Generics

Table of contents

Setup 1
§1 Generic Bus 3
§2 Bus Stop 6
A note on the spec tags 11

Practical for 2026-05-21-correct-programming[1] (Week 12). Extends java-generics[2] (Week 9) with practice designing a generic class, bounding its type parameter, and using wildcards and generic methods.

All of the classes below are supplied in a `provided.zip` on the course website. Make a new IntelliJ project and drop the contents in.

Setup

Every later question refers back to one class hierarchy. Each child has **one solid arrow** pointing straight up into its superclass, and a **hollow diamond** sits on the `Bus` end of the `Bus`–`Passenger` line.

`Passenger` is the common supertype of everything that can ride; `Pet` and `Person` are its two direct subtypes; `TransportWorker` (with subtypes `MaintenanceStaff` and `BusDriver`) and `GeneralPublic` (with subtype `Concession`) are both `Person`s. A `Bus` holds passengers up to a fixed capacity.

Keep Figure 1 in view — every wildcard bound in §2 is just a region of this tree.

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

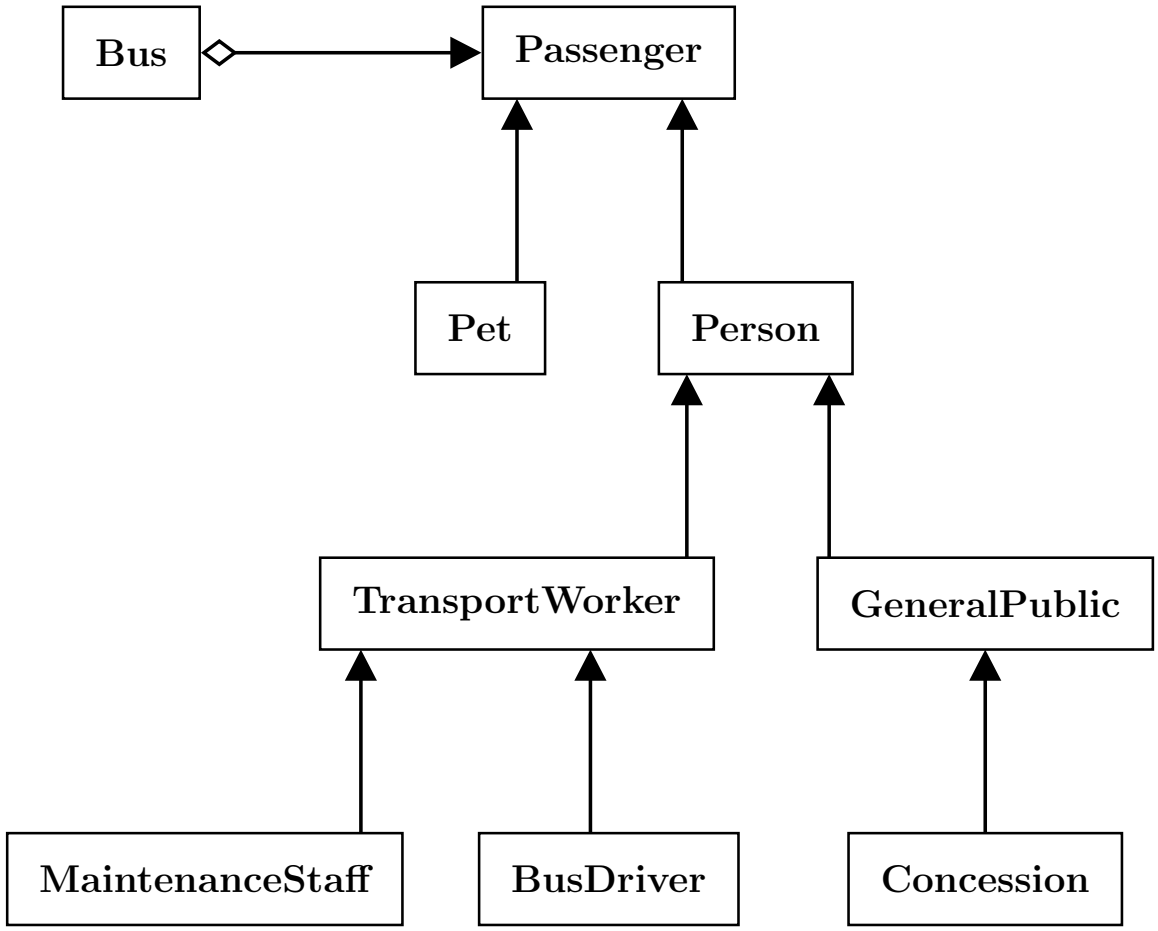


Figure 1: The passenger hierarchy the whole practical is built on. Each child has one solid arrowhead into its superclass, and a hollow aggregation diamond sits on the Bus end of the Bus–Passenger line.

