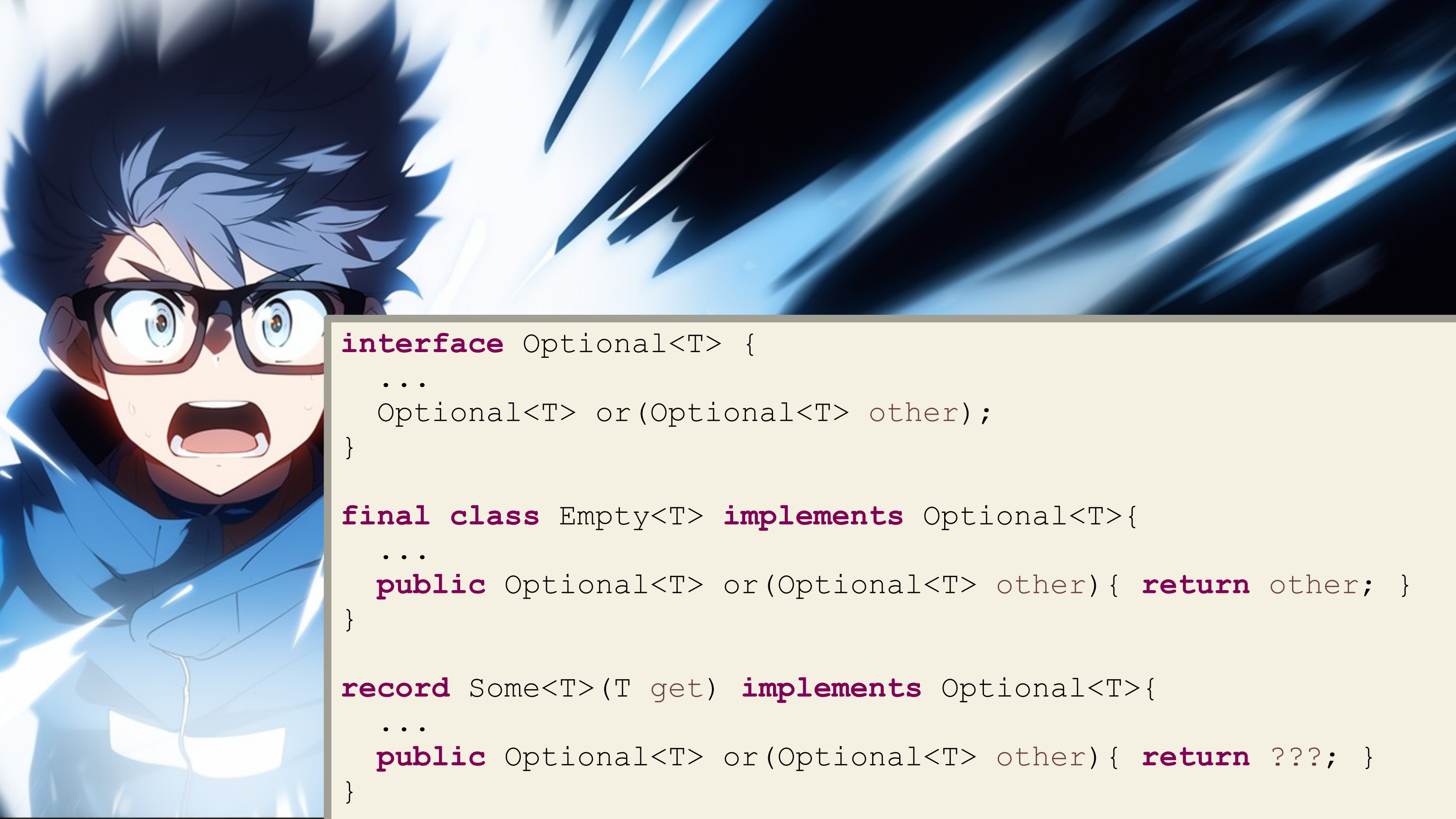




```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```



```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}
```

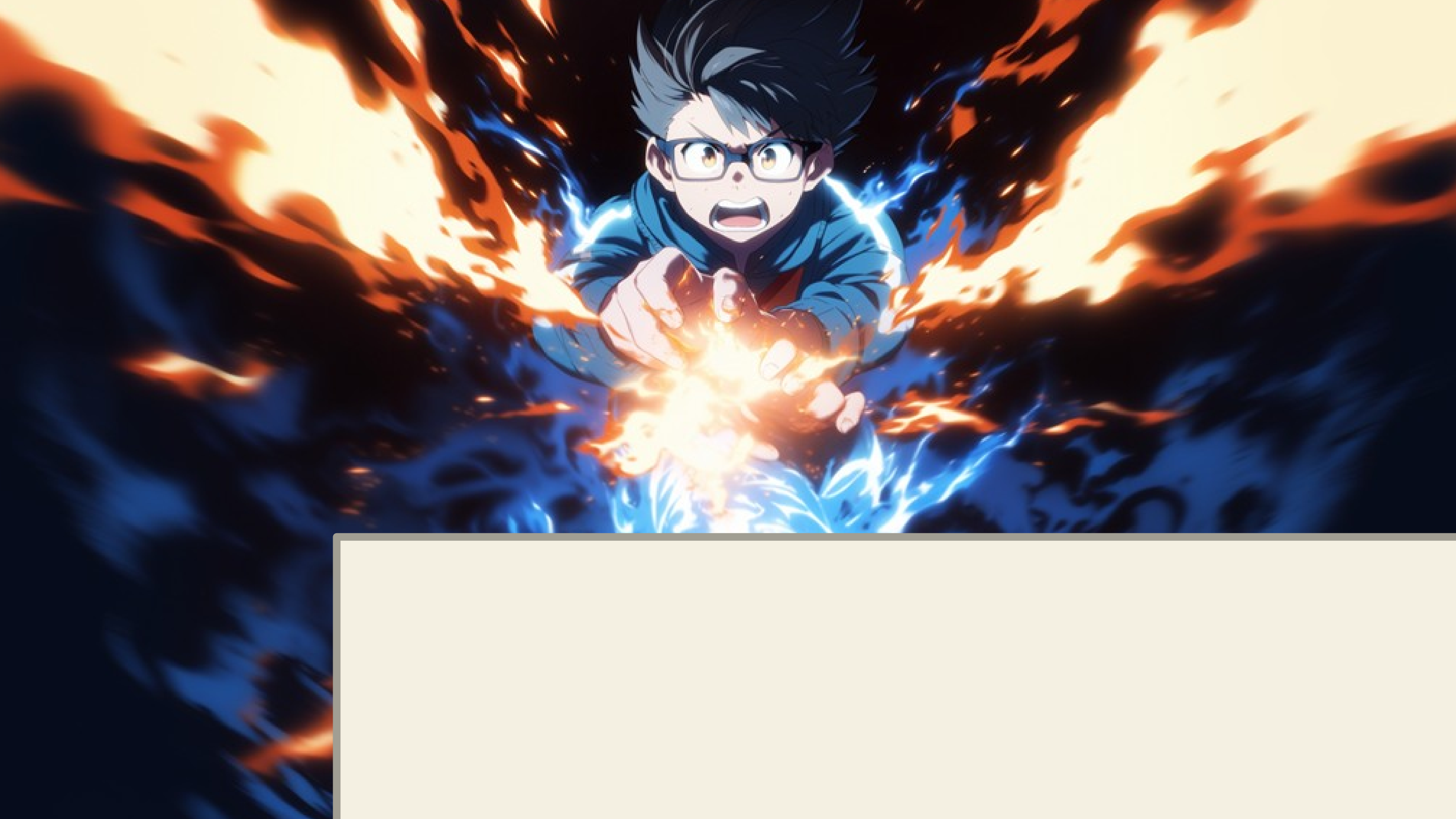



