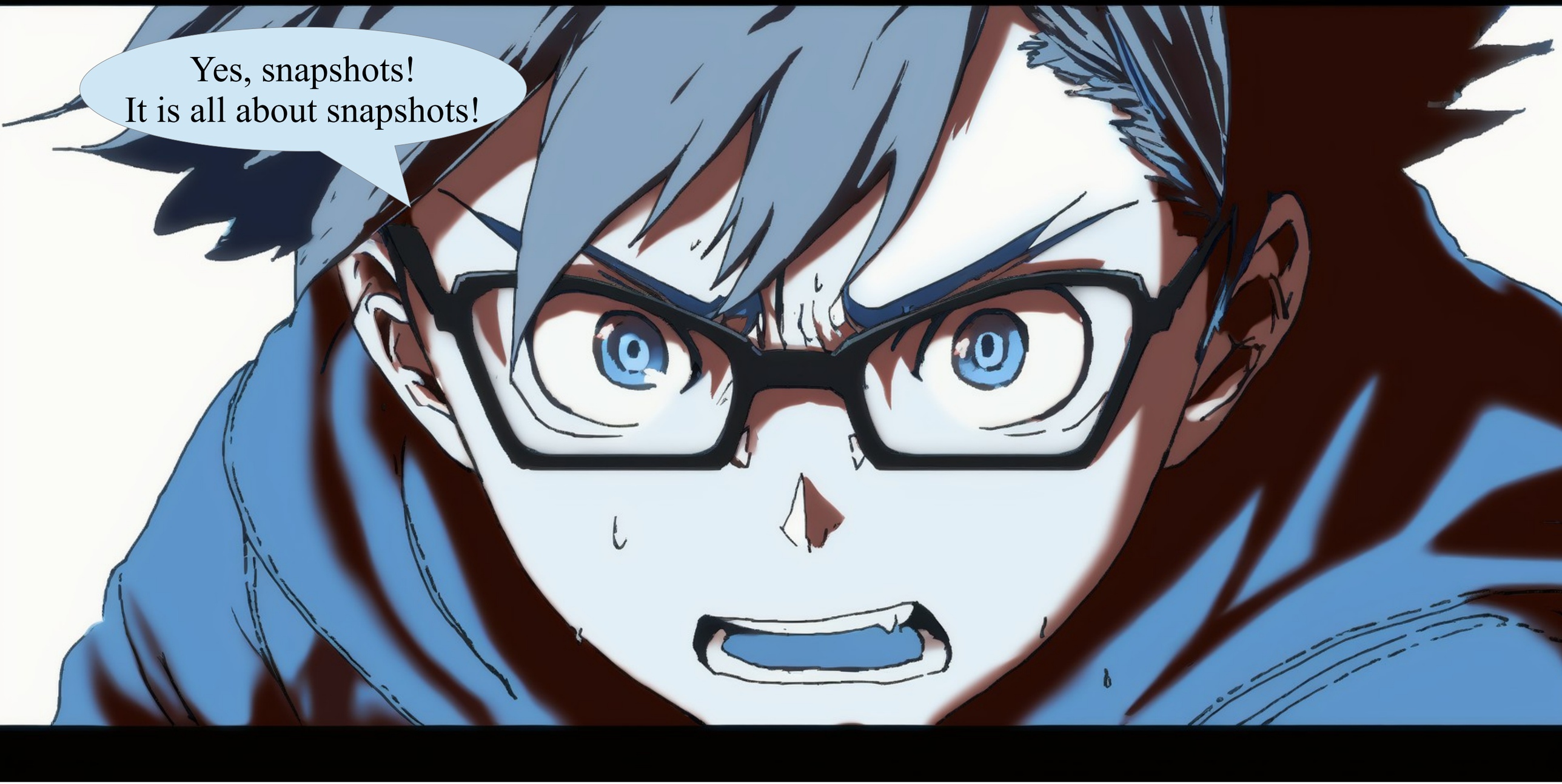

Oh, not again!

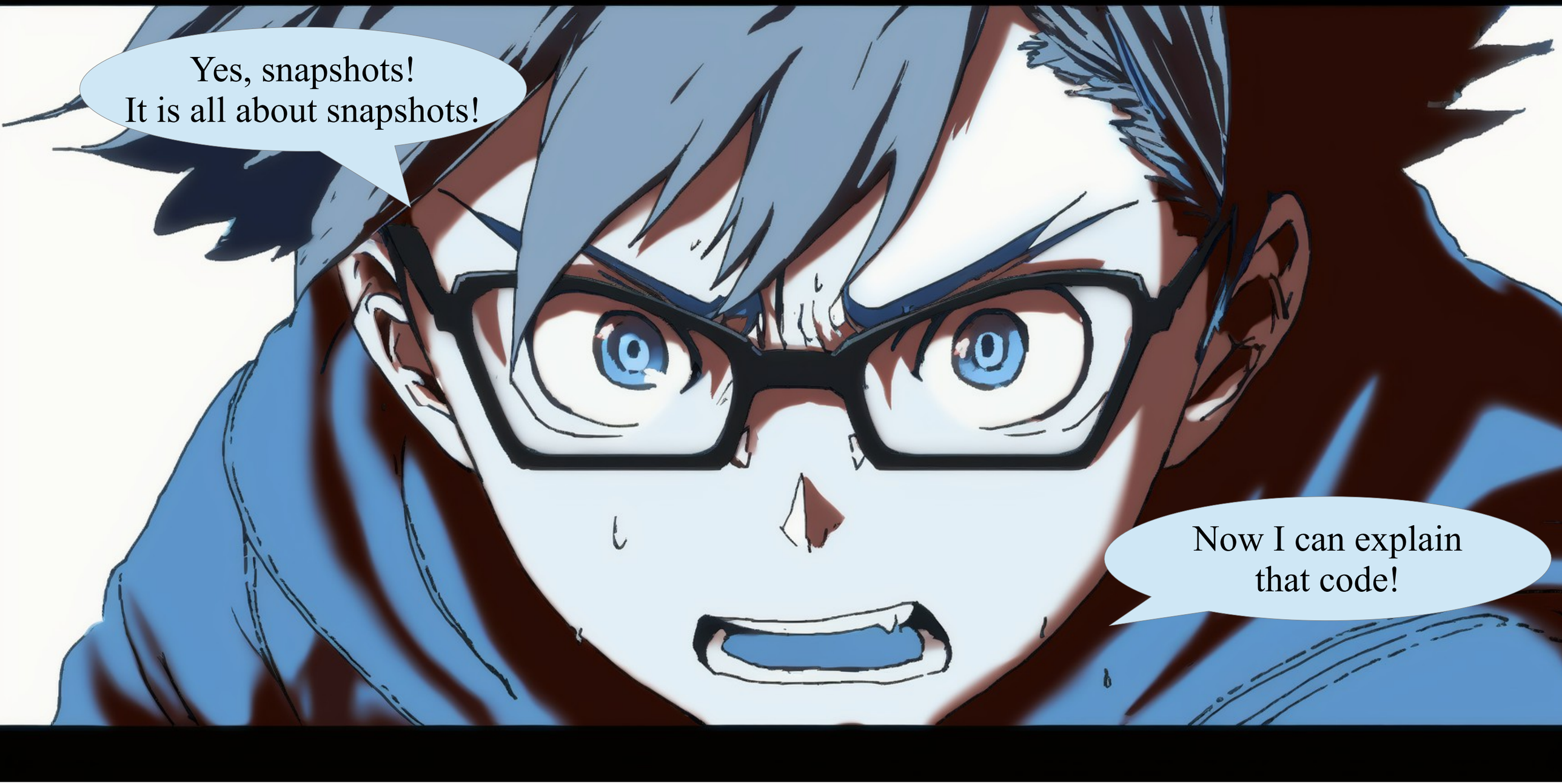

Hey Dany,
are you... ok?
Oh, not again!


