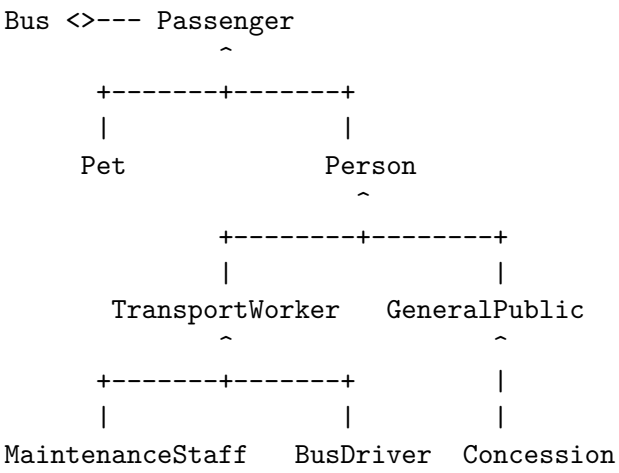
Generics

Table of contents

Practical for 2026-05-21-correct-programming[1] (Week 12). Extends java-generics[2] (Week 9) with practice designing generic classes and using bounded wildcards.

Setup

A simple `Bus` domain models a class hierarchy of things that can ride a bus:



[1] courses/csse2002/2026-05-21-correct-programming.pdf
[2] courses/csse2002/java-generics.pdf

`Passenger` is the common supertype every rider implements/extends; `Pet` and `Person` are its two direct subtypes; `TransportWorker` (with subtypes `MaintenanceStaff`, `BusDriver`) and `GeneralPublic` (with subtype `Concession`) are both `Person`s. A `Bus` holds a collection of `Passenger`s up to some capacity.

§1 Generic Bus

Task 0. The starting `Bus` class is not generic — it stores plain `Passenger`s, so a single `Bus` instance could hold a mix of `Pet`s and `BusDriver`s with no way to restrict it further:

```
public class Bus {
    private final List<Passenger> passengers;
    private final int capacity;

    public Bus(int capacity) {
        this.capacity = capacity;
        this.passengers = new ArrayList<>();
    }

    /**
     * @requires passenger != null
     * @ensures passenger is added to this bus and true is returned,
     *          unless this bus is already at capacity, in which case
     *          nothing changes and false is returned
     */
    public boolean addPassenger(Passenger passenger) {
        if (passengers.size() >= capacity) {
            return false;
        }
        passengers.add(passenger);
        return true;
    }

    public List<Passenger> getPassengers() {
        return passengers;
    }
}
```

Rewrite `Bus` to be generic, so a single `Bus` instance can be restricted to carrying just one kind of passenger (e.g. `Bus<Pet>`, `Bus<BusDriver>`).

Working

```
public class Bus<T> {
    private final List<T> passengers;
    private final int capacity;

    public Bus(int capacity) {
        this.capacity = capacity;
        this.passengers = new ArrayList<>();
    }

    /**
     * @requires passenger != null
     * @ensures passenger is added to this bus and true is returned,
     *          unless this bus is already at capacity, in which case
     *          nothing changes and false is returned
     */
    public boolean addPassenger(T passenger) {
        if (passengers.size() >= capacity) {
            return false;
        }
        passengers.add(passenger);
        return true;
    }

    public List<T> getPassengers() {
        return passengers;
    }
}
```

Task 1. As written, `Bus<T>` accepts *any* type argument at all — including types with nothing to do with `Passenger` (e.g. `Bus<String>`). Add a bound to `T` that restricts it to `Passenger` and its subtypes.

Working

```
public class Bus<T extends Passenger> {
    // ...unchanged from Task 0...
}
```

Bounding `T extends Passenger` restricts `Bus`'s type argument to `Passenger` or one of its subtypes (`Pet`, `Person`, `TransportWorker`, …), while a *particular* `Bus` instance still only carries one such type (e.g. `Bus<Pet>` can't also hold a `BusDriver`). It also means code inside `Bus<T>` could safely call any `Passenger` method directly on a `T` value, if needed.

Task 2. Write a `NoPetsBus<T>` subclass of `Bus<T>` that statically (at compile time, not with a runtime check) excludes `Pet` as a valid type argument.

