

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

```
enum Empty implements Optional<Object> { Instance; }
```

Pupon's full solution 1/2



```

public sealed interface Optional<T>
    extends Serializable permits Empty, 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);
    }
    default T orElseGet(Supplier<T> s){ return s.get(); }
    default Optional<T> or(Supplier<Optional<T>> s){
        return Objects.requireNonNull(s.get());
    }
    default <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
    default Optional<T> filter(Predicate<T> p){
        return map(e->p.test(e) ? e : null);
    }
    default <U> Optional<U> flatMap(Function<T, Optional<U>> m){
        return Optional.empty();
    }
}

```

Pupon's full solution 2/2

```
record Some<T>(T get) implements Optional<T>{  
    public T orElseGet(Supplier<T> unused){ return get; }  
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }  
    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));  
    }  
}  
  
enum Empty implements Optional<Object>{ Instance; }
```



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

```
enum Empty implements Optional<Object> { Instance; }
```

*The code starts
by showing us
the 'Some' case*




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

```
enum Empty implements Optional<Object> { Instance; }
```

*The code starts
by showing us
the 'Some' case*

This case looks pretty much
identical to what I wrote



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

```
enum Empty implements Optional<Object> { Instance; }
```

*The code starts
by showing us
the 'Some' case*

This case looks pretty much
identical to what I wrote

But somehow, 'filter' is missing!



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

```
enum Empty implements Optional<Object>{ Instance; }
```




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

```
enum Empty implements Optional<Object>{ Instance; }
```




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

```
enum Empty implements Optional<Object> { Instance; }
```



Wait.. why is the empty
case an ENUM?!?



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

```
enum Empty implements Optional<Object> { Instance; }
```



Wait.. why is the empty
case an ENUM?!?

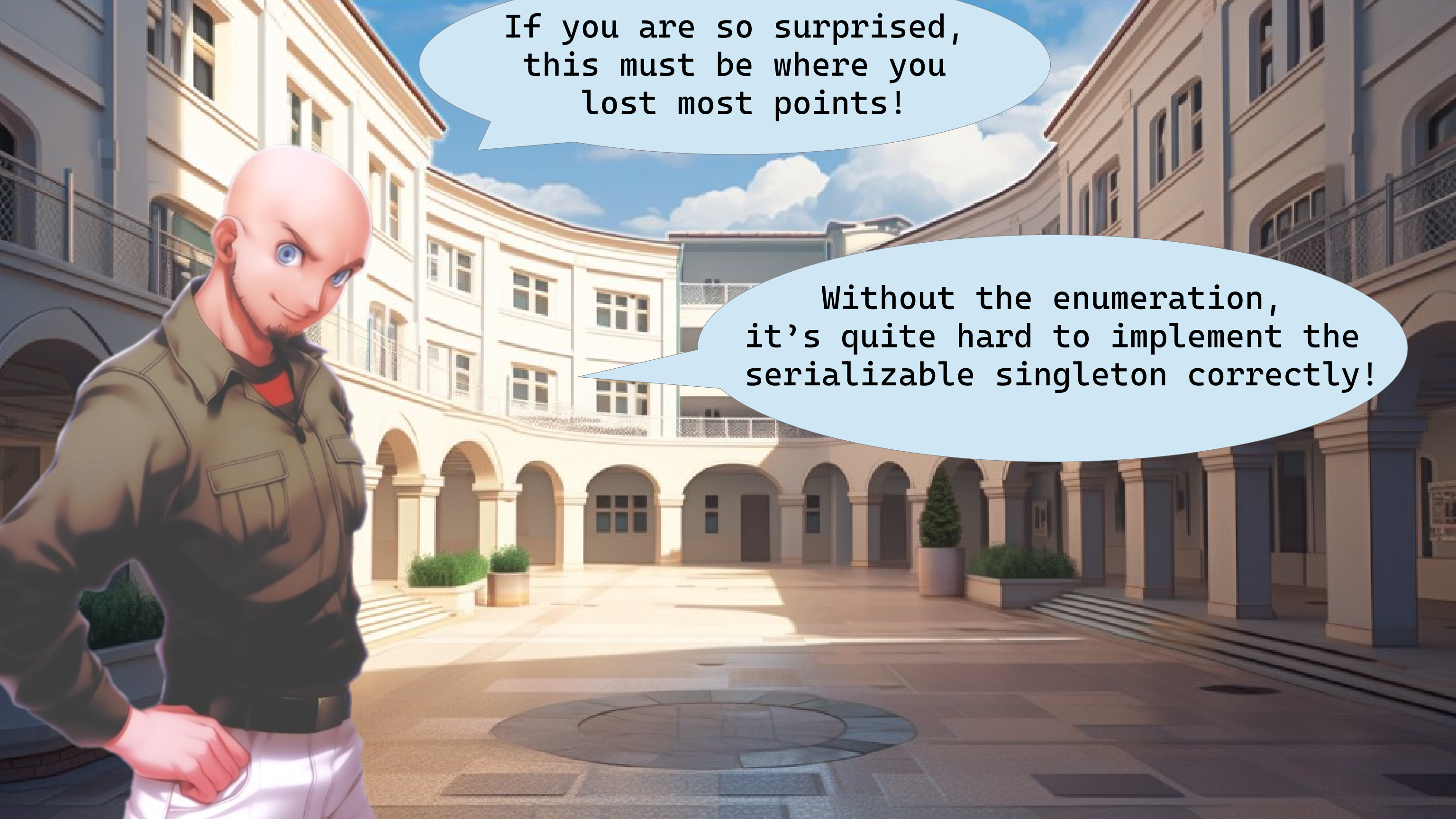
Not even a reasonable enum, but an
absurd enum with a single element!