Oh, not again!
Hey Dany, are you... ok?
It looks like he is in some sort of trance

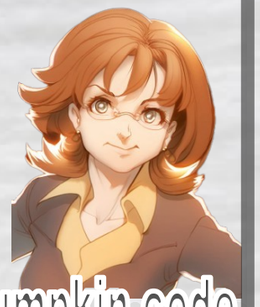

Hey Dany,
are you... ok?
It looks like he is in
some sort of trance
Oh, not again!
He will be ok,
he just panics easily


Oh, not again!
Hey Dany, are you... ok?
It looks like he is in some sort of trance
He will be ok, he just panics easily
Hey Dany, snap out of it!


Yes, snapshots!
It is all about snapshots!


Yes, snapshots!
It is all about snapshots!
Now I can explain
that code!

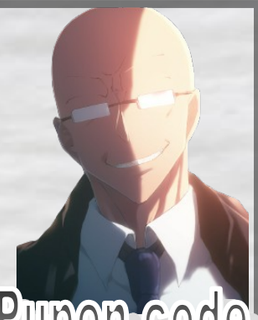
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

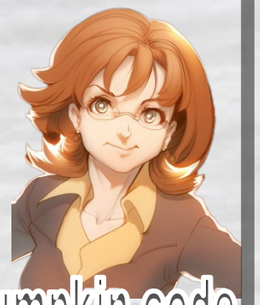
```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```



Pupon code

Pumpkin is using interfaces and records,
while Pupon is using classes and abstract classes.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

```
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

Pupon code

Pumpkin is using interfaces and records,
while Pupon is using classes and abstract classes.
In interfaces the ‘public’ and ‘abstract’ keywords are
implicit, but in classes they are required.

```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

Pupon code

Pumpkin is using interfaces and records,
while Pupon is using classes and abstract classes.
In interfaces the ‘public’ and ‘abstract’ keywords are
implicit, but in classes they are required.

Also, interfaces need this funny ‘default’ keyword to
implement methods.

```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

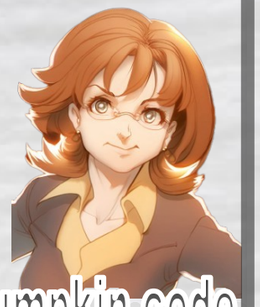

```java
record Bonneted<T>
  (Point location, CargoC<List<T>> cargo) implements Truck<Bonneted<T>,T>{
  public Bonneted<T> withLocation(Point p){ return new Bonneted<>(p,cargo); }
  public int aerodynamics(){ ... }
}
record CabOver<T>
 (Point location, CargoC<List<T>> cargo) implements Truck<CabOver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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

Pumpkin is using interfaces and records,
while Pupon is using classes and abstract classes.
In interfaces the ‘public’ and ‘abstract’ keywords are
implicit, but in classes they are required.

Also, interfaces need this funny ‘default’ keyword to
implement methods.
Keywords ‘implements’ and ‘extends’ means the
same thing but they need to be used in specific locations

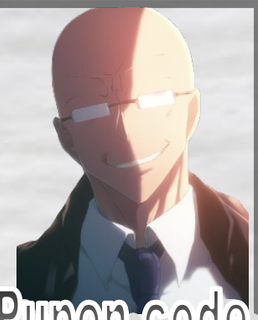
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

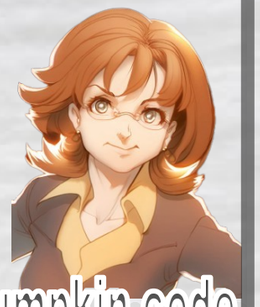
```java
record Bonneted<T>
  (Point location, CargoC<List<T>> cargo) implements Truck<Bonneted<T>,T>{
  public Bonneted<T> withLocation(Point p){ return new Bonneted<>(p,cargo); }
  public int aerodynamics(){ ... }
}
record CabOver<T>
 (Point location, CargoC<List<T>> cargo) implements Truck<CabOver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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```



Pupon code

```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

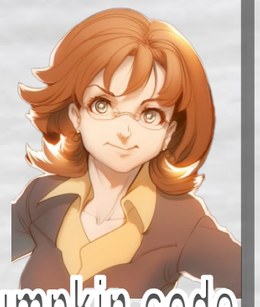
```java
record Bonneted<T>
  (Point location, CargoC<List<T>> cargo) implements Truck<Bonneted<T>,T>{
  public Bonneted<T> withLocation(Point p){ return new Bonneted<>(p,cargo); }
  public int aerodynamics(){ ... }
}
record CabOver<T>
 (Point location, CargoC<List<T>> cargo) implements Truck<CabOver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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

Pupon code

The more I learn Java,
the more it looks like a

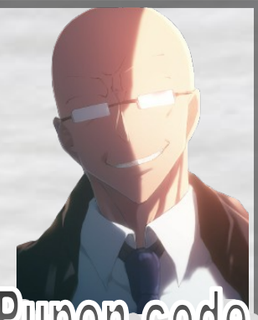
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

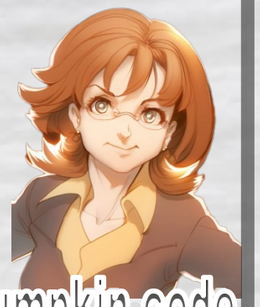
```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```



Pupon code

The more I learn Java,
the more it looks like a
very inconsistent language

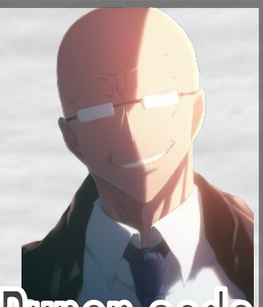
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

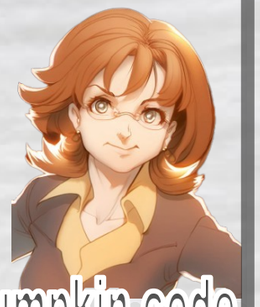


Pupon code

The more I learn Java, the more it looks like a very inconsistent language

Yes it is. For our own understanding, lets remove