> **i** Working
>
> ```
> public class NoPetsBus<T extends Person> extends Bus<T> {
>     public NoPetsBus(int capacity) {
>         super(capacity);
>     }
> }
> ```
>
> Bounding the *subclass*'s type parameter to `T extends Person` (rather than `Passenger`) is stricter than `Bus`'s own bound. Since `Pet` is a `Passenger` but not a `Person`, `NoPetsBus<Pet>` simply doesn't compile — the exclusion is enforced entirely by the type system, with no `instanceof` checks needed anywhere.

§2 Bus Stop

Task 3. Write a `trainingBus` method that takes a `Bus<BusDriver>` and a list of `TransportWorker`s, and adds every `BusDriver` among them to the bus (stopping early if the bus becomes full):

```
public static void trainingBus(Bus<BusDriver> bus, List<? extends TransportWorker>
 ↪  trainees) {
    ...
}
```

Working

```
/**
 * @requires bus != null && trainees != null
 * @ensures every BusDriver in trainees (in order) is added to bus,
 *          stopping as soon as bus is full
 */
public static void trainingBus(Bus<BusDriver> bus, List<? extends
 ↪  TransportWorker> trainees) {
    for (TransportWorker trainee : trainees) {
        if (trainee instanceof BusDriver busDriver) {
            boolean added = bus.addPassenger(busDriver);
            if (!added) {
                return;
            }
        }
    }
}
```

`trainees` only needs to be **read** from (never written to), so it takes the *upper-bounded* wildcard `List<? extends TransportWorker>` — this lets callers pass a `List<TransportWorker>`, `List<BusDriver>`, or `List<MaintenanceStaff>` alike, at the cost of the compiler only guaranteeing each element is *at least* a `TransportWorker` (hence the `instanceof` check before adding).

Task 4. Write a `transferStudents` method that moves every passenger out of a `Bus<Concession>` and into another bus that's allowed to carry `Concession` passengers (or any of their supertypes):

```
public static void transferStudents(Bus<Concession> from, Bus<? super Concession> to)
 ↪  {
    ...
}
```

Working

```
/**
 * @requires from != null && to != null
 * @ensures passengers are moved from `from` into `to`, in order, until either
 *          `from` is empty or `to` is full (any remaining passengers stay in
 ↪  `from`)
 */
public static void transferStudents(Bus<Concession> from, Bus<? super
 ↪  Concession> to) {
    Iterator<Concession> iterator = from.getPassengers().iterator();
    while (iterator.hasNext()) {
        Concession passenger = iterator.next();
        boolean added = to.addPassenger(passenger);
        if (!added) {
            return;
        }
        iterator.remove();
    }
}
```

`to` only needs to be **written** to, so it takes the *lower*-bounded wildcard `Bus<? super Concession>` — this lets callers pass a `Bus<Concession>`, `Bus<GeneralPublic>`, `Bus<Person>`, or `Bus<Passenger>` alike, since all of them can legally accept a `Concession` passenger via `addPassenger`.

Task 5. `transferStudents` only works for exactly `Bus<Concession>` as its source. Generalise it into a fully generic method that works for `Bus<T>` for *any* `T` that's at least a `Concession`.

Working

```java
/**
 * @requires from != null && to != null
 * @ensures passengers are moved from `from` into `to`, in order, until either
 *          `from` is empty or `to` is full (any remaining passengers stay in
 ↪  `from`)
 */
public static <T extends Concession> void transferAllConcession(Bus<T> from,
 ↪  Bus<? super T> to) {
    Iterator<T> iterator = from.getPassengers().iterator();
    while (iterator.hasNext()) {
        T passenger = iterator.next();
        boolean added = to.addPassenger(passenger);
        if (!added) {
            return;
        }
        iterator.remove();
    }
}
```

Introducing the type parameter `<T extends Concession>` on the *method itself* (rather than fixing `T = Concession`) means `from` can be `Bus<Concession>` **or** any more specific subtype-bus (e.g. a hypothetical `Bus<StudentConcession>`), while `to`'s wildcard bound `? super T` is resolved relative to whatever `T` the call site infers — `transferStudents` from Task 4 is just the special case where `T` is fixed to `Concession`.