If you are so surprised,
this must be where you
lost most points!





If you are so surprised,
this must be where you
lost most points!

Without the enumeration,
it's quite hard to implement the
serializable singleton correctly!




If you just implement
a singleton naively



If you just implement
a singleton naively

by serializing it
and deserializing it..




A 3D rendered character, resembling a stylized soldier or hero, stands in the center of a courtyard. He is bald with blue eyes, wearing a dark green tactical shirt and white pants. He has a red hand on his hip and the other raised in a 'V' sign. The background features a large, multi-story building with arched windows and balconies, under a blue sky with clouds. Three speech bubbles are present, containing text about singleton implementation.

If you just implement
a singleton naively

by serializing it
and deserializing it..

you will end up
with two instances
of it in memory at
the same time!




If you just implement
a singleton naively

by serializing it
and deserializing it...

you will end up
with two instances
of it in memory at
the same time!

```
new Empty()
```



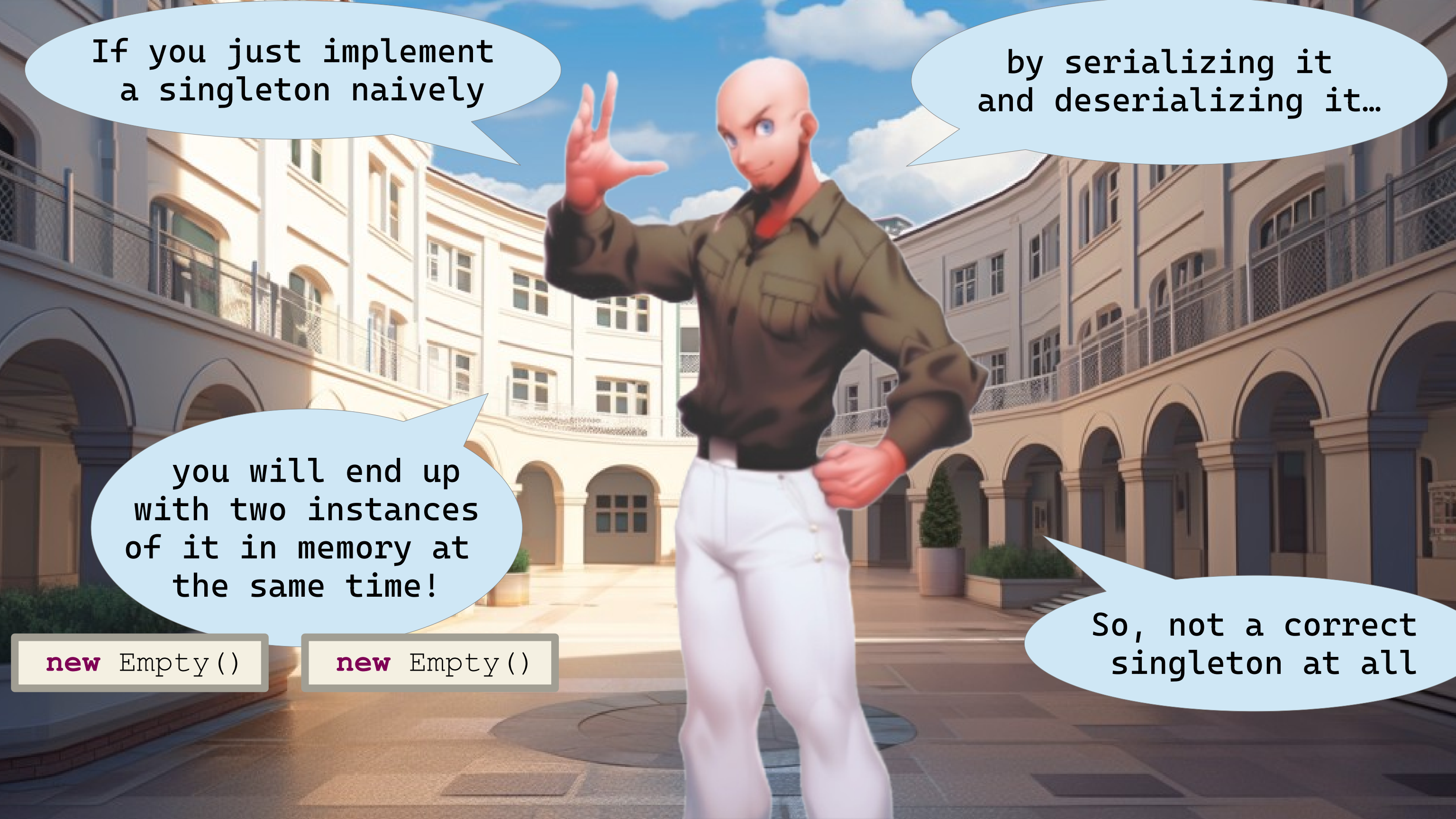
If you just implement
a singleton naively

by serializing it
and deserializing it...

you will end up
with two instances
of it in memory at
the same time!

```
new Empty()
```

```
new Empty()
```

If you just implement
a singleton naively

by serializing it
and deserializing it..


you will end up
with two instances
of it in memory at
the same time!

```
new Empty()
```