```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

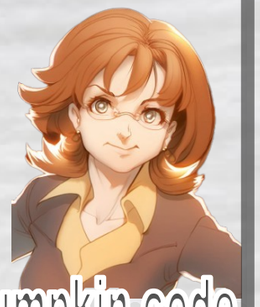
```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

Pupon code

The more I learn Java, the more it looks like a very inconsistent language

Yes it is. For our own understanding, lets remove all the ‘public’, ‘abstract’ and ‘default’ keyword,

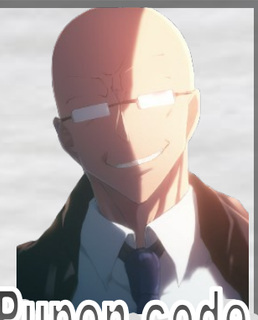
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```

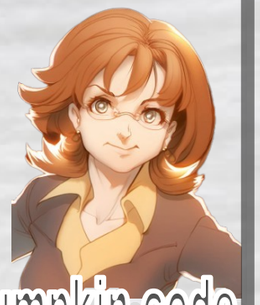


Pupon code

The more I learn Java, the more it looks like a very inconsistent language

Yes it is. For our own understanding, lets remove all the ‘public’, ‘abstract’ and ‘default’ keyword, and replace ‘implements’ and ‘extends’ with just ‘:’

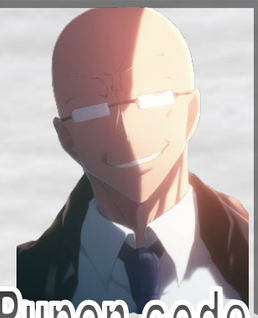
```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  default Self driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```



Pumpkin code

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

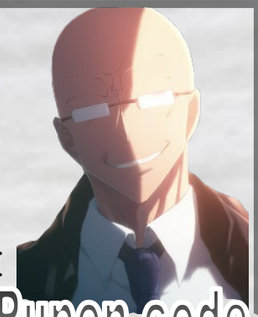
```java
abstract class Truck<T>{
  public abstract int aerodynamics();
  public CargoC<List<T>> cargo(){ return cargo; }
  public Point location(){ return location; }
  public void location(Point p){ location=p; }

  void driveToward(Point destination, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){ location=p; cargo=c; }
}

class Bonneted<T> extends Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}

class CabOver<T> extends Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  public int aerodynamics(){ ... }
}
```



Pupon code


```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

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

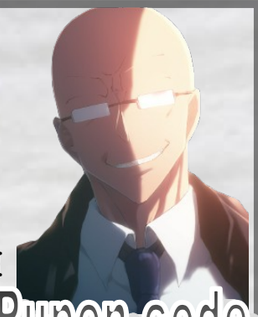



```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

Pumpkin code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

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



Pupon code

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}
```

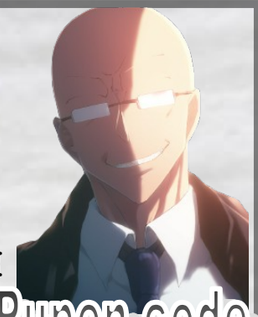
```
class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



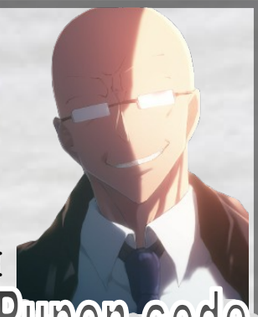
Pupon code

Now we see more similarity between their code.

Pumpkin code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

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



Pupon code

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}
```

```
class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

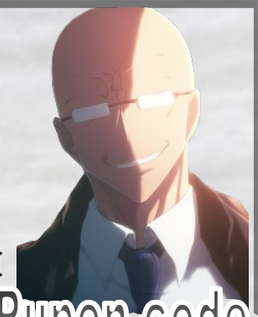
Now we see more similarity between their code.
The most important one is that they both want to reuse the ‘driveToward’ implementation for both kinds of trucks.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

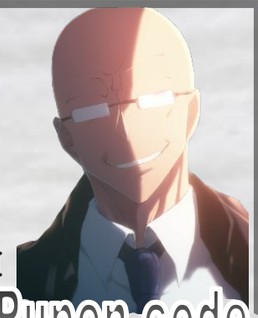
They both have Truck, Bonneted and CabOver concepts, with getters for location and cargo.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

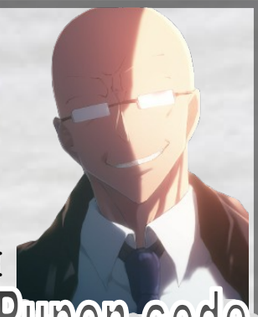


Pupon code

They both have Truck, Bonneted and CabOver concepts,
with getters for location and cargo.
Truck has an abstract ‘aerodynamics’ method
that they implement in Bonneted and CabOver.

Pumpkin code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}

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



Pupon code

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

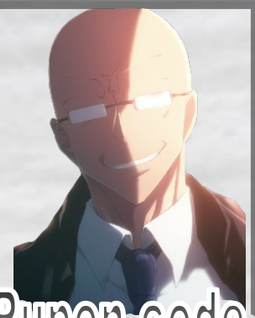
They both have Truck, Bonneted and CabOver concepts,
with getters for location and cargo.
Truck has an abstract ‘aerodynamics’ method
that they implement in Bonneted and CabOver.
Except for the return type and the last line, the ‘driveToward’ method
is identical.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ...
    Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class Bonneted<T> : Truck<T>{
  Bonneted(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

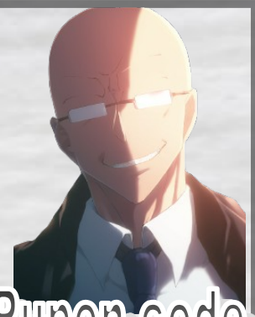
They both have Truck, Bonneted and CabOver concepts,
with getters for location and cargo.
Truck has an abstract ‘aerodynamics’ method
that they implement in Bonneted and CabOver.
Except for the return type and the last line, the ‘driveToward’ method
is identical. Overall, in both approaches the difference between Bonneted and
CabOver is irrelevant, so we can just consider CabOver going forward.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

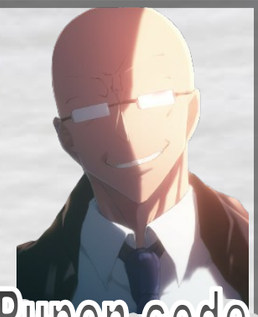
We can now focus on the main conceptual difference:

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

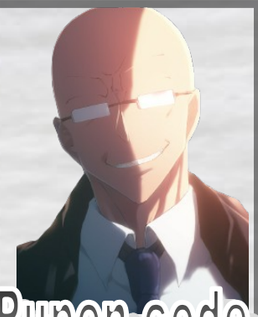
We can now focus on the main conceptual difference:
Pumpkin records have all fields final,
while Pupon classes can update the location field.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

We can now focus on the main conceptual difference:
Pumpkin records have all fields final,
while Pupon classes can update the location field.

