PECS_Explanation.md

Understanding PECS: Producer Extends, Consumer Super

The problem

A vehicle recycling system can be modelled with the following classes.

- `Vehicle`
- `Car` (extends `Vehicle`)
- `Scrap`
- `RefinedScrap` (extends `Scrap`)

A first attempt at a method to simulate car recycling:

```
public List<Scrap> mapVehicles(
            List<Car> cars,
            Function<Car, Scrap> mapper) {

        List<Scrap> result = cars
            .stream()
            .map(mapper)
            .toList();
        return result;
    }
```

This works perfectly fine! But it has some limitations.

Firstly, we might expect a method that accepts a `Function<Car, Scrap>` to also accept:

- `Function<Car, RefinedScrap>` (since `RefinedScrap` is-a `Scrap`, a function that turns a `Car` into `RefinedScrap` is also turning it into `Scrap`)
- `Function<Vehicle, Scrap>` (since `Car` is-a `Vehicle`, a function that can turn a `Vehicle` into `Scrap` can turn a `Car` into `Scrap`)

Secondly, if this method was modified to accept `List<Vehicle>` to generalise it, it would stop accepting `List<Car>`.

Java Generics Are Invariant

In Java, both `T` and `R` are invariant in `Function<T, R>`. So :

- A `List<Car>` is-NOT-a `List<Vehicle>`, though `Car` is-a `Vehicle`
- A `Function<Car, RefinedScrap>` is-NOT-a `Function<Car, Scrap>`, though `RefinedScrap` is-a `Scrap`
- A `Function<Vehicle, Scrap>` is-NOT-a `Function<Car, Scrap>`, though `Car` is-a `Vehicle`

The PECS Solution

- **Producer Extends**: When a generic type produces values, use `? extends T`
- **Consumer Super**: When a generic type consumes values, use `? super T`

Let's make `Function<T, R>` more general.

- `T` is consumed, so we'll use `? super T`
- `R` is produced, so we'll use `? extends R`

After this, we get `Function<? super T, ? extends R>`.

Applying PECS to Our Mapper

Instead of this signature:

```
public List<Scrap> mapVehicles(
           List<Car> cars,
           Function<Car, Scrap> mapper) { ... }
```

We can use PECS to make it more flexible like so.

```
public <T extends Vehicle> List<Scrap> mapVehicles(
           List<T> vehicles,
           Function<? super T, ? extends Scrap> mapper) { ... }
```

- The method now accepts `List` of `Vehicle` or any of its children
- `Function<Car, RefinedScrap>` works because `RefinedScrap` extends `Scrap`
- `Function<Vehicle, Scrap>` works because `Car` extends `Vehicle`
- `Function<Vehicle, RefinedScrap>` works for both of the reasons above

Why This Works

1. `? super T`, where `T` extends `Vehicle`, means "`Vehicle`, or any of its children's superclasses".

- This works because if a function can process a `Vehicle`, it can certainly process a `Car`, which is-a `Vehicle`.
- If a function can process a superclass of `Vehicle`, it can certainly process a `Vehicle`.

2. `? extends Scrap` means "`Scrap` or any of its subclasses"

- This works as `RefinedScrap` is-a `Scrap`.

Remembering PECS

An easy way to remember PECS:

- **Producer** (output) **Extends**: "Get me a T or better"
- **Consumer** (input) **Super**: "Give me a T or more general"