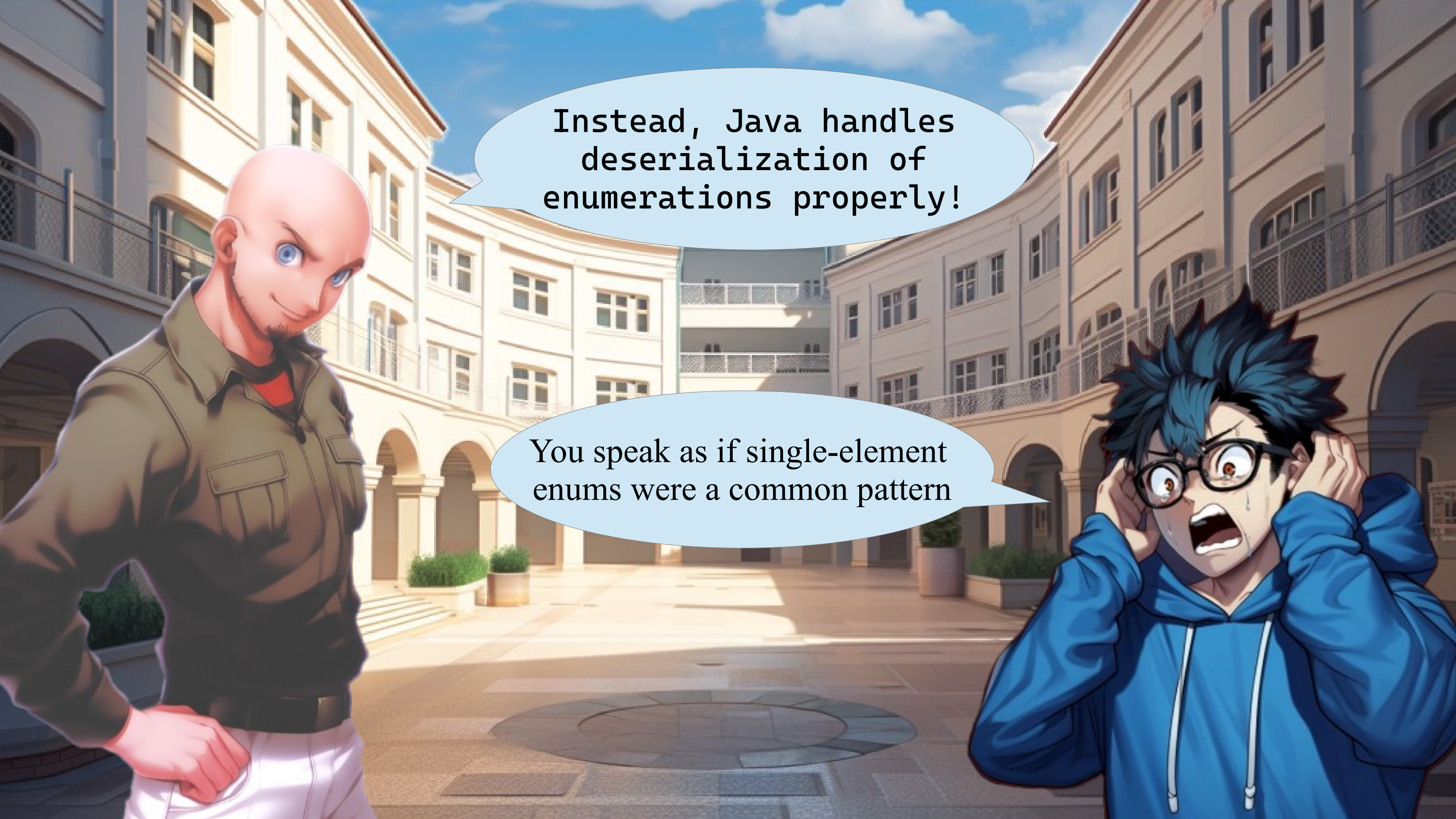
```
new Empty()
```

So, not a correct
singleton at all



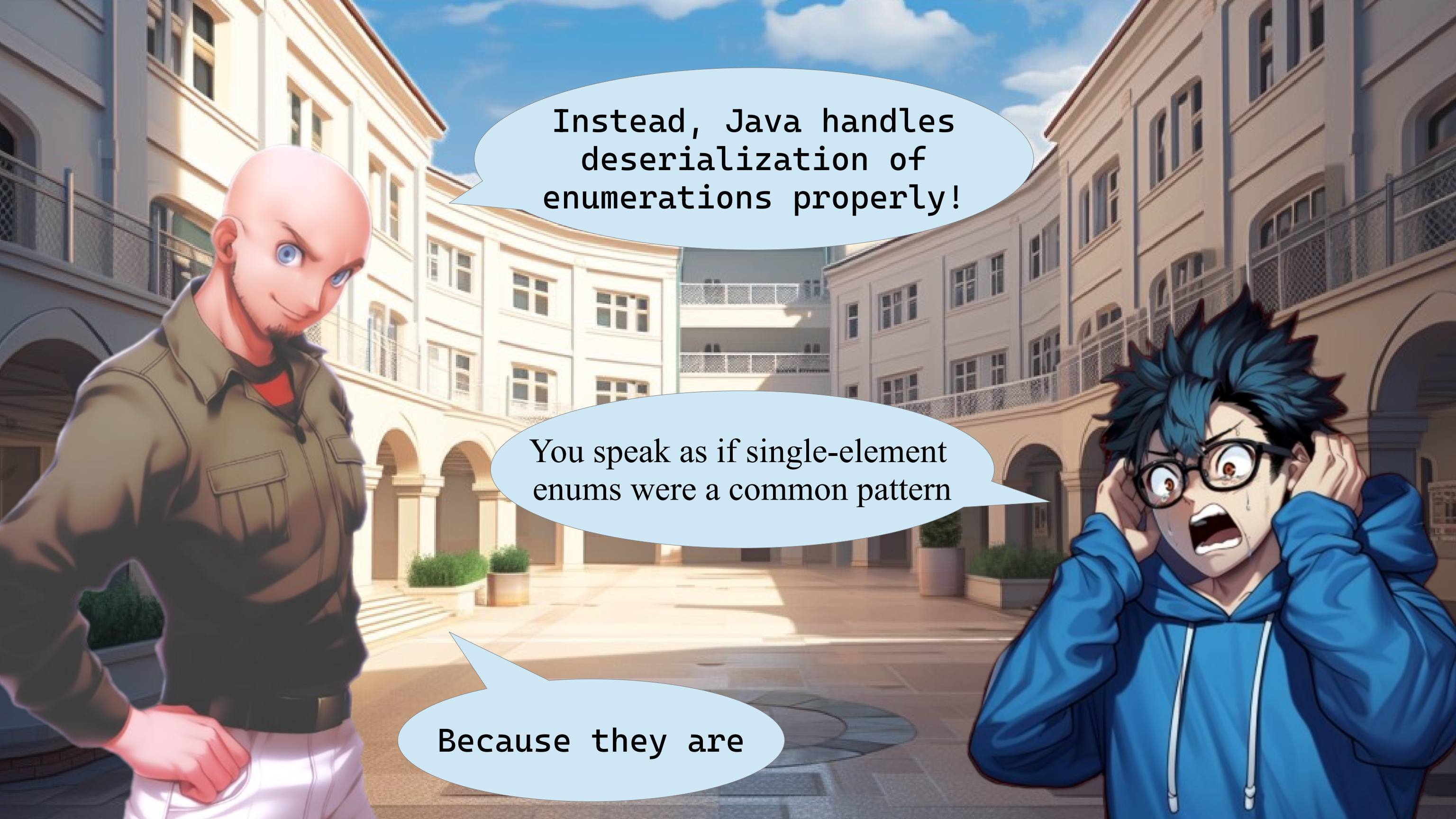
The image features two anime-style characters in a courtyard. On the left, a bald man with blue eyes and a goatee, wearing a green military-style shirt and white pants, stands with his hand on his hip. On the right, a man with spiky blue hair, glasses, and a blue hoodie looks shocked, with his hands to his ears. A speech bubble from the man on the left contains the text:

Instead, Java handles
deserialization of
enumerations properly!



Instead, Java handles
deserialization of
enumerations properly!

You speak as if single-element
enums were a common pattern



Instead, Java handles
deserialization of
enumerations properly!

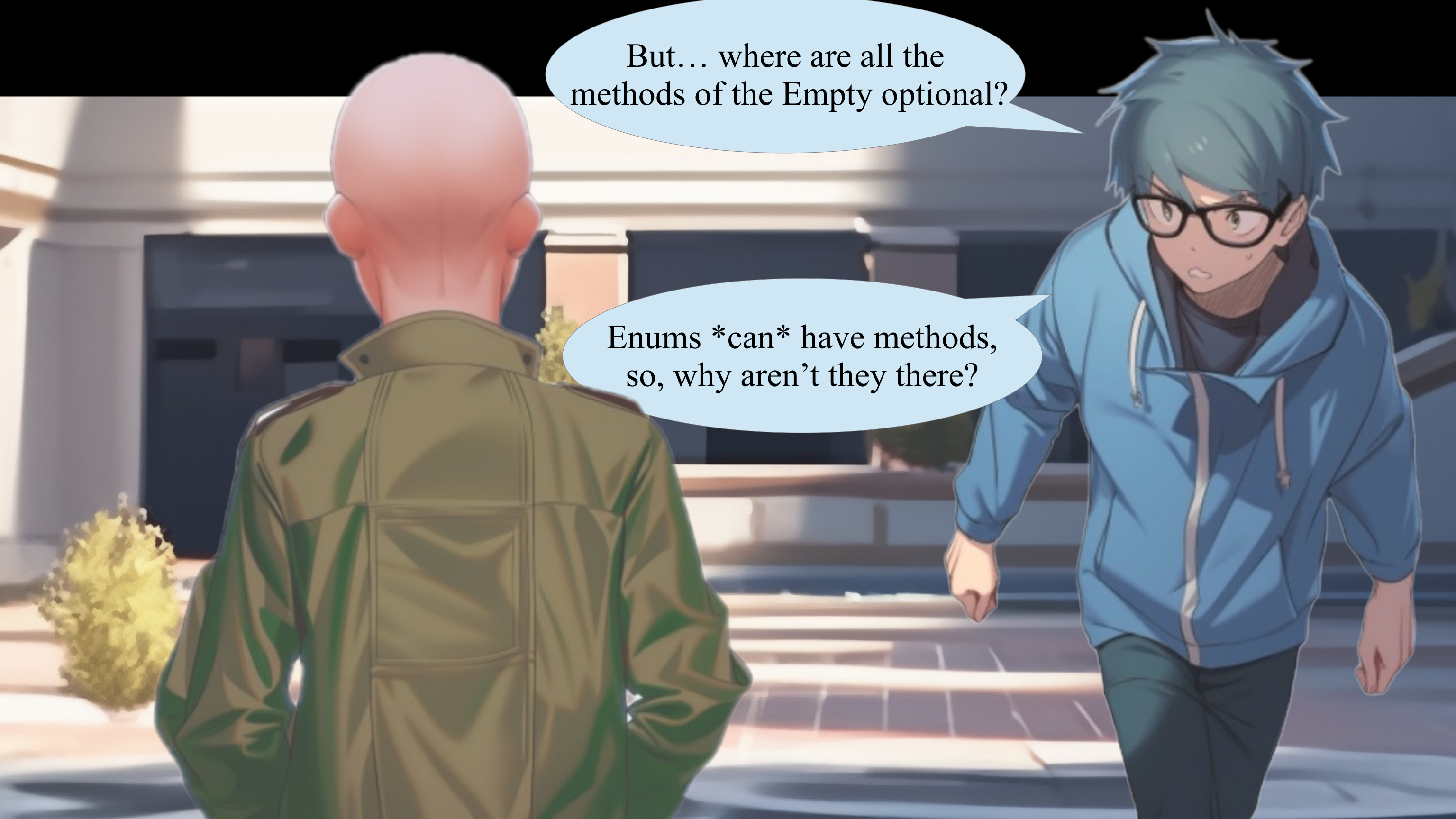
You speak as if single-element
enums were a common pattern

