CS2030S Recitation

Week 7: Problem Set 4

brian

2026-02-19

National University of Singapore

Recap

Recap: Wildcards

- A substitute for any type
- Can be upper bounded (`? extends T`)
  - `?` can be `T` and its subtypes
  - This gives rise to covariance behaviour
- Can be lower bounded (`? super T`)
  - `?` can be `T` and its supertypes
  - This gives rise to contravariance behaviour
- Use unbounded when you know nothing about `T`. Better than rawtyping
  - If you have `List<?>` what are the types that are safe to put inside?
  - If you have `List<?>` what are the types that are safe to take out?

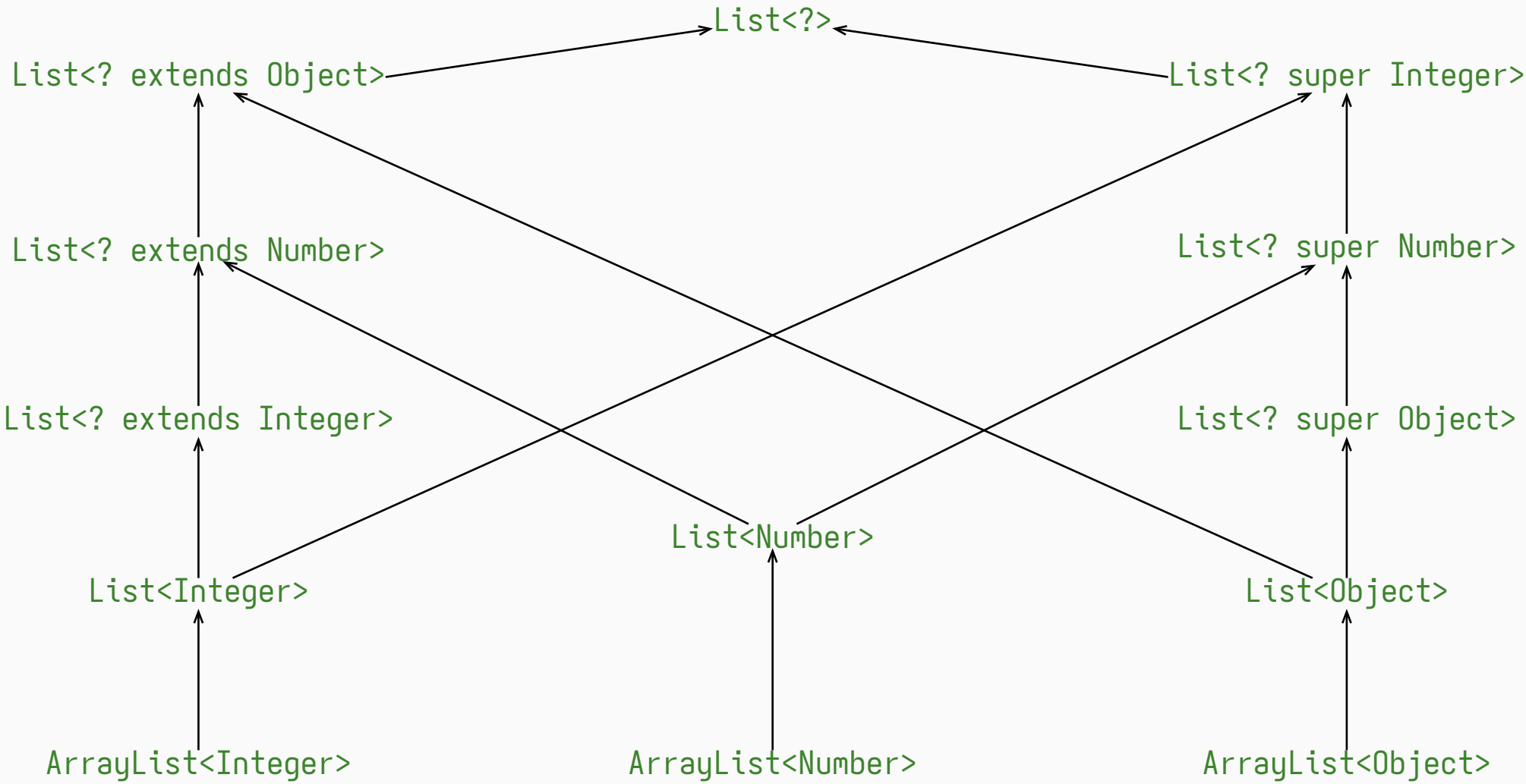
Recap: PECS

- Think from the perspective of the container
- Producer extends
  - Think of the types that something that produces `T` can produce, `T` and its subtypes
- Consumer super
  - Think of the types that can `T` can bind to, `T` and its supertypes

Q1: Hasse Diagram

- We have the following types.
  - `Integer` <: `Number` <: `Object`
  - `ArrayList<T>` <: `List<T>` <: `Object`
- Draw the Hasse diagram for all the `ArrayList` and `List`
- Draw `S` ⟶ `T` to represent `S` <: `T`
  - Omit transitive arrow