```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return ???; }  
}
```



What about Some?

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return ???; }  
}
```







```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}
```





```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```



“true.or(other)” for booleans

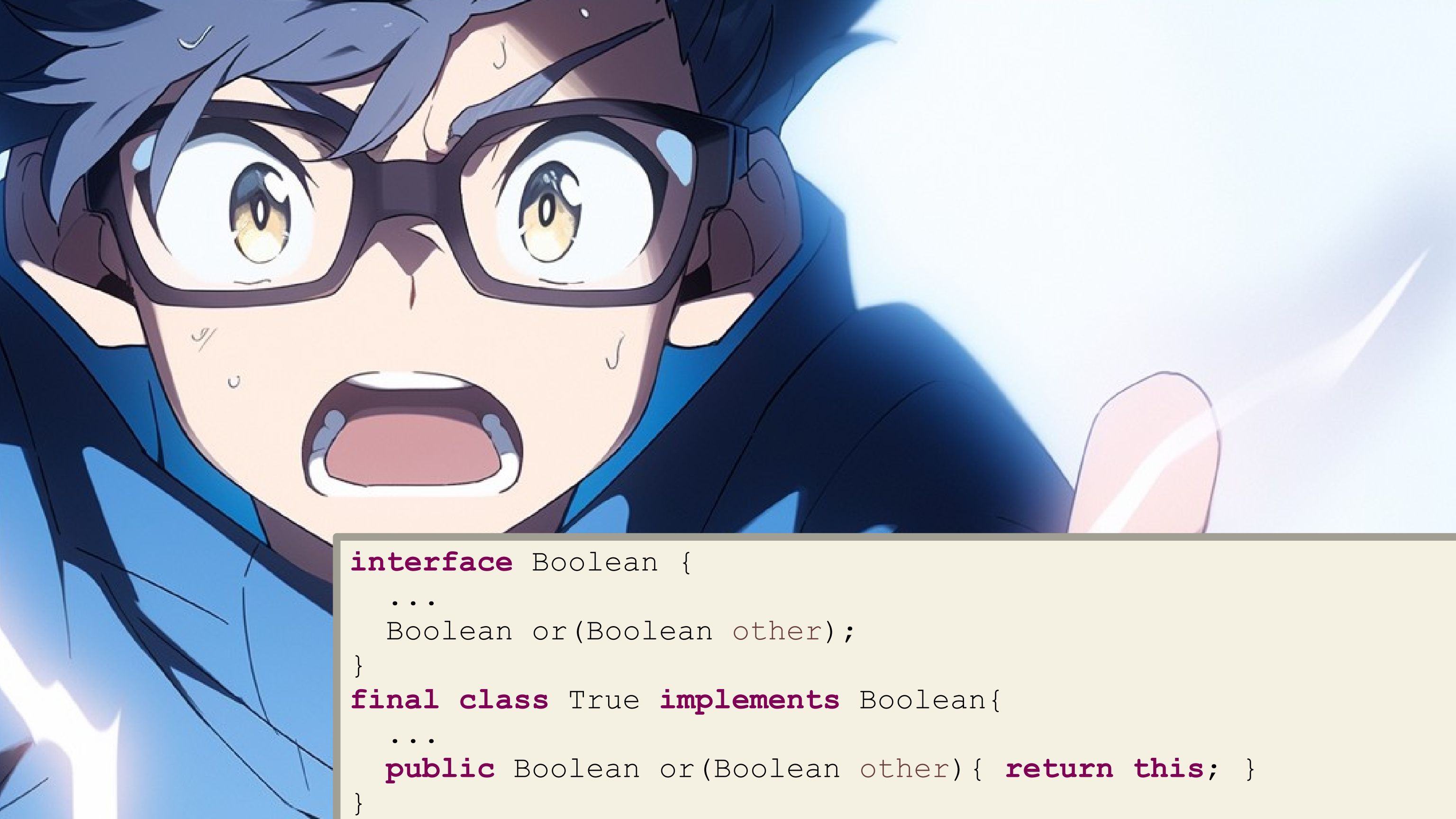
```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```


“true.or(other)” for booleans

Just returns itself
ignoring the argument

```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```

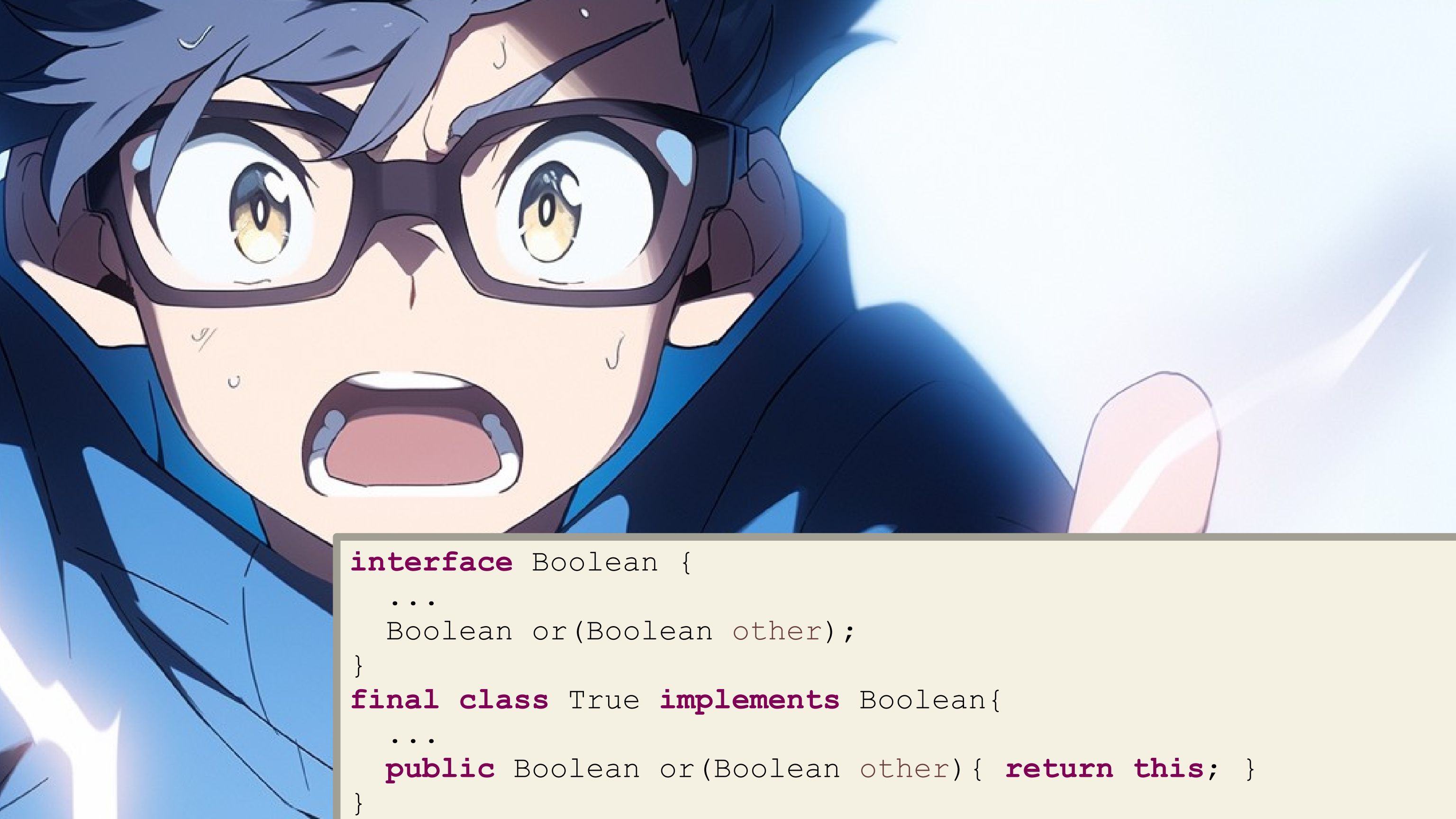