Because they are



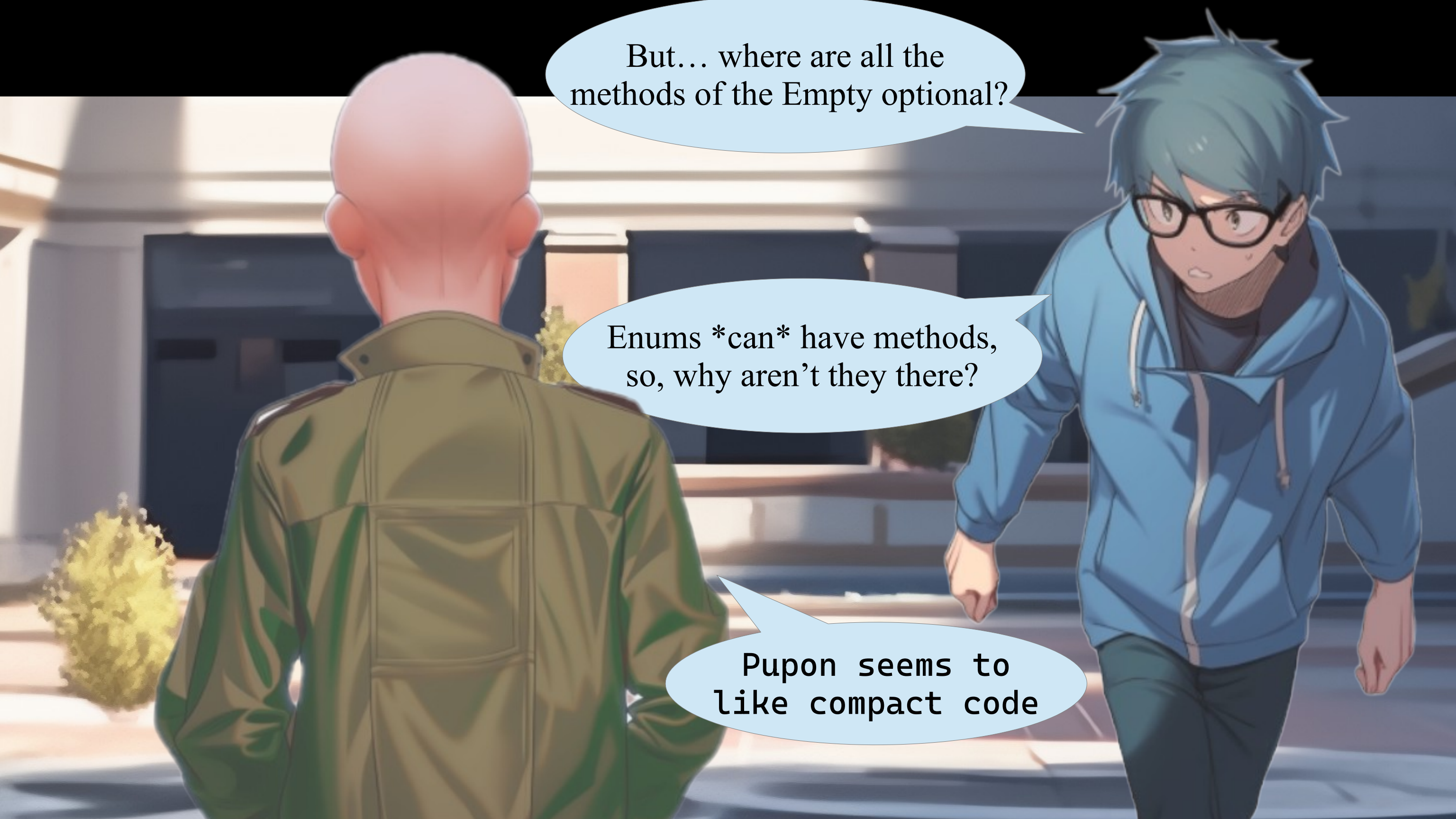
But... where are all the
methods of the Empty optional?





But... where are all the methods of the Empty optional?

Enums **can** have methods, so, why aren't they there?



But... where are all the
methods of the Empty optional?

Enums **can** have methods,
so, why aren't they there?

Pupon seems to
like compact code

```

public sealed interface Optional<T>
    extends Serializable permits Empty, 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);
    }
    default T orElseGet(Supplier<T> s){ return s.get(); }
    default Optional<T> or(Supplier<Optional<T>> s){
        return Objects.requireNonNull(s.get());
    }
    default <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
    default Optional<T> filter(Predicate<T> p){
        return map(e->p.test(e) ? e : null);
    }
    default <U> Optional<U> flatMap(Function<T, Optional<U>> m){
        return Optional.empty();
    }
}

```



There's no need to declare them as abstract in Optional

```
public sealed interface Optional<T>
    extends Serializable permits Empty, Some
@SuppressWarnings("unchecked")
static <E> Optional<E> empty() { return (Optional<E>) Empty; }
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);
}
default T orElseGet(Supplier<T> s) { return s.get(); }
default Optional<T> or(Supplier<Optional<T>> s) {
    return Objects.requireNonNull(s.get());
}
default <U> Optional<U> map(Function<T, U> m) { return Optional.empty(); }
default Optional<T> filter(Predicate<T> p) {
    return map(e->p.test(e) ? e : null);
}
default <U> Optional<U> flatMap(Function<T, Optional<U>> m) {
    return Optional.empty();
}
}
```




```

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

```

There's no need to declare them as abstract in Optional

only to repeat the implementation in None



```

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

```

There's no need to declare them as abstract in Optional

only to repeat the implementation in None

Writing the Empty implementation directly in Optional is more compact





Is being compact for the sake of compactness any good?