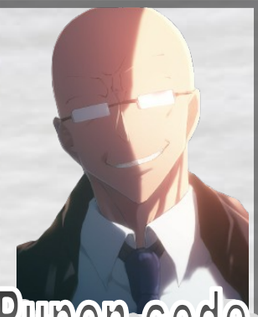
So, while Pupon represent trucks moving by modifying the current object,
Pumpkin uses objects as snapshots, and creates new Truck objects to represent the moved trucks.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }

  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
```



Pupon code

```
class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

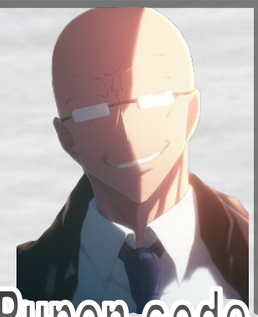
To have usable classes we need to manually define private fields and constructors.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }

  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
```



Pupon code

```
class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```

To have usable classes we need to manually define private fields and constructors.

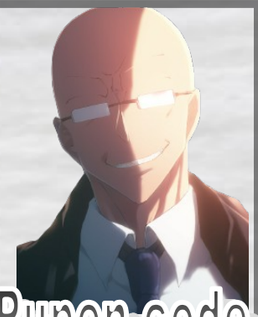
Plus, every derived class must define their own constructor and call the superconstructor explicitly!

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }

  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}
```



Pupon code

```
class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}  //Much more code should go here to make it usable
```

To have usable classes we need to manually define private fields and constructors.

Plus, every derived class must define their own constructor and call the superconstructor explicitly!

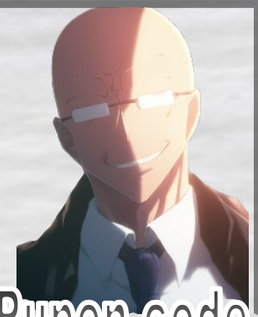
The code of Pupon is just an example. The usable version would be much longer because is should also override equals, hashCode and toString.

```java
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```java
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }

  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}
```



Pupon code

```java
class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}  //Much more code should go here to make it usable
```

To have usable classes we need to manually define private fields and constructors.

Plus, every derived class must define their own constructor and call the superconstructor explicitly!

The code of Pupon is just an example. The usable version would be much longer because is should also override equals, hashCode and toString.
They should be added to CabOver, Bonneted and any other kind of truck we may have, making Pupon code much more unwieldy.


True, Pupon code would be more verbose, but ...


True, Pupon code would be more verbose, but ...
It seams like Pumpkin code would be super slow!

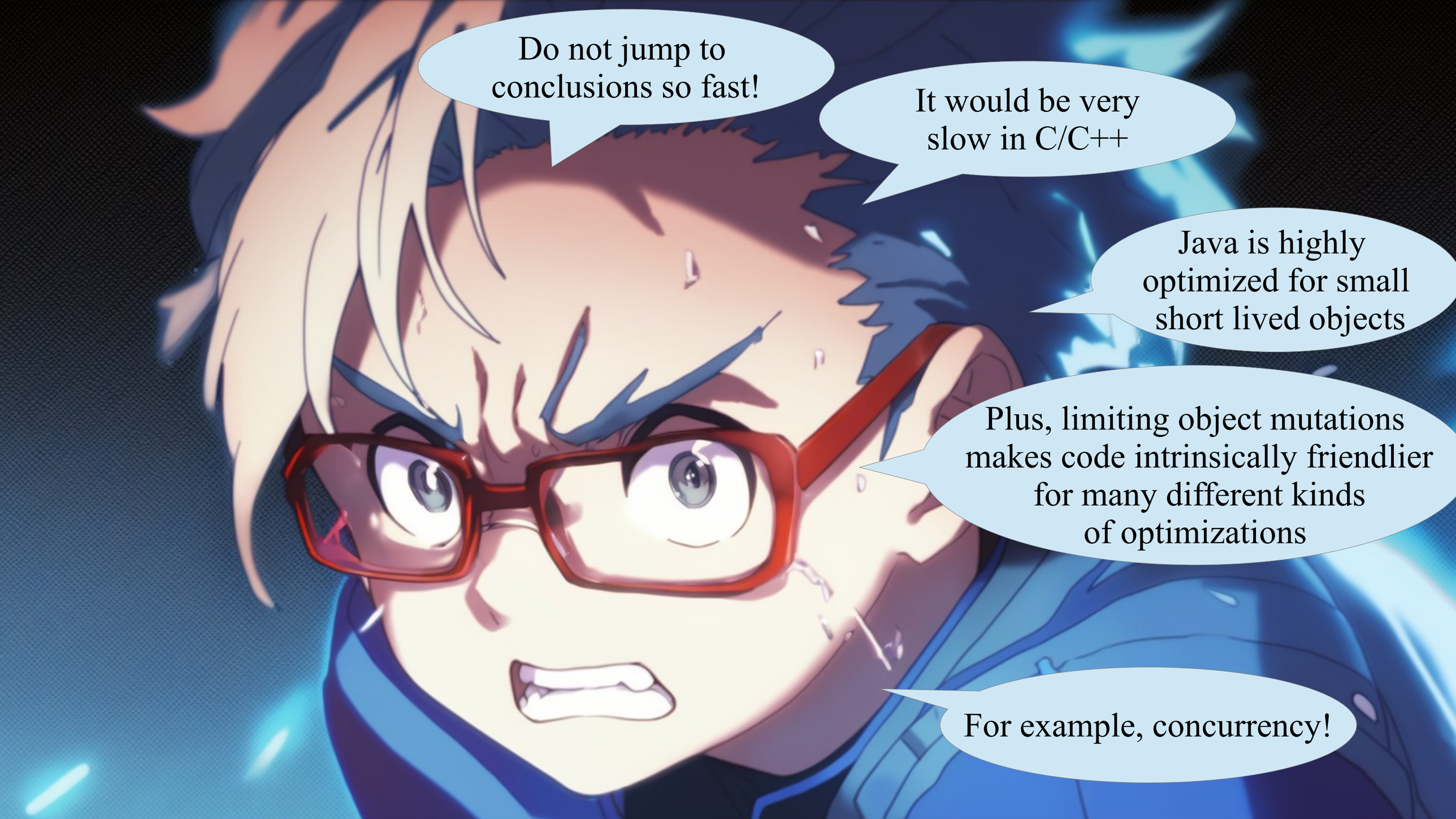

True, Pupon code would be more verbose, but ...
It seams like Pumpkin code would be super slow!
Why would anyone want to code like that?


Do not jump to conclusions so fast!


Do not jump to conclusions so fast!
It would be very slow in C/C++


Do not jump to conclusions so fast!
It would be very slow in C/C++
Java is highly optimized for small short lived objects


Do not jump to conclusions so fast!
It would be very slow in C/C++
Java is highly optimized for small short lived objects
Plus, limiting object mutations makes code intrinsically friendlier for many different kinds of optimizations

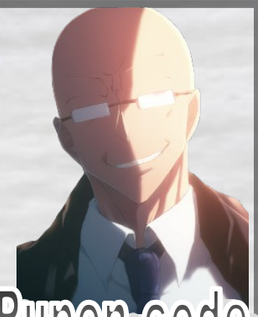

Do not jump to conclusions so fast!
It would be very slow in C/C++
Java is highly optimized for small short lived objects
Plus, limiting object mutations makes code intrinsically friendlier for many different kinds of optimizations
For example, concurrency!

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



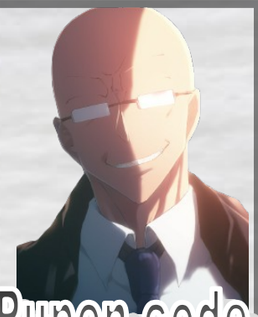
Pupon code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();

  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

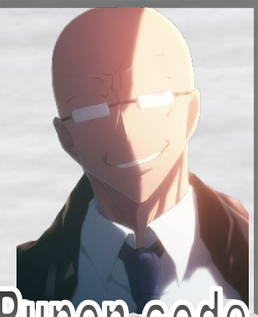
Since Pumpkin has to create new objects, she defines a wither: ‘withLocation’

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();

  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  [illegible] location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

Since Pumpkin has to create new objects, she defines a wither: ‘withLocation’

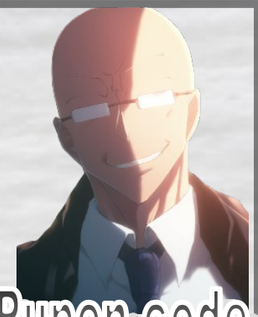
Pupon needs to mutate the current object, so he defines a setter: ‘location’

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();

  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  [illegible] location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

Since Pumpkin has to create new objects, she defines a wither: ‘withLocation’

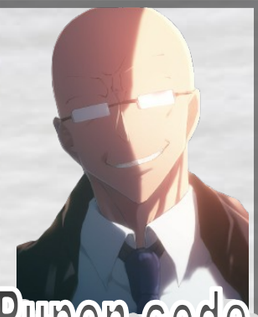
Pupon needs to mutate the current object, so he defines a setter: ‘location’
Pupon signature is easy: the current object is mutated and ‘void’ is returned.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();

  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  [illegible] location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

Since Pumpkin has to create new objects, she defines a wither: ‘withLocation’

Pupon needs to mutate the current object, so he defines a setter: ‘location’
Pupon signature is easy: the current object is mutated and ‘void’ is returned.

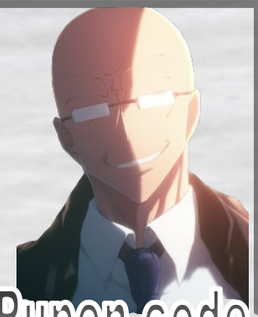
Pumpkin, however is in trouble: as we will see in a moment,
we can not just return ‘Truck’!

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



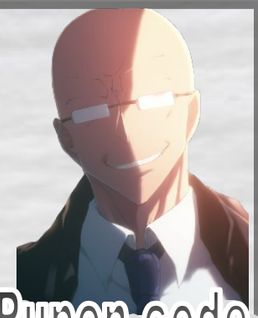
Pupon code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

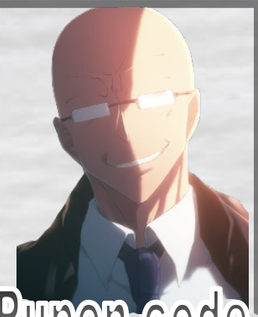
In the crucial method ‘driveToward’, we take a destination point and a max trip duration.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

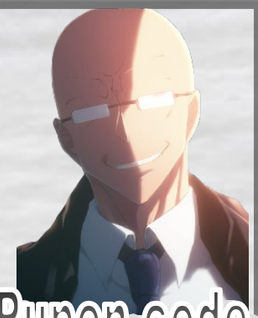
In the crucial method ‘driveToward’, we take a destination point and a max trip duration. We compute a wayPoint, that could be either the destination or some point in between if the destination is too far.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

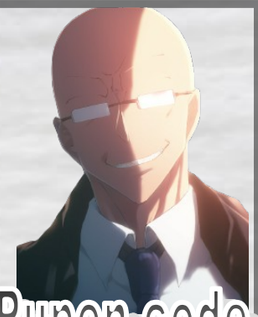
In the crucial method ‘driveToward’, we take a destination point and a max trip duration. We compute a wayPoint, that could be either the destination or some point in between if the destination is too far.
This can take quite a lot of code and so it is omitted in this example.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

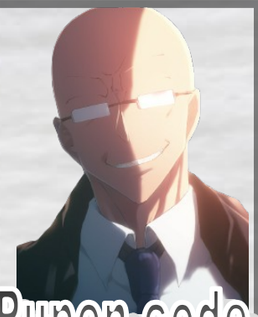
In the crucial method ‘driveToward’, we take a destination point and a max trip duration. We compute a wayPoint, that could be either the destination or some point in between if the destination is too far.
This can take quite a lot of code and so it is omitted in this example.

As you can see, the end of ‘driveToward’ uses either the wither

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    return this.withLocation(wayPoint);
  }
}

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

Pumpkin code

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

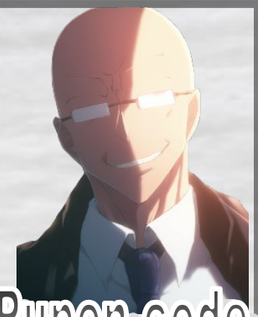
In the crucial method ‘driveToward’, we take a destination point and a max trip duration. We compute a wayPoint, that could be either the destination or some point in between if the destination is too far.
This can take quite a lot of code and so it is omitted in this example.

As you can see, the end of ‘driveToward’ uses either the wither or the setter as a way to represent the result.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    return this.withLocation(wayPoint);
  }
}

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

Pumpkin code

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

In the crucial method ‘driveToward’, we take a destination point and a max trip duration. We compute a wayPoint, that could be either the destination or some point in between if the destination is too far.
This can take quite a lot of code and so it is omitted in this example.

As you can see, the end of ‘driveToward’ uses either
the wither or the setter as a way to represent the result.
So, the return type of ‘driveToward’ must be the same of the return type of the wither


So, why this common return type can not just be 'Truck'?


So, why this common return type can not just be 'Truck'?
We are just moving Trucks, right? We are even writing this code inside of 'Truck'!


It is not so easy, …


We want to preserve the type of our CabOver Truck!
It is not so easy, …


```
...
CabOver<Pepsi> optimus = new CabOver<Pepsi>(...);
optimus = optimus.driveToward(…);
```


