
Easy, right?


Easy, right?
What?
She can explain it
better the me!


Nah, something is still wrong!


Nah, something is still wrong!
What do you mean?


Nah, something is still wrong!
What do you mean?
I see how the code reuse works,
but the subtyping is now a mess!


I can use
CabOver no problem


I can use
CabOver no problem
CabOver<Pepsi>


I can use
CabOver no problem
CabOver<Pepsi>
But, what if I want to
use Just Truck?


I can use
CabOver no problem
CabOver<Pepsi>
But, what if I want to
use Just Truck?
Truck<...,Pepsi>


I can use
CabOver no problem
CabOver<Pepsi>
But, what if I want to
use Just Truck?
Truck<...,Pepsi>
What is ‘Self’ there?


Using Truck of CabOver would not make sense
?


Using Truck of CabOver would not make sense
Truck<CabOver<Pepsi>,Pepsi>


Using Truck of CabOver would not make sense
Truck<CabOver<Pepsi>,Pepsi>
?
It would compile, but it would be a worst way to write CabOver.


Using Truck of CabOver would not make sense
Truck<CabOver<Pepsi>,Pepsi>
?
It would compile, but it would be a worst way to write CabOver.
CabOver<Pepsi>


But, what is the alternative?


But, what is the alternative?
An infinitely long type?


But, what is the alternative?
An infinitely long type?
Truck< Truck<Truck<Truck<..>..>, Pepsi >


Why not just using the wildcard?


Why not just using the wildcard?
Truck<?, Pepsi >


Wildcards? Give me a break!


Wildcards? Give me a break!
Truck<?, Pepsi >


Wildcards? Give me a break!
Truck<?, Pepsi >
With the wildcard, the method ‘driveToward’ would just return an Object.


Wildcards? Give me a break!
Truck<?, Pepsi >
With the wildcard, the method ‘driveToward’ would just return an Object.
Truck<?, Pepsi > truck = ...;
Object pointless = truck.driveToward(..);
//nothing else would compile


Hey Tommy, check out the Pumpkin notes!


Hey Tommy, check out the Pumpkin notes!
Maybe there is an answer to this


Hey Tommy, check out the Pumpkin notes!
Maybe there is an answer to this
Great idea!

```java
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

Yes, according to the notes that was a known problem

```java
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

Yes, according to the notes that was a known problem

Pumpkin proved a version 2 of the code with the solution!


Any idea how this new code works?!


Any idea how this new code works?!
Nope!


Come on Dany,
Focus again,
you can do it!

```
interface Truck<T>{




}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{




}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{


}
```

```java
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{









}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{


}
```

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{


}
```

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

Here we have an additional concept: MovingTruck

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

Here we have an additional concept: MovingTruck

MovingTruck is designed to focus on code reuse,

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

Here we have an additional concept: MovingTruck

MovingTruck is designed to focus on code reuse,
while Truck is designed to focus on subtyping!


We would not use
MovingTruck as a type


We would not use MovingTruck as a type
We would only extend it!

Pumpkin code

```
interface Truck<T>{

}

interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{

}
```

T

Self

T

T

Pumpkin code

```
interface Truck<T>{

}

interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{

}
```

T

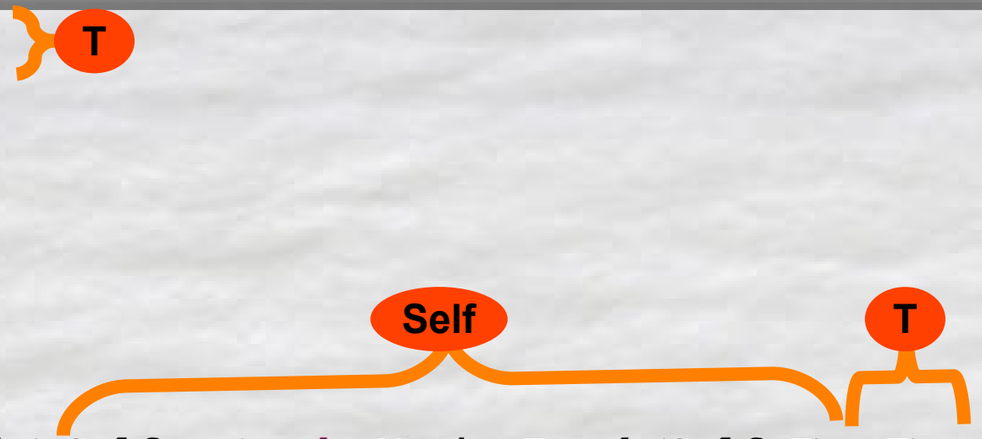
Self

T

T

As you can see, MovingTruck extends the simpler Truck type.