```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```

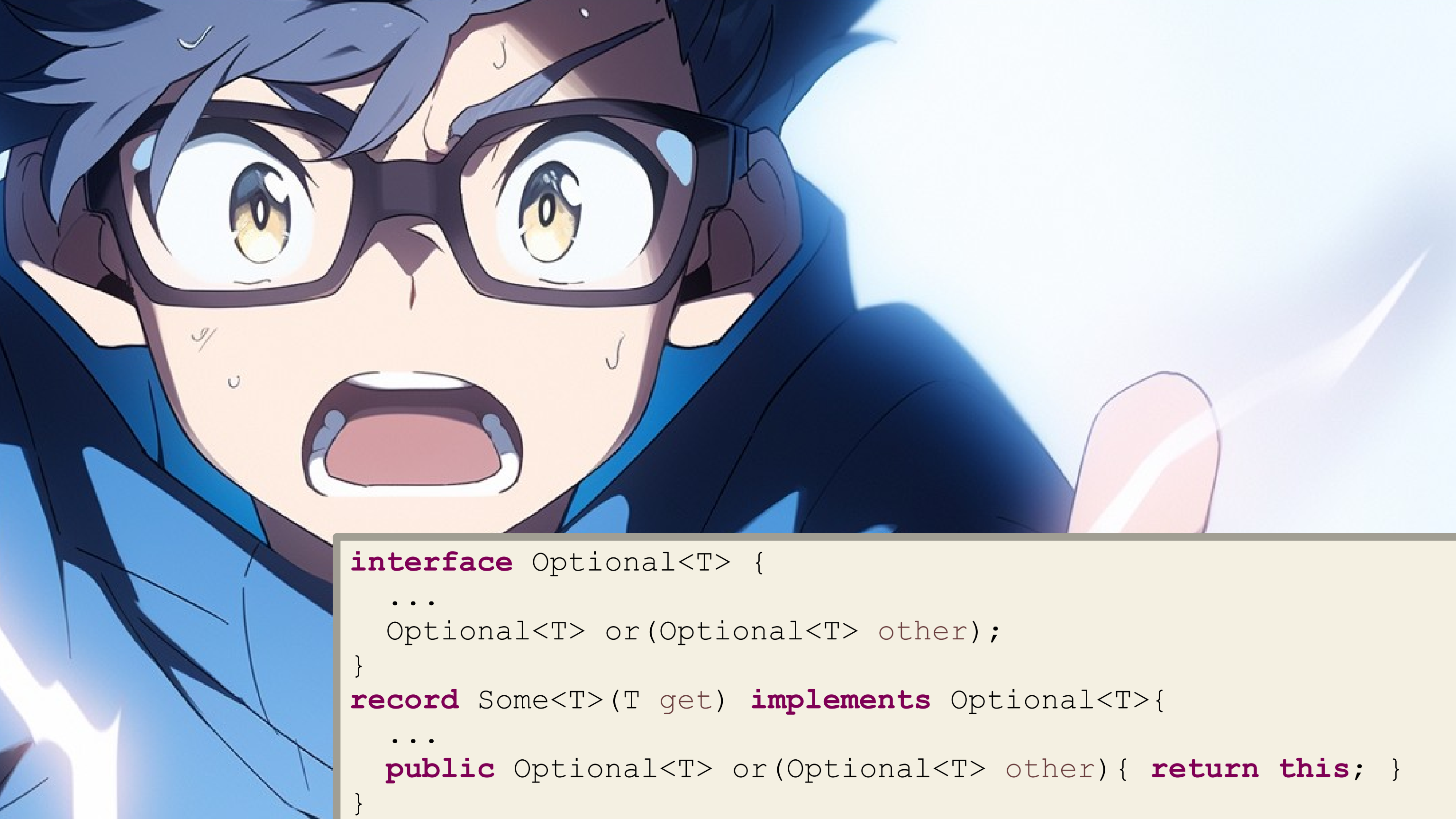



We are 'True', so we can return
'this' or 'new True' as we wish

```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```



```
interface Boolean {  
    ...  
    Boolean or(Boolean other);  
}  
final class True implements Boolean{  
    ...  
    public Boolean or(Boolean other) { return this; }  
}
```

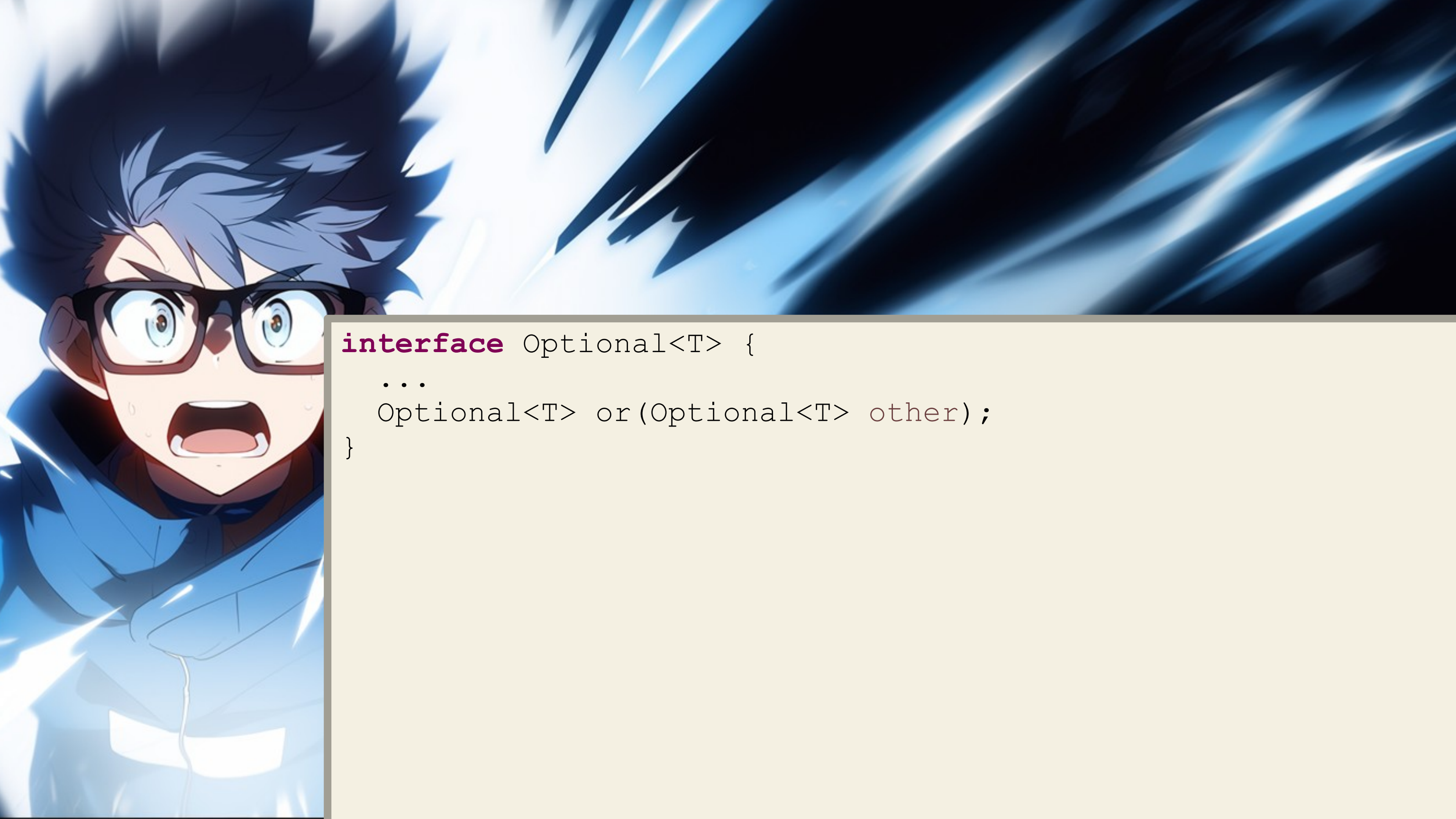
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return this; }  
}
```