Is being compact for the sake of compactness any good?

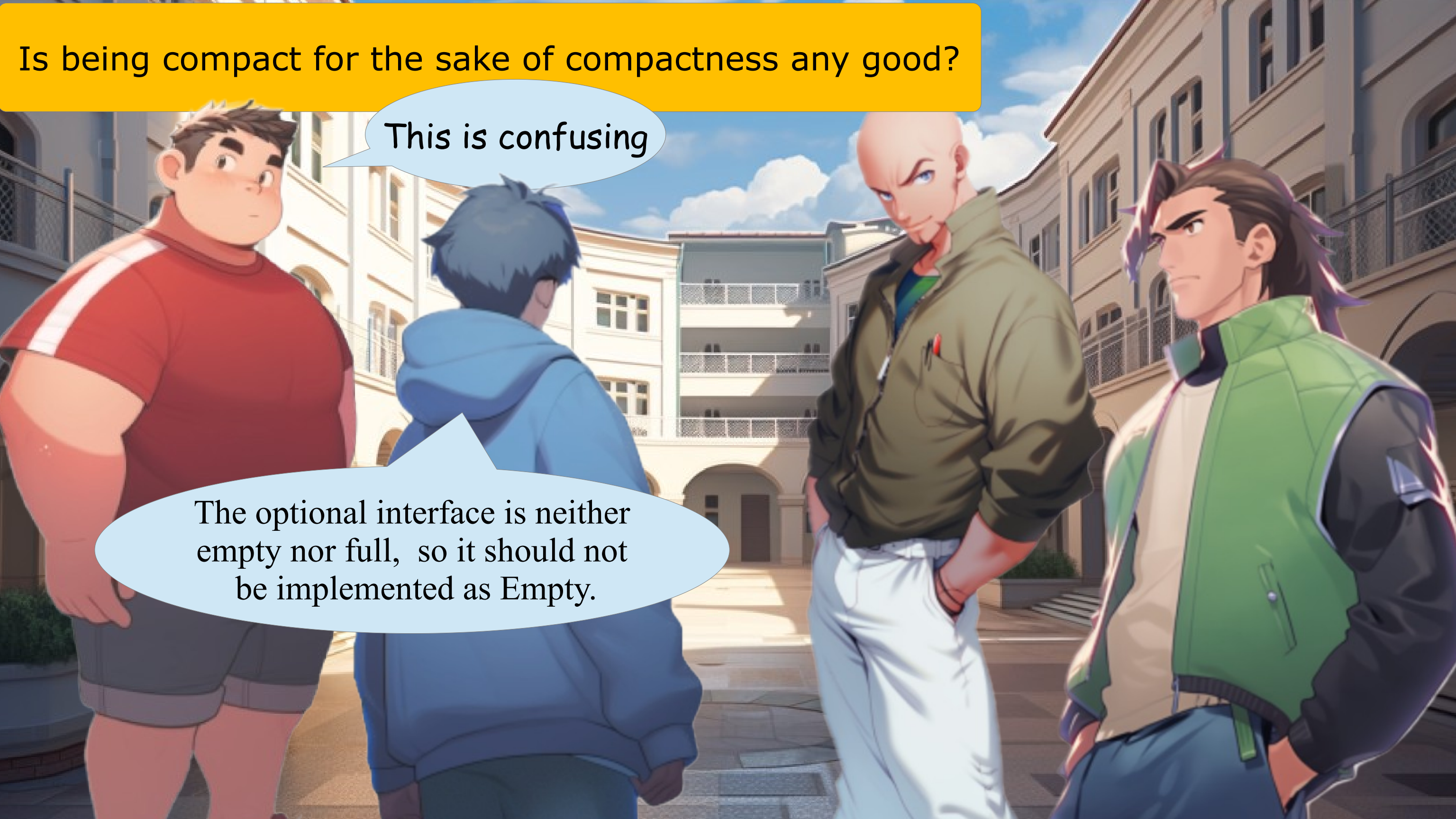
This is confusing



Is being compact for the sake of compactness any good?

This is confusing

The optional interface is neither empty nor full, so it should not be implemented as Empty.



Is being compact for the sake of compactness any good?

This is confusing

The optional interface is neither empty nor full, so it should not be implemented as Empty.