```
interface Truck<T>{

}

interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{

  }
}

record CabOver<T>(                                    )
    implements MovingTruck< CabOver<T>, T >{

}
```



As you can see, MovingTruck extends the simpler Truck type.

In this way both Truck and CabOver can have a single convenient generic argument,

```
interface Truck<T>{

}

interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{

  }
}

record CabOver<T>(                                    )
    implements MovingTruck< CabOver<T>, T >{

}
```

As you can see, MovingTruck extends the simpler Truck type.

In this way both Truck and CabOver can have a single convenient generic argument,
and we can hide the second argument in MovingTruck,
that we are never going to use directly.

```
interface Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}


interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{
  public CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
  public int aerodynamics(){ ...; }
}
```

T

Self

T

T

Self

T

Pumpkin code

As you can see, MovingTruck extends the simpler Truck type.

In this way both Truck and CabOver can have a single convenient generic argument,
and we can hide the second argument in MovingTruck,
that we are never going to use directly.

```
Truck<T> //Truck has one generic argument
```

```
Truck<T> //Truck has one generic argument

MovingTruck< Self ..., T>//MovingTruck has two generic argument
    extends Truck<T>
//extends a definition with one generic argument
```

```
Truck<T> //Truck has one generic argument

MovingTruck< Self ..., T>//MovingTruck has two generic argument
    extends Truck<T>
//extends a definition with one generic argument

CabOver<T>(..)//MovingTruck has one generic argument
    implements MovingTruck< CabOver<T>, T >
//implements a definition with two generic argument
```

```
Truck<T> //Truck has one generic argument

MovingTruck< Self ..., T>//MovingTruck has two generic argument
    extends Truck<T>
//extends a definition with one generic argument

CabOver<T>(..)//MovingTruck has one generic argument
    implements MovingTruck< CabOver<T>, T >
//implements a definition with two generic argument
```

```
Truck<T> //Truck has one generic argument

MovingTruck< Self ..., T>//MovingTruck has two generic argument
    extends Truck<T>
//extends a definition with one generic argument

CabOver<T>(..)//MovingTruck has one generic argument
    implements MovingTruck< CabOver<T>, T >
//implements a definition with two generic argument
```

```
interface Truck<T>{
  ...
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

```
interface Truck<T>{
   ...
   Truck<T> withLocation(Point location);
   Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
     extends Truck<T>{
   Self withLocation(Point location);
   default Self driveToward(Point dest, int duration){
     ... Point wayPoint=...;
     return withLocation(wayPoint);
   }
}
```

```
interface Truck<T>{
   ...
   Truck<T> withLocation(Point location);
   Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
     extends Truck<T>{
   Self withLocation(Point location);
   default Self driveToward(Point dest, int duration){
     ... Point wayPoint=...;
     return withLocation(wayPoint);
   }
}
```

Even after removing all the non crucial code, the interplay between Truck and MovingTruck is still challenging to follow!

```
interface Truck<T>{
   ...
   Truck<T> withLocation(Point location);
   Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
     extends Truck<T>{
   Self withLocation(Point location);
   default Self driveToward(Point dest, int duration){
     ... Point wayPoint=...;
     return withLocation(wayPoint);
   }
}
```

Even after removing all the non crucial code, the interplay between Truck and MovingTruck is still challenging to follow!

Methods ‘withLocation’ and ‘driveToward’ are present in both.
But… they have a different return type!

```
interface Truck<T>{
   ...
   Truck<T> withLocation(Point location);
   Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
     extends Truck<T>{
   Self withLocation(Point location);
   default Self driveToward(Point dest, int duration){
     ... Point wayPoint=...;
     return withLocation(wayPoint);
   }
}
```

Even after removing all the non crucial code, the interplay between Truck and MovingTruck is still challenging to follow!