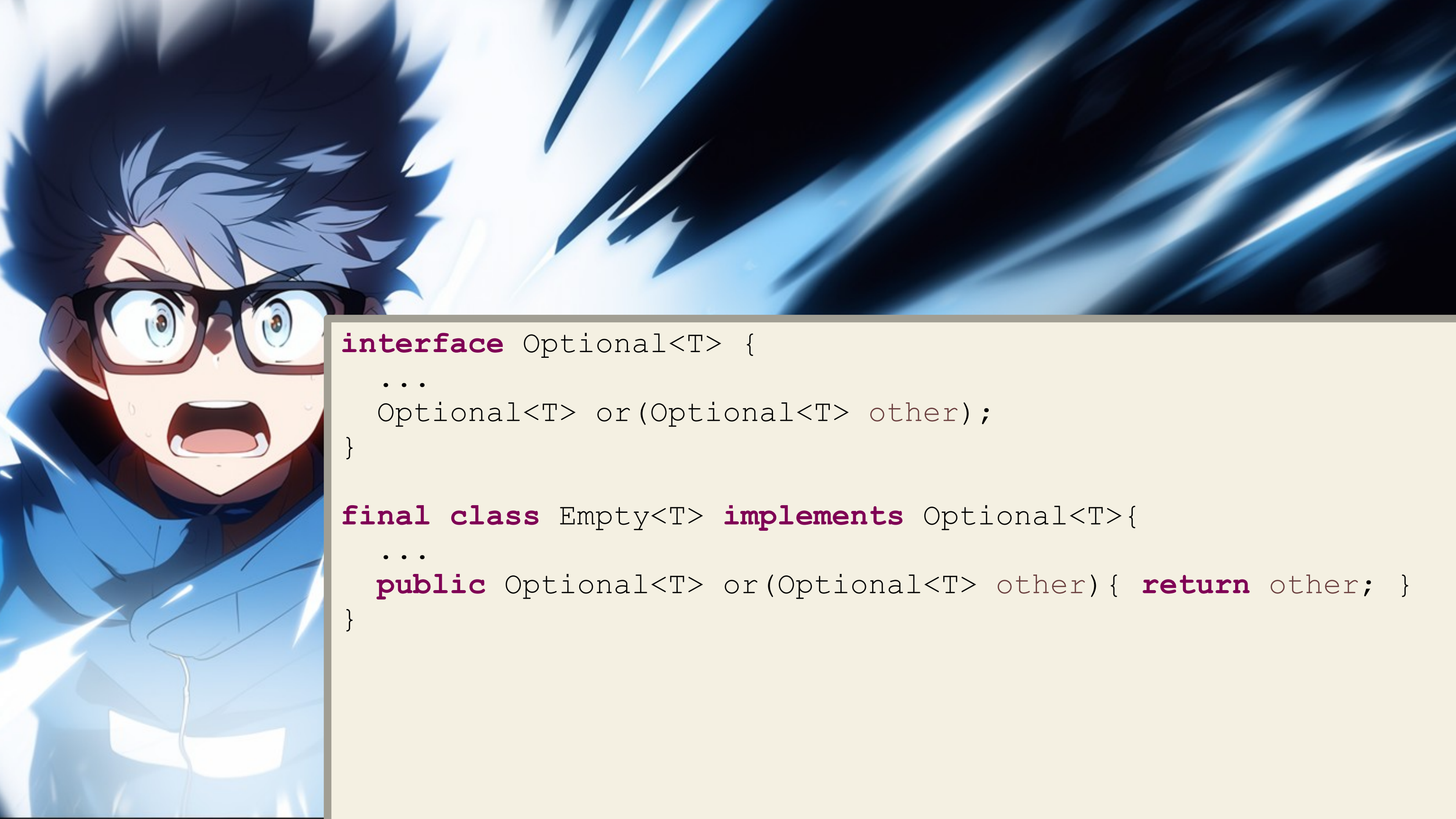
This seems to be working
the same for Optionals

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other){ return this; }  
}
```

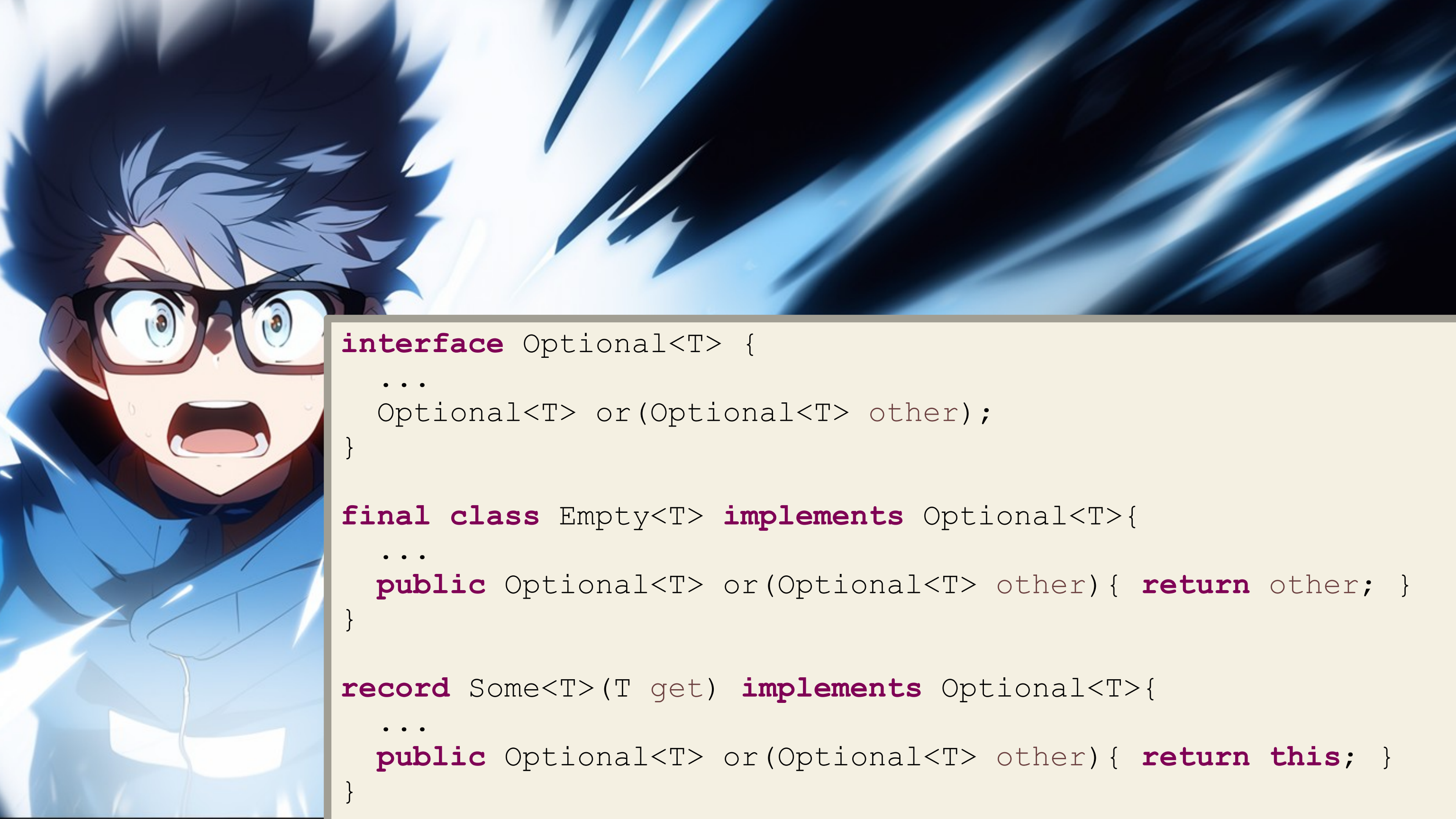


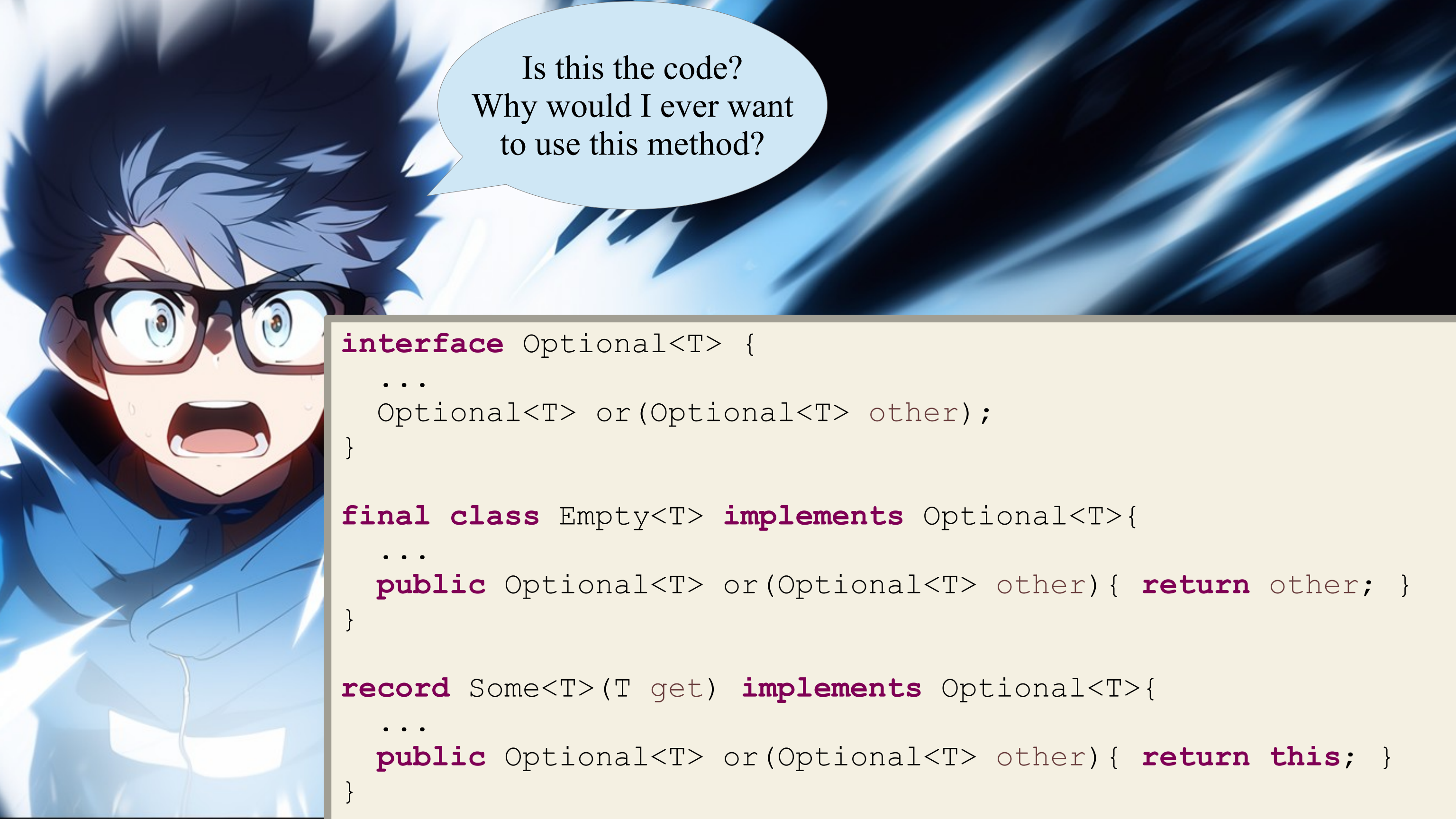
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}
```