Who cares,
behavior is
what matters





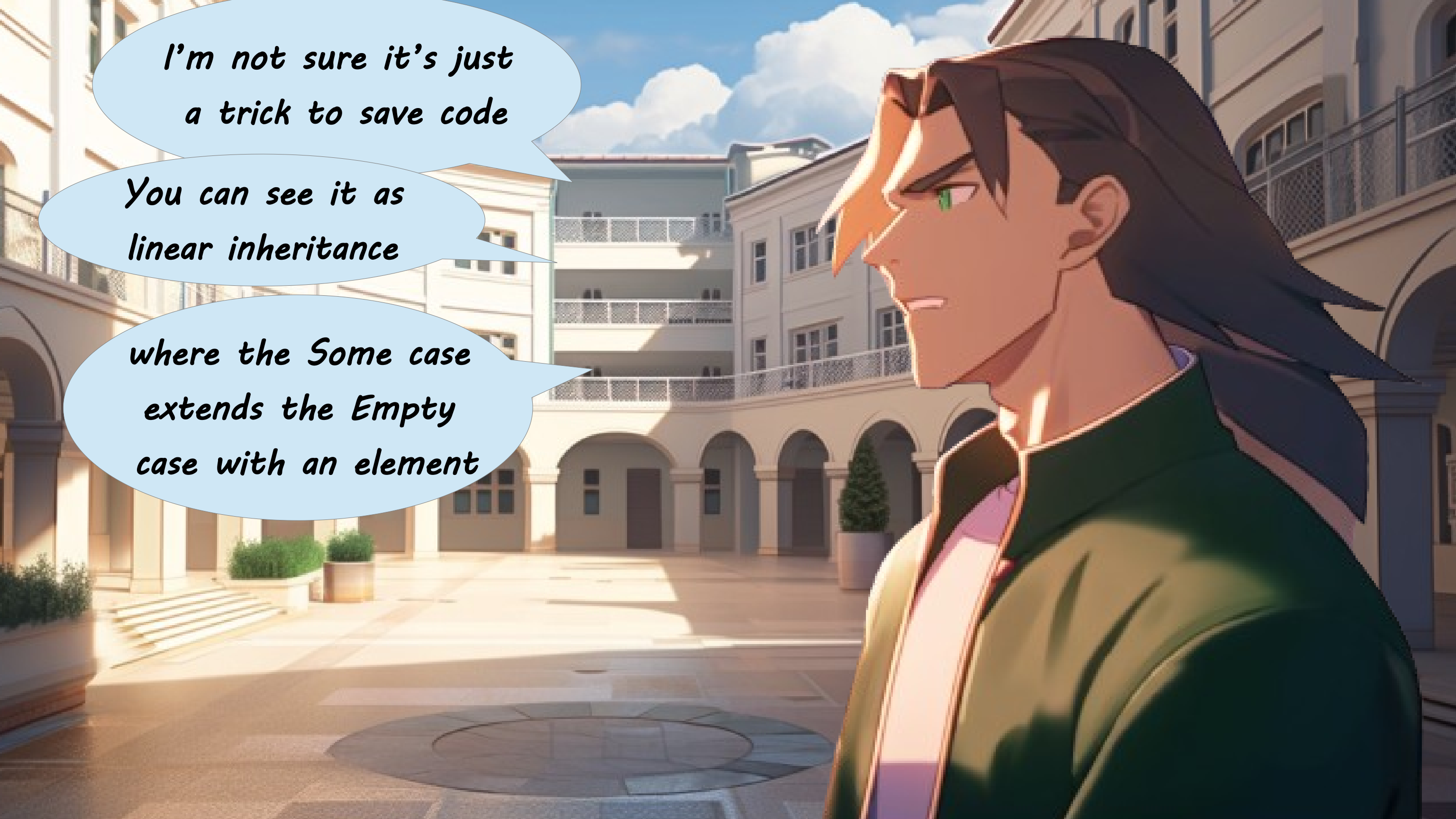
*I'm not sure it's just
a trick to save code*