Methods ‘withLocation’ and ‘driveToward’ are present in both.
But… they have a different return type!
- In Truck they just return Truck of T.
- In MovingTruck they return Self!

```
interface Truck<T>{
   ...
   Truck<T> withLocation(Point location);
   Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
     extends Truck<T>{
   Self withLocation(Point location);
   default Self driveToward(Point dest, int duration){
     ... Point wayPoint=...;
     return withLocation(wayPoint);
   }
}
```

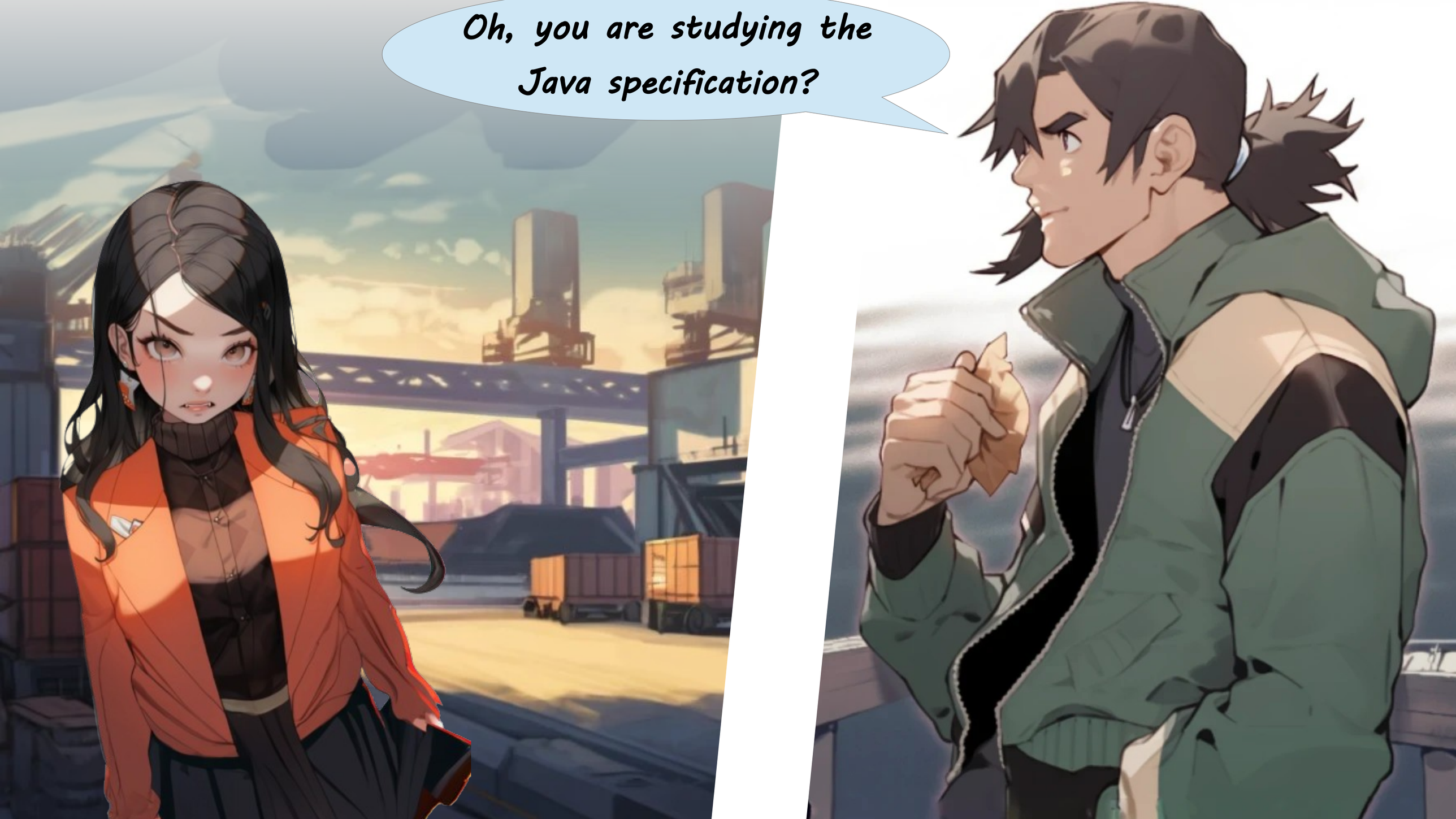
Even after removing all the non crucial code, the interplay between Truck and MovingTruck is still challenging to follow!