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}
```





```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return this; }  
}
```



Is this the code?
Why would I ever want
to use this method?

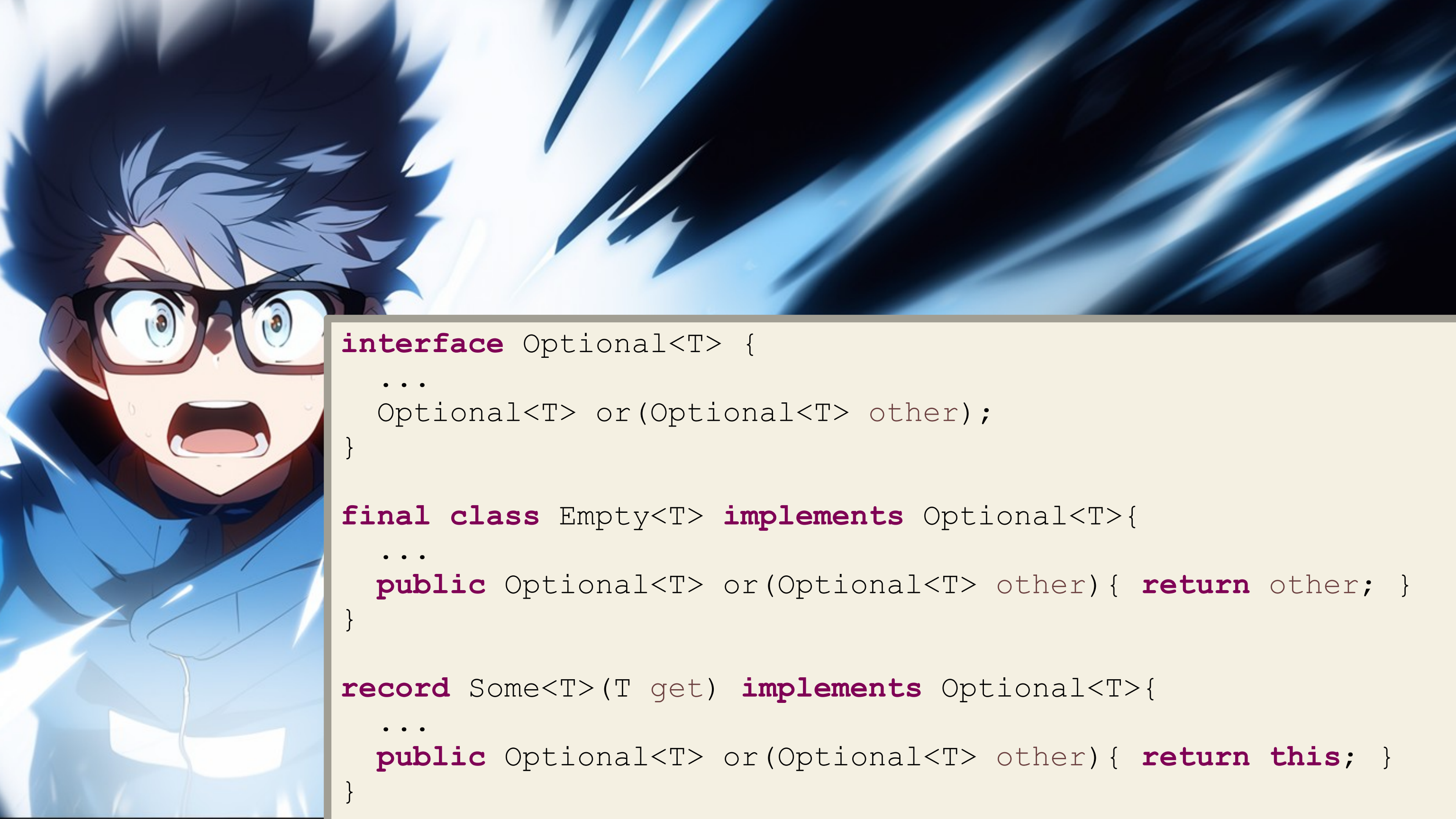
```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return this; }  
}
```

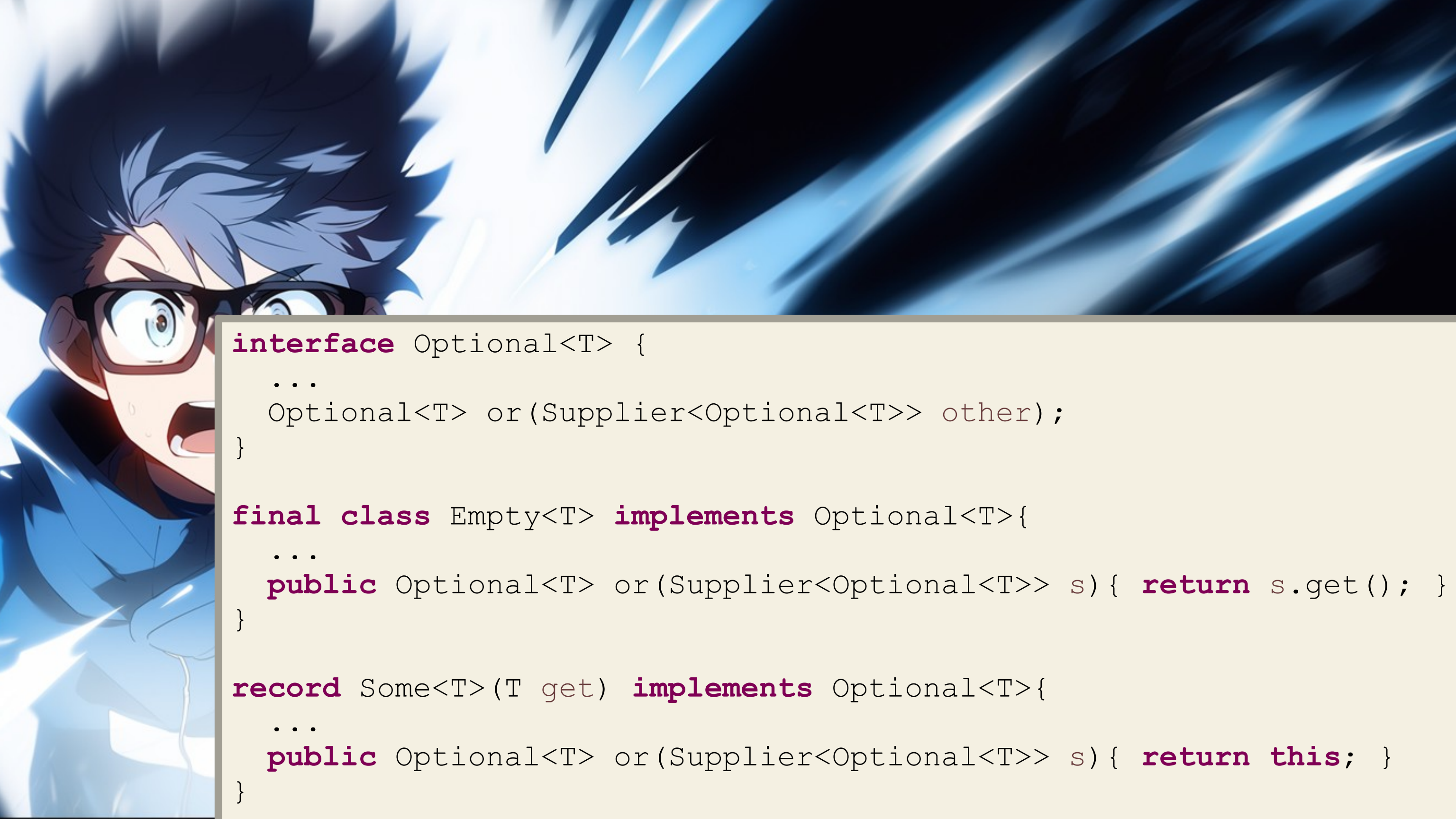
Is this the code?
Why would I ever want
to use this method?

Unless....