```
...
CabOver<Pepsi> optimus = new CabOver<Pepsi>(...);
optimus = optimus.driveToward(…); //if the return type was just ‘Truck’
                                  //we would get a type error here
```

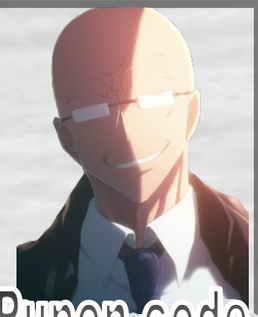

```
...
CabOver<Pepsi> optimus = new CabOver<Pepsi>(...);
optimus = optimus.driveToward(…); //if the return type was just ‘Truck’
                                  //we would get a type error here
```

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



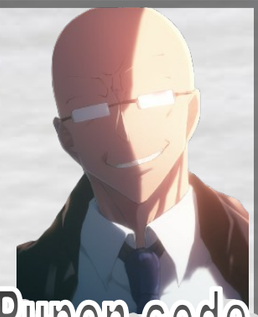
Pupon code

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

And here is why we use an additional generic type argument: Self!

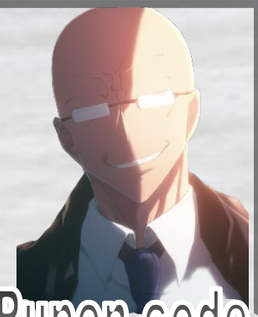
```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

Self T

Pumpkin code

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

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

And here is why we use an additional generic type argument: Self!

While T represents the content of the Truck, Self represents the specific kind of truck!

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

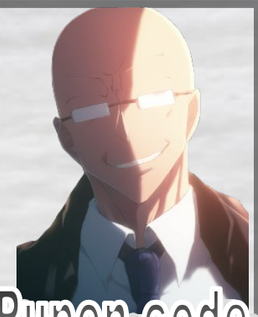
Self T

Pumpkin code

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

Self T

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

And here is why we use an additional generic type argument: Self!

While T represents the content of the Truck, Self represents the specific kind of truck!

You see? CabOver of T implements Truck of CabOver of T and T.

```
interface Truck<Self,T>{
  int aerodynamics();
  CargoC<List<T>> cargo();
  Point location();
  Self withLocation(Point p);

  Self driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    return this.withLocation(wayPoint);
  }
}
```

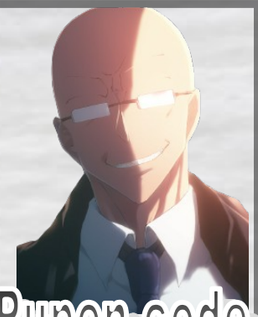
Self T

Pumpkin code

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

Self T

```
class Truck<T>{
  int aerodynamics();
  CargoC<List<T>> cargo(){ return cargo; }
  Point location(){ return location; }
  void location(Point p){ location=p; }

  void driveToward(Point dest, int duration){
    ... Point wayPoint=...;
    location(wayPoint);
  }
  private CargoC<List<T>> cargo;
  private Point location;
  Truck(Point p, CargoC<List<T>> c){location=p; cargo=c;}
}

class CabOver<T> : Truck<T>{
  CabOver(Point p, CargoC<List<T>> c){ super(p,c); }
  int aerodynamics(){ ... }
}
```



Pupon code

And here is why we use an additional generic type argument: Self!

While T represents the content of the Truck, Self represents the specific kind of truck!

You see? CabOver of T implements Truck of CabOver of T and T.

We instantiate the Self generic with our own type!


Oh, so elegant!


Oh, so elegant!
Wait, you understand what is going on here?