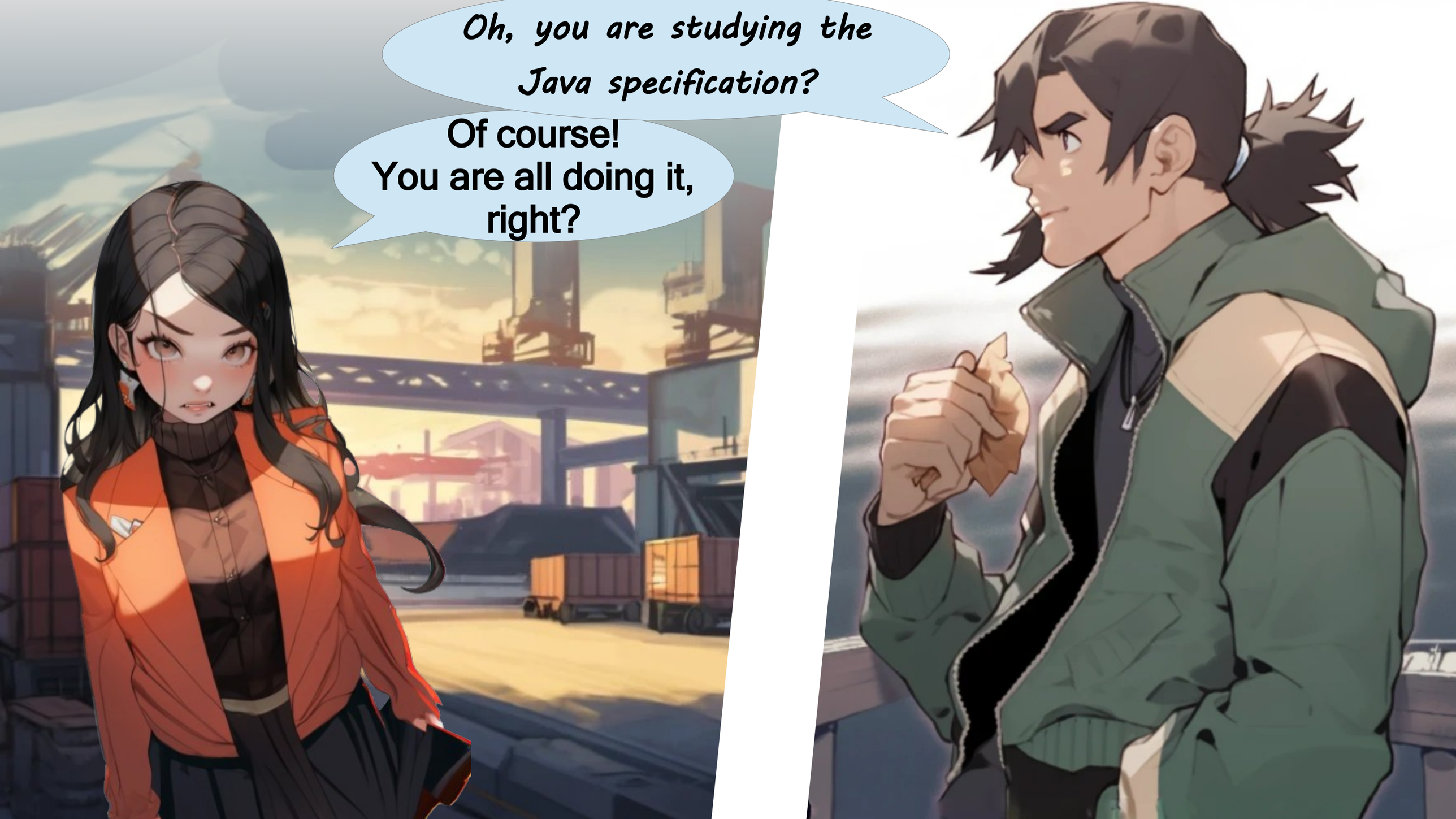
Methods ‘withLocation’ and ‘driveToward’ are present in both.
But… they have a different return type!
- In Truck they just return Truck of T.
- In MovingTruck they return Self!
How can this work?