```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return this; }  
}
```





```
interface Optional<T> {  
    ...  
    Optional<T> or(Optional<T> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return other; }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Optional<T> other) { return this; }  
}
```

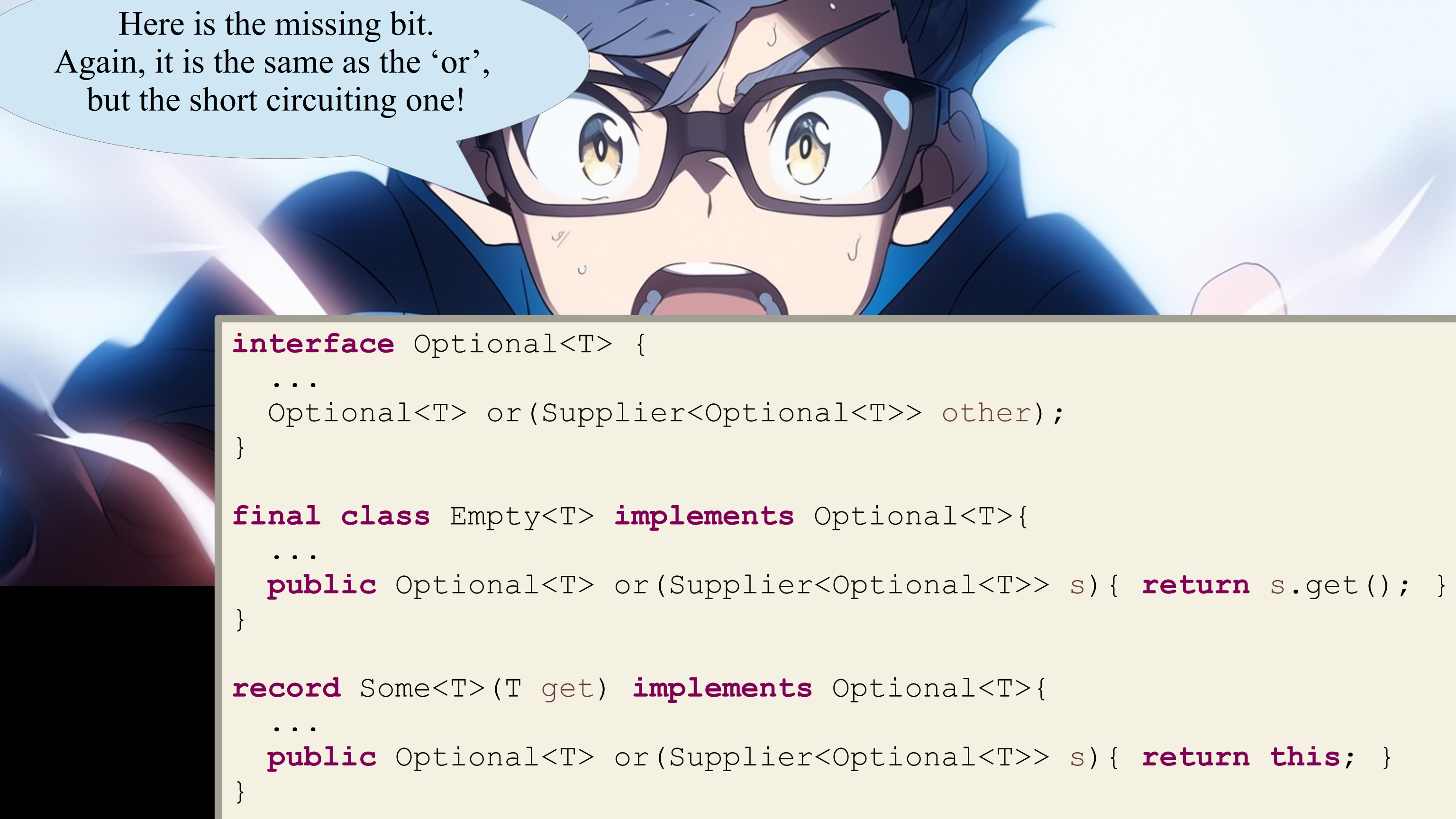


```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```





```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```



Here is the missing bit.
Again, it is the same as the 'or',
but the short circuiting one!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T> (T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```






```
myList.isEmpty() || myList.get(0) != null
```




For example, this code would not call 'get(0)' if the list is empty.

```
myList.isEmpty() || myList.get(0) != null
```



For example, this code would not call 'get(0)' if the list is empty.

```
myList.isEmpty() || myList.get(0) != null
```

```
myOptData.or( () -> allData.stream().filter(x -> x.skill() > 10).findFirst() );
```




For example, this code would not call 'get(0)' if the list is empty.

```
myList.isEmpty() || myList.get(0) != null
```

Similarly, if I have my data already, I would not search in all the data.