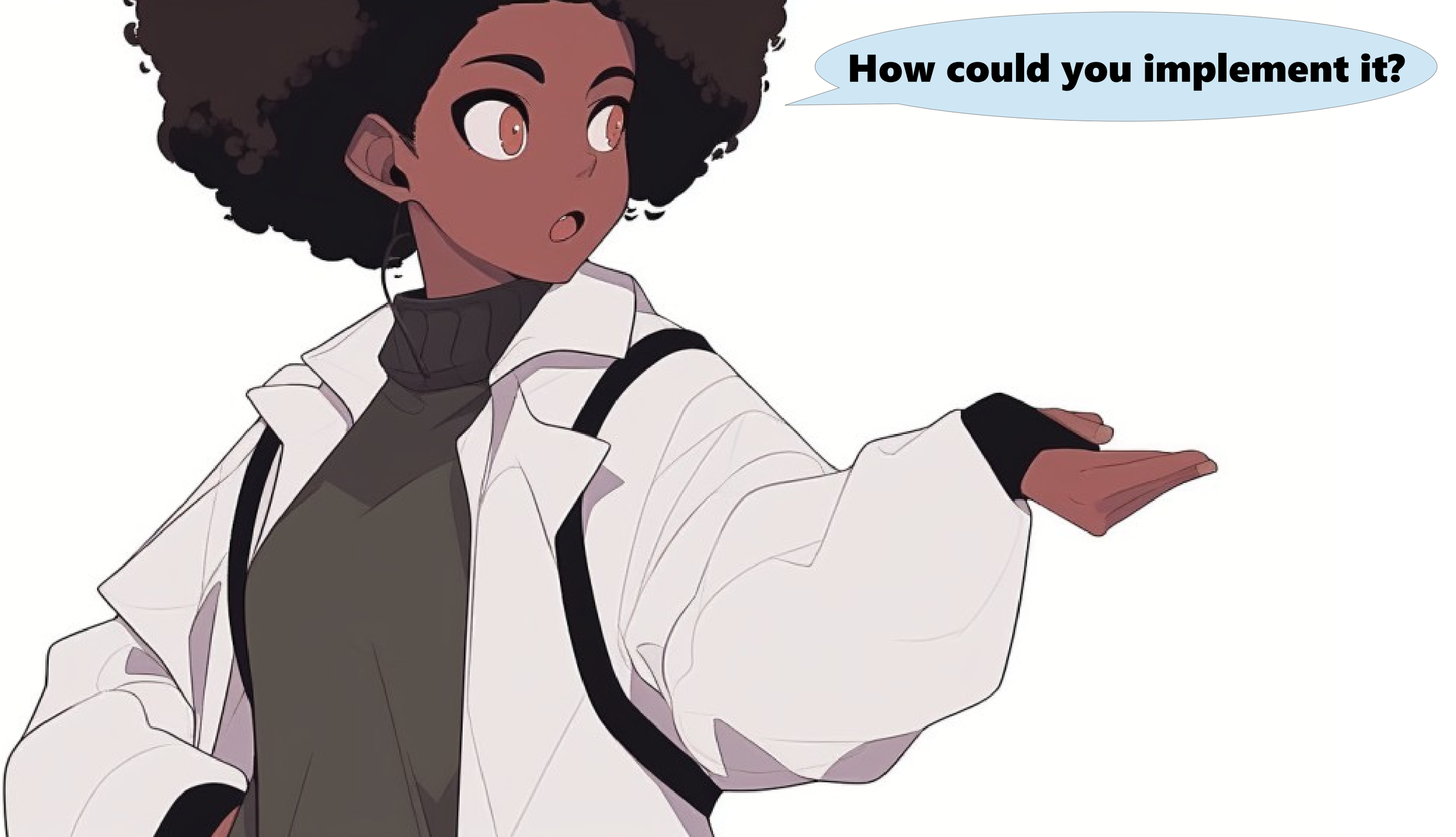

Yes, now it is clear

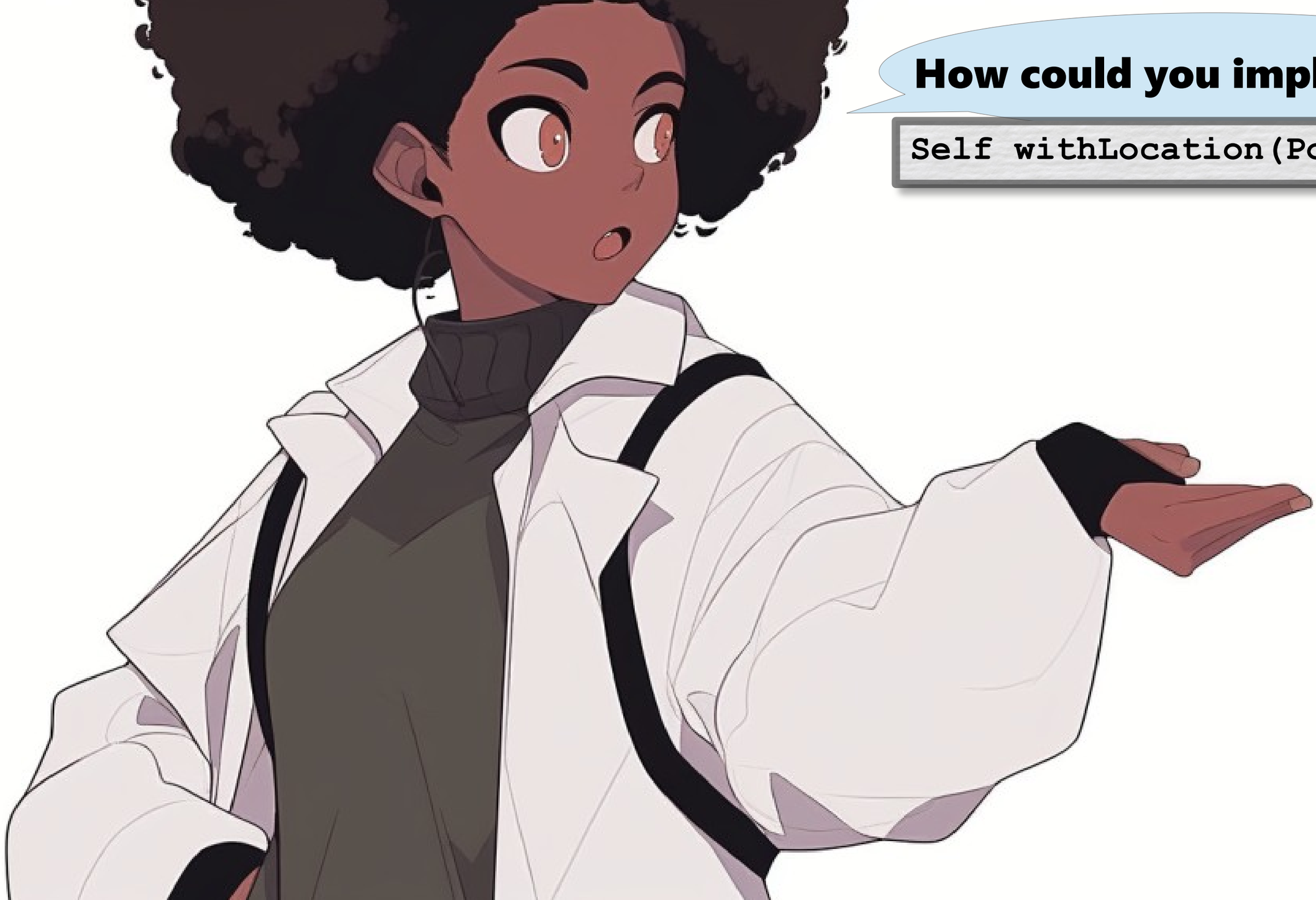

Yes, now it is clear
In Truck we have no way to create an object that looks

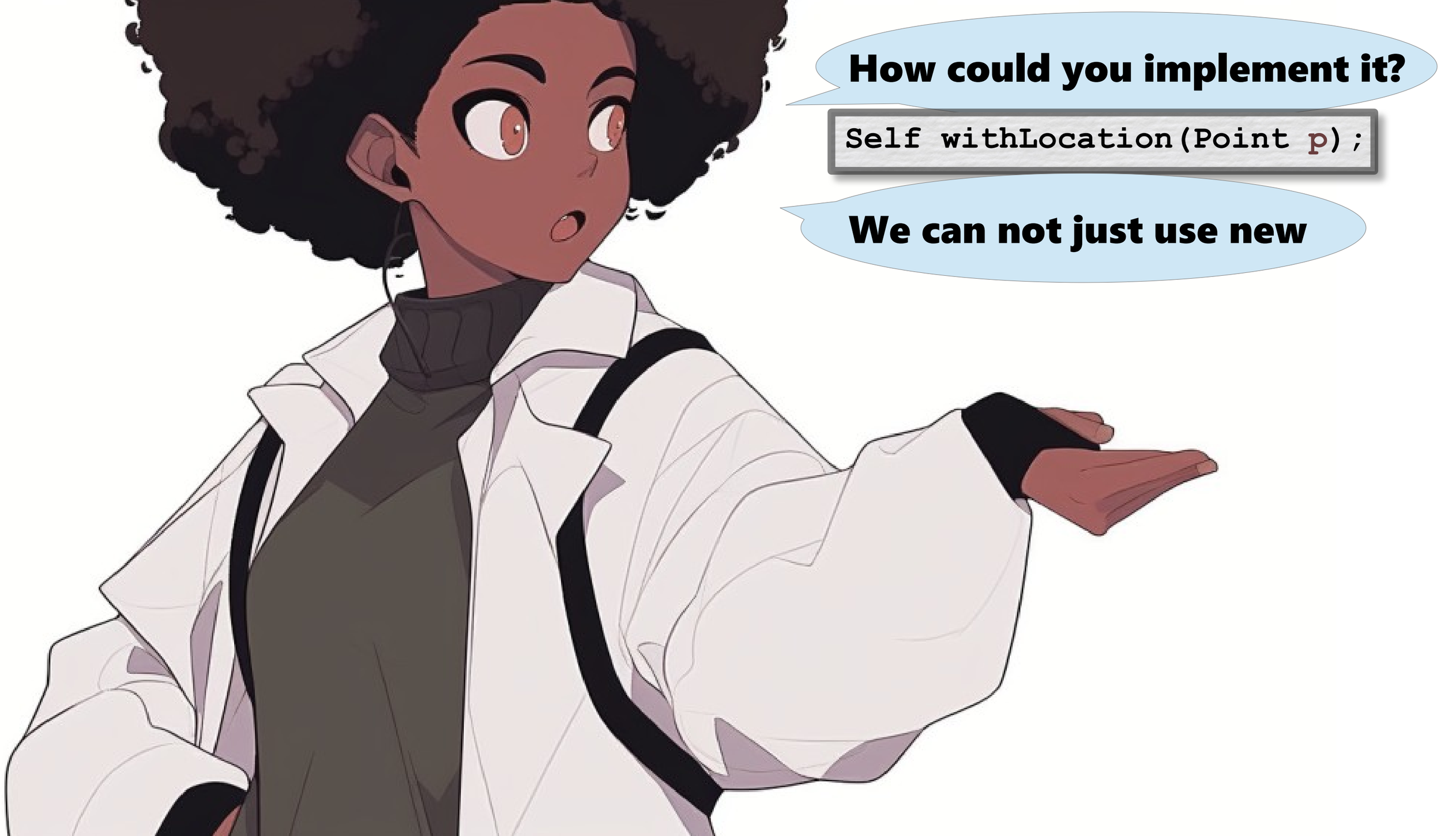

Yes, now it is clear
In Truck we have no way to create an object that looks like ‘this’, but with a single updated field

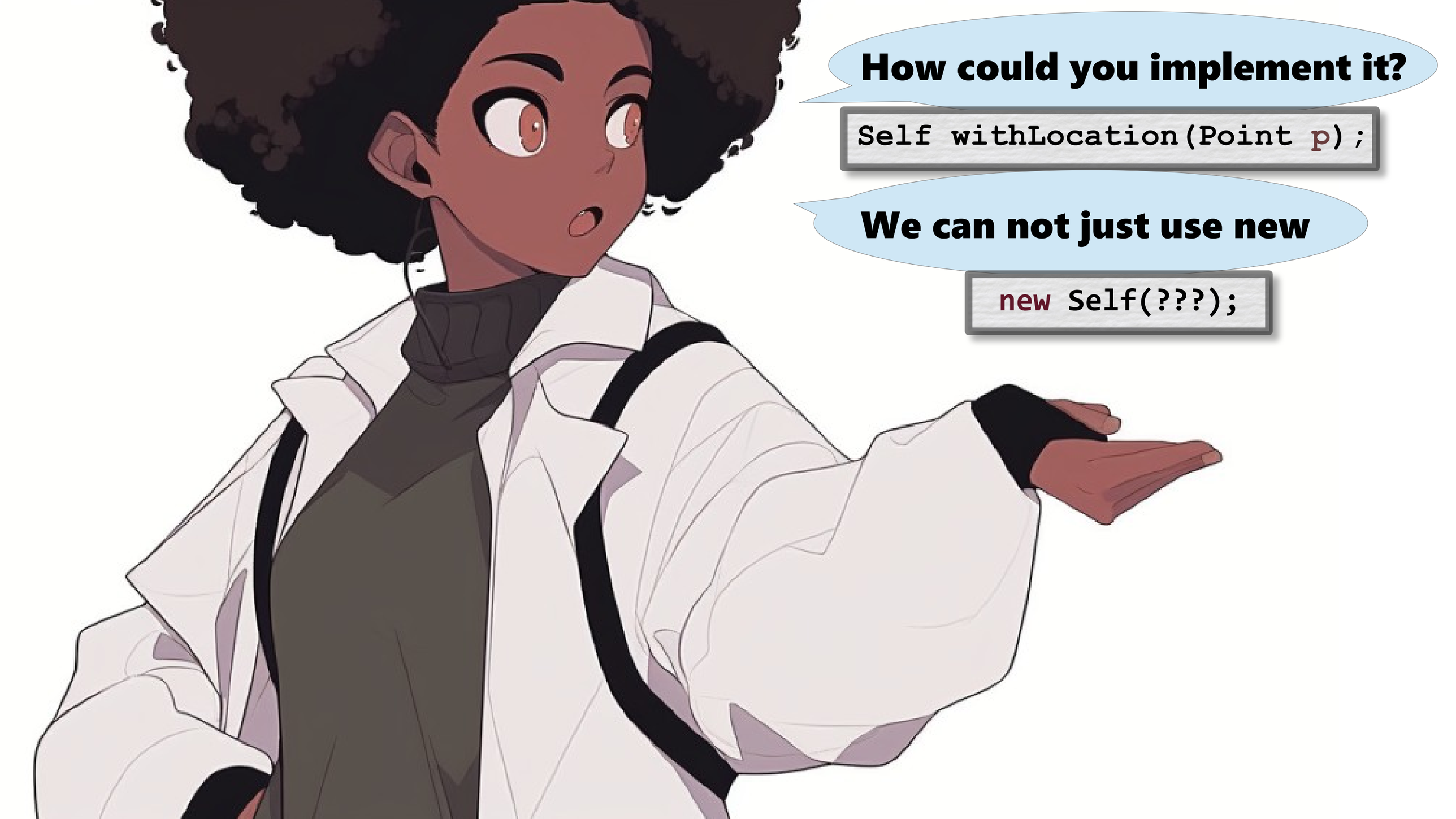

That is why
Truck.withLocation is abstract!


That is why
Truck.withLocation is abstract!
Self withLocation(Point p);


How could you implement it?


How could you implement it?
Self withLocation(Point p);


How could you implement it?
Self withLocation(Point p);
We can not just use new


How could you implement it?
Self withLocation(Point p);
We can not just use new
new Self(???);


What arguments would the constructor take?


What arguments would the constructor take?
new Self(?#?#?)


What arguments would the constructor take?
new Self(?#?#?)
The compiler have no clue


What arguments would the constructor take?
new Self(?#?#?)
The compiler have no clue
Self is just a type argument, not a type!


That is why we
have to wait patiently


That is why we
have to wait patiently
And implement it in CabOver!


That is why we have to wait patiently
CabOver<T> withLocation(Point p){ return new CabOver<>(p,cargo); }
And implement it in CabOver!


Truck can then rely on withers to ...


Truck can then rely on withers to ...
return withLocation(...anything!...);


Truck can then rely
on withers to ...
return withLocation(...anything!...);
encode any kind of behavior
needing new objects of its own kind


Truck can then rely on withers to ...
return withLocation(...anything!...);
encode any kind of behavior needing new objects of its own kind
Here ‘driveToward’ is the only such method,


Truck can then rely
on withers to ...
return withLocation(...anything!...);
encode any kind of behavior
needing new objects of its own kind
Here ‘driveToward’ is the only such method,
but we could have dozens like it in a real application!


Easy, right?