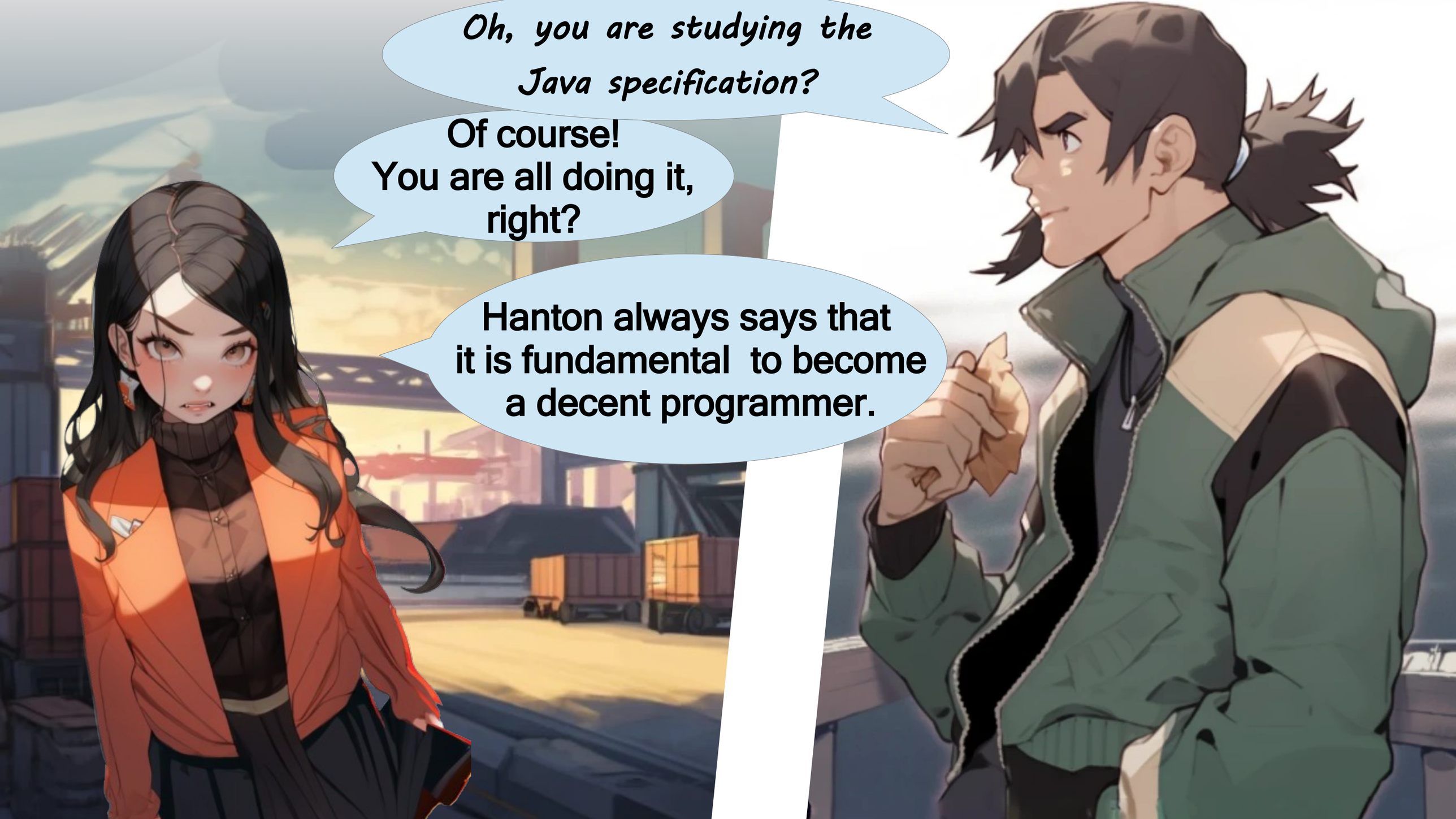

Well...