*I'm not sure it's just
a trick to save code*

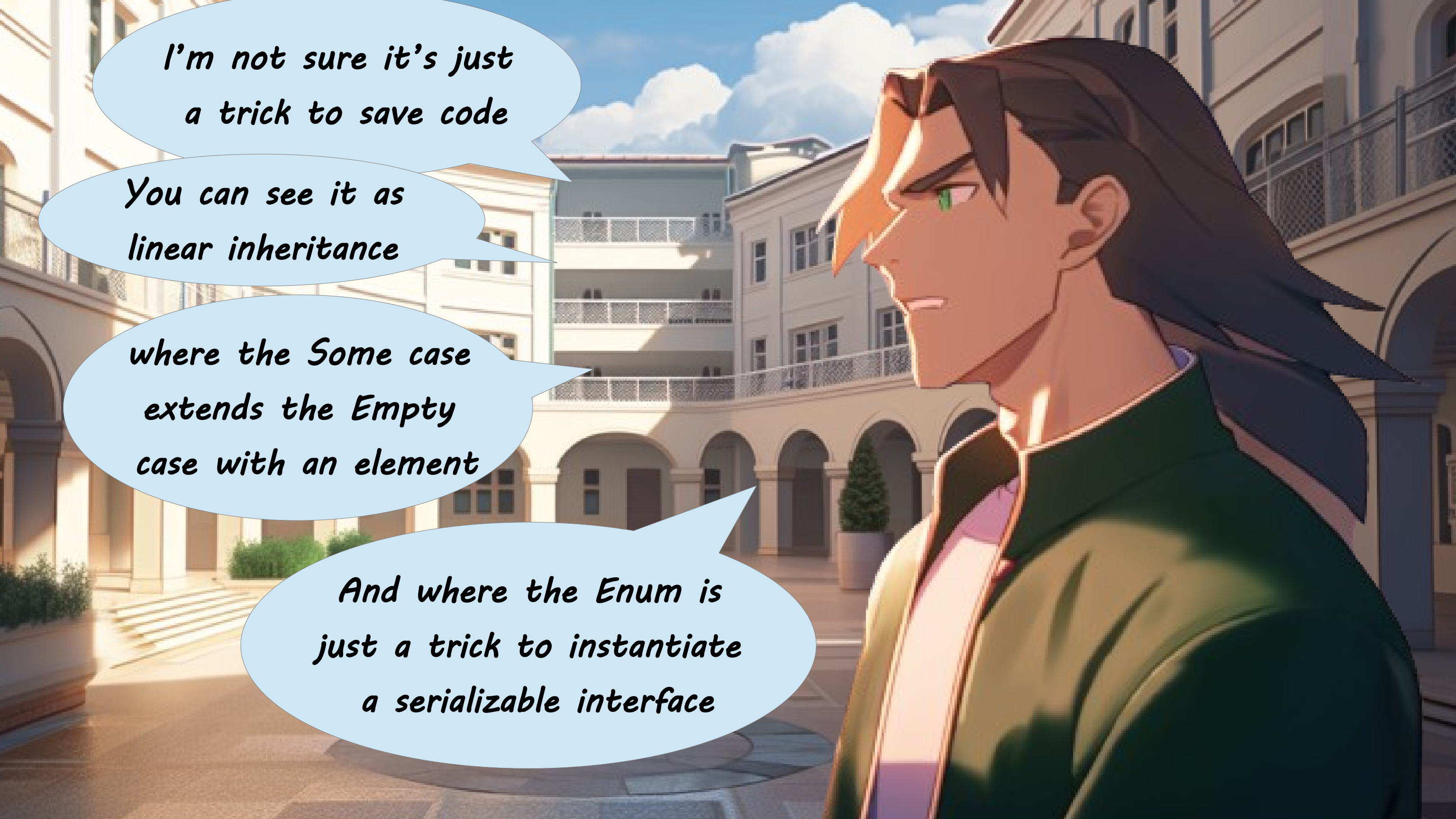
*You can see it as
linear inheritance*



*I'm not sure it's just
a trick to save code*

*You can see it as
linear inheritance*

*where the Some case
extends the Empty
case with an element*



*I'm not sure it's just
a trick to save code*

*You can see it as
linear inheritance*

*where the Some case
extends the Empty
case with an element*

*And where the Enum is
just a trick to instantiate
a serializable interface*



Interfaces have no state





Interfaces have no state

*The Empty enum
also has no state*



Interfaces have no state

*The Empty enum
also has no state*

*Without the serialization issues,
Pupon would have probably used
an empty anonymous inner class*



Interfaces have no state

*The Empty enum
also has no state*

*Without the serialization issues,
Pupon would have probably used
an empty anonymous inner class*

```
static <E> Optional<E> empty() { return new Optional<>(){}; }
```



Interfaces have no state

*The Empty enum
also has no state*

*Without the serialization issues,
Pupon would have probably used
an empty anonymous inner class*

*And you wouldn't be
complaining so much
in that case*

```
static <E> Optional<E> empty() { return new Optional<>(){}; }
```



```

public sealed interface Optional<T> extends Serializable permits Empty, 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);
    }
    default T orElseGet(Supplier<T> s){ return s.get(); }
    default Optional<T> or(Supplier<Optional<T>> s){ return Objects.requireNonNull(s.get()); }
    default <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
    default Optional<T> filter(Predicate<T> p){ return map(e->p.test(e) ? e : null); }
    default <U> Optional<U> flatMap(Function<T, Optional<U>> m){ return Optional.empty(); }
}

```

```

enum Empty implements Optional<T>{ Instance; }

```


```

record Some<T> implements Optional<T>{
    public T get(){ return get; }
    public Optional<T> or(Supplier<Optional<T>> s){ return this; }
    public Optional<U> map(Function<T, Optional<U>> m){
        return Optional.empty();
    }
    public Optional<U> flatMap(Function<T, Optional<U>> m){
        return Objects.requireNonNull(m.apply(get));
    }
}

```

*Should they discuss filter too?
May be not?*





And it is now time to complete
the first year ending ceremony




First, if you are
one of the 58 students



First, if you are
one of the 58 students

who completed at
least one question





First, if you are
one of the 58 students

who completed at
least one question

please form a line in
front of Pupon and Pumpkin



**Dear students, as we reach
the end of this phase at UPU,
it's clear that our paths
diverge here**





**Dear students, as we reach
the end of this phase at UPU,
it's clear that our paths
diverge here**

**We cannot let you go
forward into UPU**



**Dear students, as we reach
the end of this phase at UPU,
it's clear that our paths
diverge here**

**We cannot let you go
forward into UPU**

However



**Dear students, as we reach
the end of this phase at UPU,
it's clear that our paths
diverge here**

**We cannot let you go
forward into UPU**

However

**your skills haven't
gone unnoticed**





**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**





**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**

**Each of you will receive two
recommendation letters.**



**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**

**Each of you will receive two
recommendation letters.**



**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**

**Each of you will receive two
recommendation letters.**

One for MIT
and one for Oxford

An illustration featuring a man on the left and a woman on the right. The man is bald, wearing glasses, a white shirt, a striped tie, and a dark jacket. The woman has short, wavy brown hair, wears glasses, an orange top, a brown blazer, and a necklace. They are standing in a garden with pink flowers and a stone wall. Four light blue speech bubbles are positioned between them, containing text. The background is a stylized, low-poly illustration of a garden scene.

**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**

**Each of you will receive two
recommendation letters.**

One for MIT
and one for Oxford

Consider them your gateway to the
best of what the regular world offers



**As a nod to your achievements in
this exceptional place, we're offering something
rather... conventional**

**Each of you will receive two
recommendation letters.**

One for MIT
and one for Oxford

Consider them your gateway to the
best of what the regular world offers

Do with them what you will




Yes, they might not
be good enough for UPU



Yes, they might not
be good enough for UPU

but they'll definitely thrive in
normal universities like those






Yes, they might not
be good enough for UPU
but they'll definitely thrive in
normal universities like those

What...
What?!?





Wait, does this mean that if we failed one question, we could've then moved on to Oxford?



Yes



Yes

basically they've lost a year
but they'll get a good push forward
for their future anyway



Yes

basically they've lost a year
but they'll get a good push forward
for their future anyway

They discussed this
the week you were sick...
sorry we forgot to tell you





Yes

basically they've lost a year
but they'll get a good push forward
for their future anyway

They discussed this
the week you were sick...
sorry we forgot to tell you

Guess it doesn't
matter much now!