Type Projection

Module Objectives

In this module, you will look at:

- The problem of mismatching argument types
- Implicit type projections as a solution
- Compatible conversions

Type Projection by Flowlets

Flowlets perform an implicit projection on the input objects if they do not match exactly what the process method accepts as arguments

This allows you to write a single process method that can accept multiple **compatible** types

---

Example of a Process Method

For example, if you have a process method:

```
@ProcessInput
count(String word) {
  ...
}
```

- You send data of type `Long` to this Flowlet
- That type does not exactly match what the process method expects

---

A Solution: Write A Lot of Methods

You could write another process method for `Long` numbers:

```
@ProcessInput count(Long number) {
count(number.toString());
}
```

- Need to do this for every type that you might possibly want to count
- Tedious and error-prone

A Better Solution: Type Projection

Type projection does this for you automatically

- If no process method is found that matches the type of an object exactly, it picks a method that is compatible with the object
- Because `Long` can be converted into a `String`, it is compatible with the original process method

---

Compatible Conversions 1 of 3

- Every primitive type that can be converted to a `String` is compatible with `String`
- A byte array is compatible with a `ByteBuffer` and vice-versa
- Any numeric type is compatible with numeric types that can represent it

For example:

- `int` is compatible with `long`, `float` and `double`
- `long` is compatible with `float` and `double`
- **but** `long` is not compatible with `int` because `int` cannot represent every `long` value

---

Compatible Conversions 2 of 3

- A collection of type A is compatible with a collection of type B, if type A is compatible with type B
- A collection can be an array or any Java `Collection`
- Example: `List<Integer>` is compatible with a `String[]` array
- Two maps are compatible if their underlying types are compatible
- Example: `TreeMap<Integer, Boolean>` is compatible with `HashMap<String, String>`

Other Java objects can be compatible if their fields are compatible.

Compatible Conversions 3 of 3

Class `Point` is compatible with `Coordinate`, because all common fields between the two classes are compatible. Projecting from `Point` to `Coordinate`, the color field is dropped; projecting from `Coordinate` to `Point` will leave the `color` field as `null`

```
class Point {
  private int x;
  private int y;
  private String color;
}

class Coordinates {
  int x;
  int y;
}
```

Conclusion

- Type projections help you keep your code generic and reusable
- They interact well with inheritance
- If a Flowlet can process a specific object class, then it can also process any subclass of that class

---

Module Summary

You should now understand:

- The problem of matching argument types
- How to use type projection to solve this and reduce duplicate methods
- Compatible conversions and when to make use of them

Module Completed