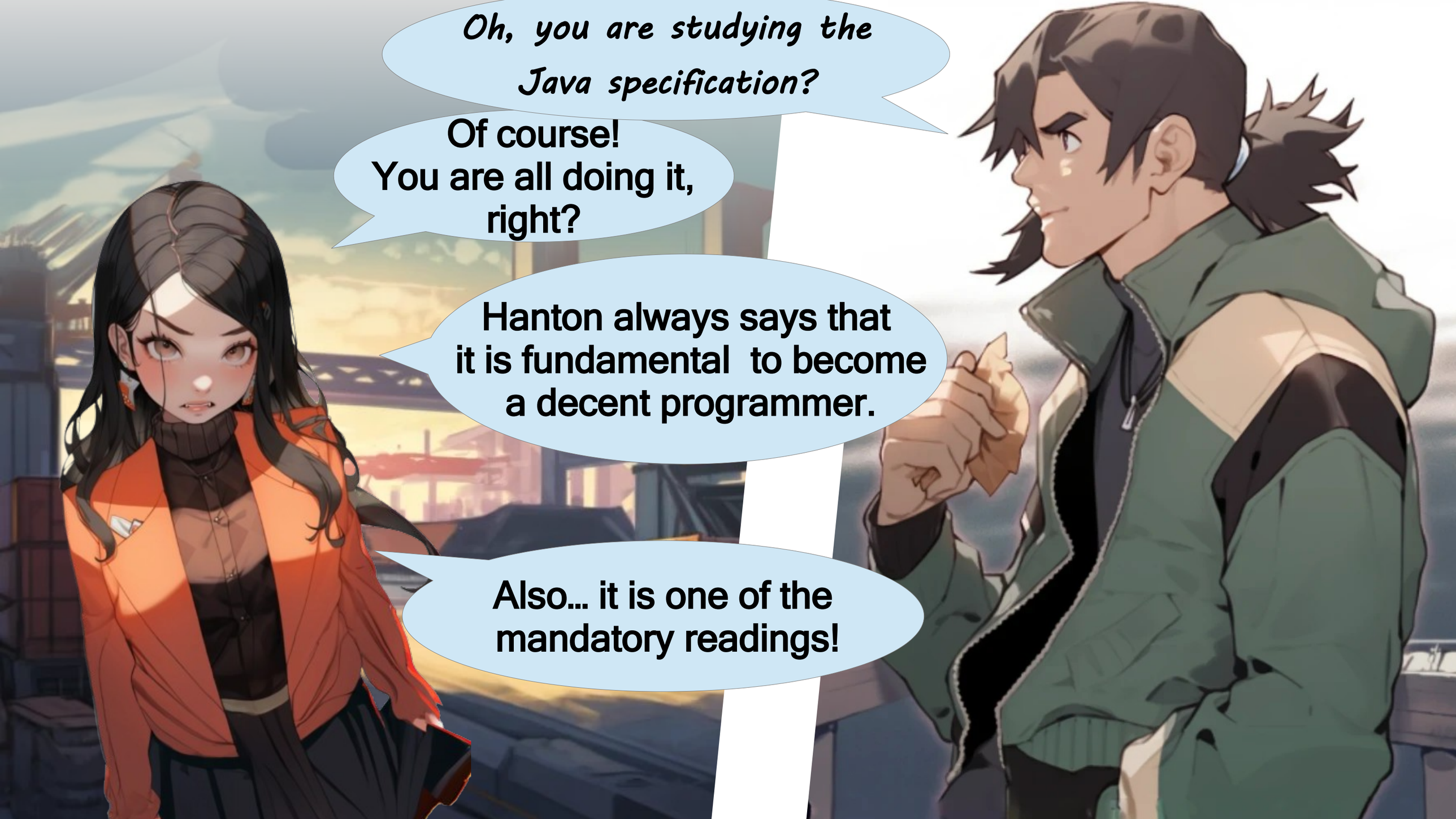

Well...
According to the Java specification, that is called ‘Method return type refinement’


Well...
According to the Java specification, that is called ‘Method return type refinement’ and it is allowed if the new type is a subtype of the old one


Oh, you are studying the Java specification?