```
myOptData.or( () -> allData.stream().filter(x -> x.skill() > 10).findFirst() );
```




With this 'or' method, we can divide
any large method into as many little
ones as we want



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```

```
class Compute{ //return new Compute().of(input);  
    int tmp= 0;  
    int aux;  
    int input;  
    int of(int i){  
        input = i;  
        aux = 10+input;  
        return doBar()  
            .or(()->doFoo())  
            .orElseGet(()->doBeer());  
    }  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```

```
class Compute{ //return new Compute().of(input);  
    int tmp= 0;  
    int aux;  
    int input;  
    int of(int i){  
        input = i;  
        aux = 10+input;  
        return doBar()●  
            .or(()->doFoo())●  
            .orElseGet(()->doBeer());●  
    }  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```

```
class Compute{ //return new Compute().of(input);  
    int tmp= 0;  
    int aux;  
    int input;  
    int of(int i){  
        input = i;  
        aux = 10+input;  
        return doBar()  
            .or(()->doFoo())  
            .orElseGet(()->doBeer());  
    }  
    Optional<Integer> doBar(){  
        while(xxx) {  
            tmp = xxx;  
            if(xxx){ return Optional.of(xxx); }  
            aux = xxx;  
        }  
        return Optional.empty();  
    }  
}
```



With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```

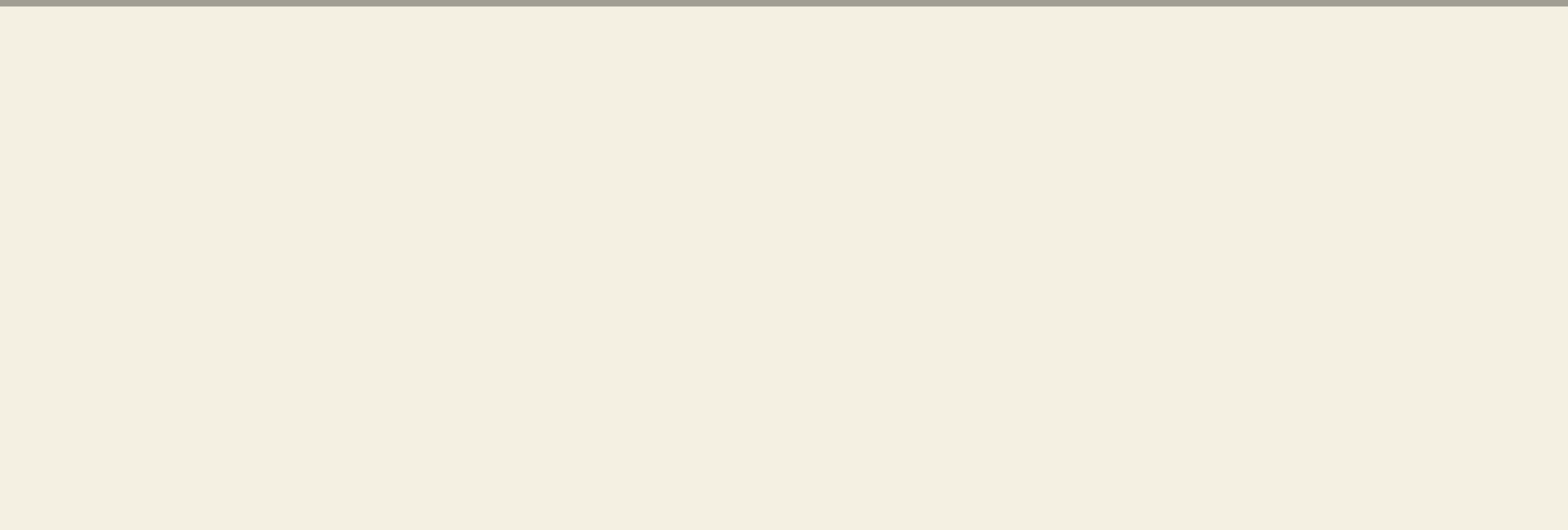
```
class Compute{ //return new Compute().of(input);  
    int tmp= 0;  
    int aux;  
    int input;  
    int of(int i){  
        input = i;  
        aux = 10+input;  
        return doBar()●  
            .or(()->doFoo())●  
            .orElseGet(()->doBeer())●;  
    }  
    Optional<Integer> doBar(){  
        while(xxx) {  
            tmp = xxx;  
            if(xxx){ return Optional.of(xxx); }  
            aux = xxx;  
        }  
        return Optional.empty();  
    }  
    Optional<Integer> doFoo(){  
        for(xxxxx) {  
            tmp = xxx;  
            for(xxxx) {  
                aux = xxx;  
                if(xxx) {return Optional.of(xxx);}  
            }  
        }  
        return Optional.empty();  
    }  
}
```

With this 'or' method, we can divide any large method into as many little ones as we want

```
int badGiantMethod(int input) {  
    int tmp= 0;  
    int aux= 10+input;  
    //do Bar  
    while(xxx) {  
        tmp = xxx;  
        if(xxx){ return xxx; }  
        aux = xxx;  
    }  
    //do Foo  
    for(xxxxx) {  
        tmp = xxx;  
        for(xxxx) {  
            aux = xxx;  
            if(xxx) {return xxx;}  
        }  
    }  
    //do Beer  
    xxxx;  
    return xxx;  
}
```

```
class Compute{ //return new Compute().of(input);  
    int tmp= 0;  
    int aux;  
    int input;  
    int of(int i){  
        input = i;  
        aux = 10+input;  
        return doBar()●  
            .or(()->doFoo())●  
            .orElseGet(()->doBeer())●;  
    }  
    Optional<Integer> doBar(){  
        while(xxx) {  
            tmp = xxx;  
            if(xxx){ return Optional.of(xxx); }  
            aux = xxx;  
        }  
        return Optional.empty();  
    }  
    Optional<Integer> doFoo(){  
        for(xxxxx) {  
            tmp = xxx;  
            for(xxxx) {  
                aux = xxx;  
                if(xxx) {return Optional.of(xxx);}  
            }  
        }  
        return Optional.empty();  
    }  
    int doBeer(){  
        xxxx;  
        return xxx;  
    }  
}
```






```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}
```



```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}
```




```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```



There is still
a mistake in this code!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```


There is still
a mistake in this code!

Pause the video. Can you find the mistake?


The code as shown now, does NOT behave
like the Java Optional.or method
in an important corner case

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

```
interface  
...  
Optional  
}
```

```
final class  
...  
public  
}
```

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




There is still
a mistake in this code!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```


There is still
a mistake in this code!

```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```





```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return s.get(); }  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```





```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) {  
        return Objects.requireNonNull(s.get());  
    } //Java tries to prevent null optionals  
}  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s) { return this; }  
}
```



```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s){  
        return Objects.requireNonNull(s.get());  
    } //Java tries to prevent null optionals  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }  
}
```


Objects.requireNonNull
would throw an error if the supplier
returns 'null' instead of a valid Optional

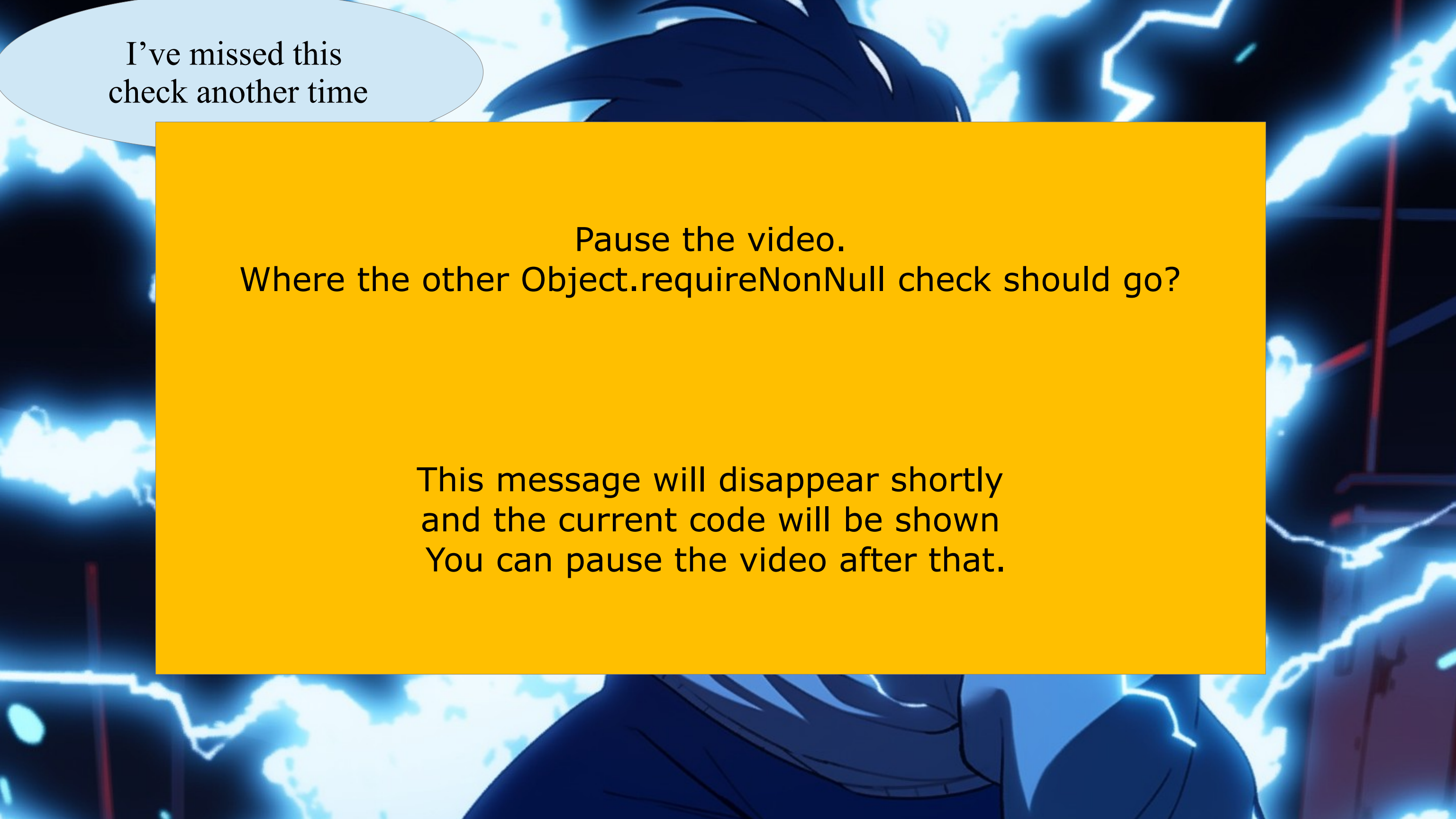


```
interface Optional<T> {  
    ...  
    Optional<T> or(Supplier<Optional<T>> other);  
}  
  
final class Empty<T> implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s){  
        return Objects.requireNonNull(s.get());  
    } //Java tries to prevent null optionals  
}  
  
record Some<T>(T get) implements Optional<T>{  
    ...  
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }  
}
```