Q1: Hasse Diagram



Recap: Type Inference

- Very algorithmic
- Find constraints
  1. Argument Type: Bounds on the argument of the method
  2. Target Type: What type does `T` bind to
  3. Bounds on Type parameter: What does `T` extend or super
- Solve the constraints
  - Ignore subtypes not explicitly mentioned (since compiler can't tell what are the subtypes)
  - But you must consider supertypes

Q2: Bunch of fruit stuff

```
1 static <T extends Comparable<T>> T max(List<T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

`Fruit` implements `Comparable<Fruit>` interface and `Apple` <: `Fruit`

And we have `apples` of type `List<Apple>` and `fruits` of type `List<Fruit>`

Q2: Bunch of fruit stuff

```
1 class Fruit implements Comparable<Fruit> {
2   public int compareTo(Fruit f) {
3     return 0; // stub
4   }
5 }
6
7 class Apple extends Fruit {
8 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

a. What would `T` be inferred as for `Fruit f = max(fruits)`

Q2: Bunch of fruit stuff

Target	Args	Bounds
`T` <: `Fruit`	`List<Fruit>` <: `List<T>` $\Longrightarrow$ `Fruit` = `T`	`T` <: `Comparable<T>`

Q2: Bunch of fruit stuff

```
1 class Fruit implements Comparable<Fruit> {
2   public int compareTo(Fruit f) {
3     return 0; // stub
4   }
5 }
6
7 class Apple extends Fruit {
8 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

b.i. What would `T` be inferred as for `Fruit f = max(apples)`

Q2: Bunch of fruit stuff

Target	Args	Bounds
`T` <: `Fruit`	`List<Apple>` <: `List<T>` ⟹ `Apple` = `T`	`T` <: `Comparable<T>`

There is no such `T` that satisfies all constraints, `Apple` </: `Comparable<Apple>`.

Q2: Bunch of fruit stuff

```
1 class Fruit implements Comparable<Fruit> {
2   public int compareTo(Fruit f) {
3     return 0; // stub
4   }
5 }
6
7 class Apple extends Fruit {
8 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

b.ii. What would `T` be inferred as for `Apple a = max(apples)`

Q2: Bunch of fruit stuff

Target	Args	Bounds
`T` <: `Apple`	`List<Apple>` <: `List<T>` ⟹ `Apple` = `T`	`T` <: `Comparable<T>`

There is no such `T` that satisfies all constraints, `Apple` </: `Comparable<Apple>`.

Q2: Bunch of fruit stuff

```
1 class Fruit implements Comparable<Fruit> {
2   public int compareTo(Fruit f) {
3     return 0; // stub
4   }
5 }
6
7 class Apple extends Fruit {
8 }
```

```
1 List<Fruit> fruits = List.of(new Fruit(), new Apple());
2 List<Apple> apples = List.of(new Apple(), new Apple());
```

b.iii. What would `T` be inferred as for `Apple a = max(fruits)`

Q2: Bunch of fruit stuff

Target	Args	Bounds
`T` <: `Apple`	`List<Fruit>` <: `List<T>` ⟹ `Fruit` = `T`	`T` <: `Comparable<T>`

There is no such `T` that satisfies all constraints, `Apple` </: `Fruit`

In fact this even more dumb than before and is not salvagable. Reason being if you're taking out from a list of Fruit how can you expect it to be an Apple?

Q2: Bunch of fruit stuff

So how do we fix this (at least for i and ii)?

Notice that the issue was that `Apple` <: `Comparable<Apple>`

We should avoid touching `Apple` declaration but how can we make the method header more flexible?

Q2: Bunch of fruit stuff

```java
static <T extends Comparable<? super T>> T max(List<T> list) {
  T max = list.get(0);
  if (list.get(1).compareTo(max) > 0) {
    return list.get(1);
  }
  return max;
}
```

Q2: Bunch of fruit stuff

b.i.

Target	Args	Bounds
`T <: Fruit`	`List<Apple> <: List<T>` ⟹ `Apple = T`	`T <: Comparable<? super T>`

`Apple <: Fruit`

`Apple <: Comparable<Fruit> <: Comparable<? super Fruit> <: Comparable<? super Apple>`

Q2: Bunch of fruit stuff

b.ii.

Target	Args	Bounds
`T` <: `Apple`	`List<Apple>` <: `List<T>` ⟹ `Apple` = `T`	`T` <: `Comparable<? super T>`

`Apple` <: `Apple`

`Apple` <: `Comparable<Fruit>` <: `Comparable<? super Fruit>` <: `Comparable<? super Apple>`

Q2: Bunch of fruit stuff

If you want to apply PECS more

```
1 static <T extends Comparable<? super T>> T max(List<? extends T> list) {
2   T max = list.get(0);
3   if (list.get(1).compareTo(max) > 0) {
4     return list.get(1);
5   }
6   return max;
7 }
```

The End

bye!