Oh, you are studying the Java specification?
Of course! You are all doing it, right?


Oh, you are studying the Java specification?
Of course! You are all doing it, right?
Hanton always says that it is fundamental to become a decent programmer.


Oh, you are studying the Java specification?
Of course! You are all doing it, right?
Hanton always says that it is fundamental to become a decent programmer.
Also... it is one of the mandatory readings!

```
interface Truck<T>{
  ...
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{...}
```

```
interface Truck<T>{
  ...
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{...}
```

Ok then!
Method type refinement is indeed accepted here since we made a more complex declaration for Self

```
interface Truck<T>{
  ...
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{...}
```

Ok then!
Method type refinement is indeed accepted here since we made a more complex declaration for Self

```
interface Truck<T>{
  ...
  Truck<T> withLocation(Point location);
  Truck<T> driveToward(Point dest, int duration);
}
interface MovingTruck< Self extends MovingTruck<Self,T>, T>
    extends Truck<T>{
  Self withLocation(Point location);
  default Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
record CabOver<T>(Point location, CargoC<List<T>> cargo)
    implements MovingTruck< CabOver<T>, T >{...}
```

Ok then!
Method type refinement is indeed accepted here since we made a more complex declaration for Self

Actually, it is quite interesting.
Self needs to be a kind of MovingTruck of Self and T

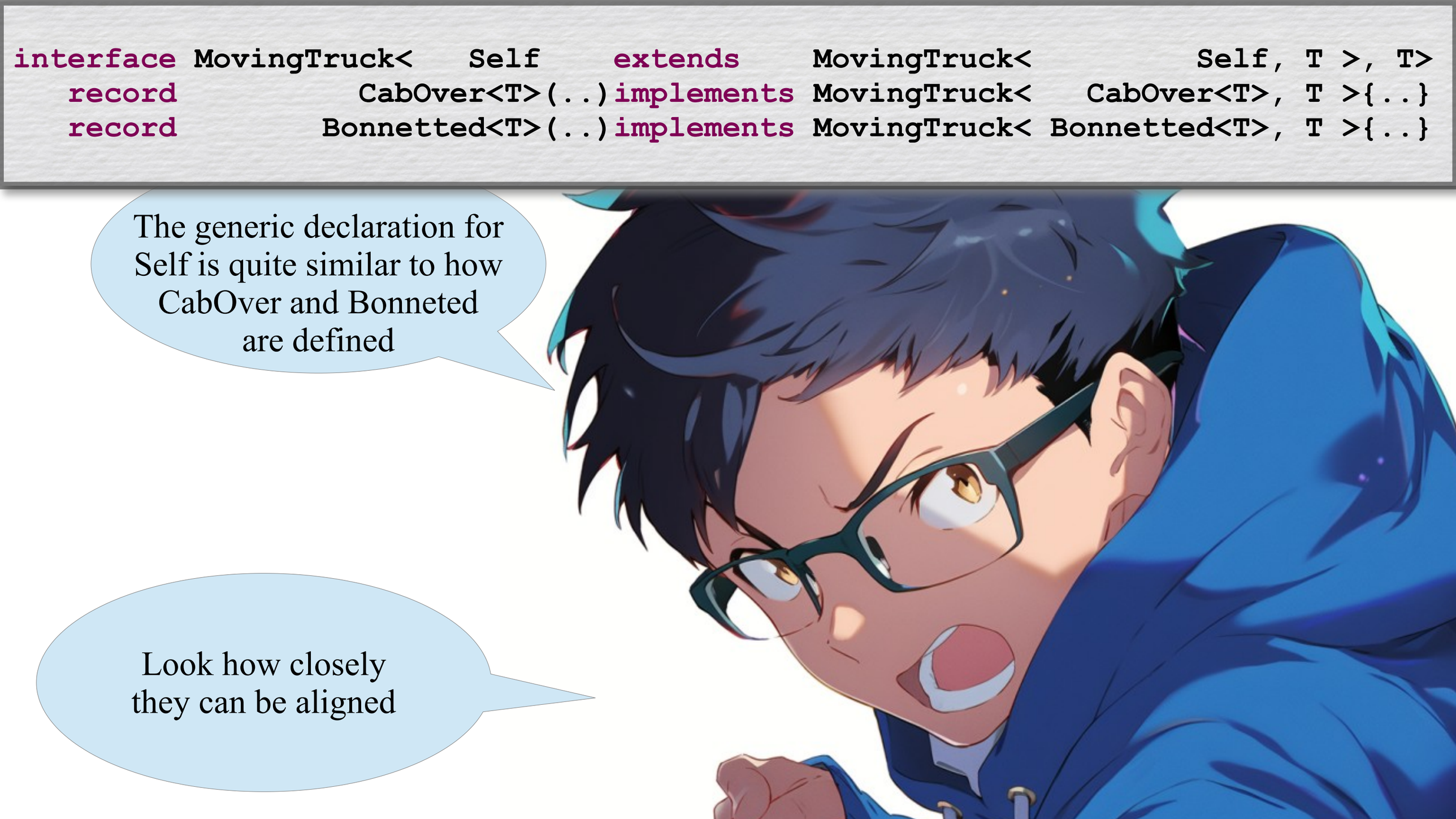
```
interface MovingTruck<    Self     extends     MovingTruck<          Self, T >, T>
```

```
interface MovingTruck<    Self     extends     MovingTruck<          Self, T >, T>
   record             CabOver<T>(..)implements MovingTruck<   CabOver<T>, T >{..}
```

```
interface MovingTruck<   Self     extends     MovingTruck<          Self, T >, T>
   record          CabOver<T>(..)implements MovingTruck<   CabOver<T>, T >{..}
   record        Bonnetted<T>(..)implements MovingTruck< Bonnetted<T>, T >{..}
```

```
interface MovingTruck<   Self    extends    MovingTruck<          Self, T >, T>
   record          CabOver<T>(..)implements MovingTruck<   CabOver<T>, T >{..}
   record        Bonnetted<T>(..)implements MovingTruck< Bonnetted<T>, T >{..}
```

```
interface MovingTruck<   Self    extends    MovingTruck<          Self, T >, T>
   record          CabOver<T>(..)implements MovingTruck<   CabOver<T>, T >{..}
   record        Bonnetted<T>(..)implements MovingTruck< Bonnetted<T>, T >{..}
```