I've missed this
check another time





I've missed this
check another time

Pause the video.
Where the other `Object.requireNonNull` check should go?

This message will disappear shortly
and the current code will be shown
You can pause the video after that.


```

public sealed interface Optional<T> extends Serializable permits Empty<T>, Some<T>{
    @SuppressWarnings("unchecked")
    static <E> Optional<E> empty(){ return (Optional<E>)Empty.Instance; }
    static <T> Optional<T> of(T value){ return new Some<T>(Objects.requireNonNull(value)); }
    static <T> Optional<T> ofNullable(T value){ return value == null ? empty() : new Some<T>(value); }
    T orElseGet(Supplier<T> s);
    Optional<T> filter(Predicate<T> p);
    <U> Optional<U> map(Function<T, U> m);
    <U> Optional<U> flatMap(Function<T, Optional<U>> m);
    Optional<T> or(Supplier<Optional<T>> s);
}

```

```

final class Empty<T> implements Optional<T>{
    static final Empty<Object> instance= new Empty<>();
    private Empty(){ }
    public T orElseGet(Supplier<T> s){ return s.get(); }
    public Optional<T> filter(Predicate<T> p){ return Optional.empty(); }
    public <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
    public <U> Optional<U> flatMap(Function<T, Optional<U>> m){ return Optional.empty(); }
    public Optional<T> or(Supplier<Optional<T>> s){ return Objects.requireNonNull(s.get()); }
}

```

```

record Some<T>(T get) implements Optional<T>{
    public T orElseGet(Supplier<T> unused){ return get; }
    public Optional<T> filter(Predicate<T> p){ return p.test(get)?this:Optional.empty(); }
    public <U> Optional<U> map(Function<T, U> m){ return m.apply(get); }
    public <U> Optional<U> flatMap(Function<T, Optional<U>> m){ return m.apply(get); }
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }
}

```

I've missed this
check another time







```
record Some<T>(T get) implements Optional<T>{  
  public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
    return m.apply(get);  
  }  
}
```




```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return m.apply(get);  
    }  
}
```



```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return Objects.requireNonNull(m.apply(get));  
    }  
}
```

RequireNotNull is needed
also for Some.flatMap

```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return m.apply(get);  
    }  
}
```



```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return Objects.requireNonNull(m.apply(get));  
    }  
}
```


RequireNotNull is needed
also for Some.flatMap

```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return m.apply(get);  
    }  
}
```



Or bad user code may
return a null Optional here!

```
record Some<T>(T get) implements Optional<T>{  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> m) {  
        return Objects.requireNonNull(m.apply(get));  
    }  
}
```





Technically, flatMap can emulate
both map and filter



Technically, flatMap can emulate
both map and filter

```
myOpt.map(e -> ...e...);  
myOpt.flatMap(e -> Optional.ofNullable(...e...));
```




Technically, flatMap can emulate
both map and filter

```
myOpt.map(e -> ...e...);  
myOpt.flatMap(e -> Optional.ofNullable(...e...));
```

```
myOpt.filter(e -> ...e...);  
myOpt.flatMap(e -> (...e...) ? myOpt : Optional.empty());
```




Technically, flatMap can emulate
both map and filter

But I still think I should include them,
since they are so important

```
myOpt.map(e -> ...e...);  
myOpt.flatMap(e -> Optional.ofNullable(...e...));
```

```
myOpt.filter(e -> ...e...);  
myOpt.flatMap(e -> (...e...) ? myOpt : Optional.empty());
```






Time to submit again!


```

public sealed interface Optional<T> extends Serializable permits Empty<T>, Some<T>{
    @SuppressWarnings("unchecked")
    static <E> Optional<E> empty(){ return (Optional<E>)Empty.Instance; }
    static <T> Optional<T> of(T value){ return new Some<T>(Objects.requireNonNull(value)); }
    static <T> Optional<T> ofNullable(T value){ return value == null ? empty() : new Some<T>(value); }
    T orElseGet(Supplier<T> s);
    Optional<T> filter(Predicate<T> p);
    <U> Optional<U> map(Function<T, U> m);
    <U> Optional<U> flatMap(Function<T, Optional<U>> m);
    Optional<T> or(Supplier<Optional<T>> s);
}

```

```

final class Empty<T> implements Optional<T>{
    static final Empty<Object> instance= new Empty<>();
    private Empty(){ }
    public T orElseGet(Supplier<T> s){ return s.get(); }
    public Optional<T> filter(Predicate<T> p){ return Optional.empty(); }
    public <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
    public <U> Optional<U> flatMap(Function<T, Optional<U>> m){ return Optional.empty(); }
    public Optional<T> or(Supplier<Optional<T>> s){ return Objects.requireNonNull(s.get()); }
}

```

```

record Some<T>(T get) implements Optional<T>{
    public T orElseGet(Supplier<T> unused){ return get; }
    public Optional<T> filter(Predicate<T> p){ return p.test(get)?this:Optional.empty(); }
    public <U> Optional<U> map(Function<T, U> m){ return Optional.ofNullable(m.apply(get)); }
    public <U> Optional<U> flatMap(Function<T, Optional<U>> m){
        return Objects.requireNonNull(m.apply(get));
    }
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }
}

```



***Is this your
submission?
Really?***



***Is this your
submission?
Really?***



***Is this your
submission?
Really?***

***Ok, this is
over the minimum
submission bar.***



***Is this your
submission?
Really?***

***Ok, this is
over the minimum
submission bar.***

***Your mark will be
computed and if over
60%; it will be
capped to 60%.***





Dany have now completed the final first year UPU exam



Dany have now completed the final first year UPU exam

And, it is done!

Dany have now completed the final first year UPU exam

And, it is done!

All good; what could
have gone wrong?



Sleep tight Dany, tomorrow they will publish the results and the model solutions



Sleep tight Dany, tomorrow they will publish the results and the model solutions



Either way, this is going to be your last night in this bed

If you passed, you will move to the Ivory tower
If you failed, you will be sent back home





Credits





Credits

- Story: Marco
- Art: MidJourney, NijiJourney, Dall-E
- - Wording: Marco, chatGPT, jfw01
- Composition: Marco
- Thanks to all my friends for providing great feedback!



Credits

- Story: Marco
- Art: MidJourney, NijiJourney, Dall-E
- - Wording: Marco, chatGPT, jfw01
- Composition: Marco
- Thanks to all my friends for providing great feedback!