So, what version should we use? The Pupon version?


So, what version should we use? The Pupon version?
Or the improved Pumpkin one?


So, what version should we use? The Pupon version?
Or the improved Pumpkin one?
Yes, I'm also quite confused.


So, what version should we use? The Pupon version?
Or the improved Pumpkin one?
Yes, I'm also quite confused. One of the two must be the best one, right?


Not really!


Not really!
Pumpkin version
can be more suited for
parallel programming


Not really!
Pumpkin version
can be more suited for
parallel programming
And for situations where you
do need to keep the
old state around, …


Not really!
Pumpkin version
can be more suited for
parallel programming
And for situations where you
do need to keep the
old state around, …
like supporting an undo
operation or making an AI exploring
potential future states


Pupon version makes sense
if you need to rely on aliasing


Pupon version makes sense if you need to rely on aliasing
For example, if you are storing those trucks …


Pupon version makes sense if you need to rely on aliasing
For example, if you are storing those trucks …
in other long lived data structures,


Pupon version makes sense if you need to rely on aliasing
For example, if you are storing those trucks …
in other long lived data structures, and you want them to be updated all at once when you update the truck itself


Also, knowing UPU,
both versions are going to be needed
for the final exam


Oh, yes, the final exam


Oh, yes, the final exam
It is next month right?


Oh, yes, the final exam
I'm so worried!
It is next month right?


Oh, yes, the final exam
I'm so worried!
I think we should keep studying together!
It is next month right?


Oh, yes, the final exam
I'm so worried!
I think we should keep studying together!
It is next month right?
But only if Tommy stops stealing stuff!


Let’s do it!
We should try to get Hanton
on board too.


Let’s do it!
We should try to get Hanton on board too.
Together we will defeat Pupon and reach the Ivory tower!


Let’s do it!
We should try to get Hanton on board too.
Together we will defeat Pupon and reach the Ivory tower!
Where the second year students live!