§1 Generic Bus

Task 0. The provided `Bus` stores instances of `Object`. That forces callers to cast on the way out — `passengerOff()` hands back an `Object`, not a `Passenger`:

```
// The starting Bus class, before the rewrite.
public void passengerOn(Object passenger) { ... }
public Object passengerOff() { ... }

Passenger p = (Passenger) bus.passengerOff();   // cast needed at every call site
```

You could rewrite `Bus` to store a `Passenger[]` instead, but then a bus that may only carry the general public would need an entirely new class storing a `GeneralPublic[]`.

Rewrite `Bus` to be generic, with **one** type parameter, so both the casting and the duplication go away. (An *Unchecked Cast* warning during the rewrite is expected and can be safely ignored here.)

Working

```java
public class Bus<T> {
    private T[] passengers;
    private int currentCount;
    private int capacity;

    public Bus(int capacity) {
        // Creation of an array of objects, that is cast to be of type
        // T, is required due to how generics are implemented in Java.
        this.passengers = (T[]) new Object[capacity];
        this.capacity = capacity;
        this.currentCount = 0;
    }

    public void passengerOn(T passenger) {
        if (currentCount < capacity) {
            passengers[currentCount++] = passenger;
        }
    }

    public T passengerOff() {
        T exitingPassenger = null;
        if (currentCount > 0) {
            exitingPassenger = passengers[--currentCount];
        }
        return exitingPassenger;
    }

    public int getCurrentCount() {
        return currentCount;
    }

    public int getCapacity() {
        return capacity;
    }
}
```

The `(T[]) new Object[capacity]` line is the awkward part: Java erases type parameters at compile time, so `new T[capacity]` is not something the runtime can do — you allocate an `Object[]` and lie to the compiler about its type. That lie is exactly what the *Unchecked Cast* warning is about.
`passengerOn`/`passengerOff` are a stack, not a queue: `passengerOff` returns `passengers[--currentCount]`, i.e. the passenger who got on most recently, and returns `null` on an empty bus rather than throwing.

The remaining tasks extend `Bus` using **bounded** generic types. As with every prac, test your

code as you go.

Task 1. Modify your class so that only instances of `Passenger` (and its subclasses) are allowed on the bus.

> **i** Working

```
public class Bus<T extends Passenger> {
    ...
    public Bus(int capacity) {
        this.passengers = (T[]) new Passenger[capacity];
        ...
    }
    ...
}
```

Two things change together. The bound `T extends Passenger` restricts the *type argument* to `Passenger` or a subtype, so `Bus<String>` no longer compiles. And because `T`'s erasure is now `Passenger` rather than `Object`, the backing array is allocated as `new Passenger[capacity]`.
A bound also means code inside `Bus<T>` may call any `Passenger` method directly on a `T` value.

Now suppose there is a type of bus that does not allow pets on board — only people.

Task 2. Write a new class `NoPetsBus` which **extends** the generic `Bus` class and only allows non-pets on board.

> **i** Working

```
public class NoPetsBus<T extends Person> extends Bus<T> {
    public NoPetsBus(int capacity) {
        super(capacity);
    }
}
```

The subclass declares a *stricter* bound than its parent: `T extends Person` rather than `T extends Passenger`. `Pet` is a `Passenger` but not a `Person` (see Figure 1), so `NoPetsBus<Pet>` simply fails to compile — the exclusion is enforced by the type system, with no `instanceof` check anywhere.

§2 Bus Stop

The remaining questions need generic methods and wildcards (`?`). Create a new **non-generic** class called `BusStop`; each method below goes in it.

Training Bus

Bus stops are the starting points of training routes for new bus drivers. While a driver is being trained, only the trainee driver and other transport workers are allowed on board. The crew training the driver could be any combination of transport workers: only other bus drivers (`List<BusDriver>`), only maintenance staff (`List<MaintenanceStaff>`), or a mix (`List<TransportWorker>`).

Task 3. Write the `trainingBus` method:

```
static Bus<TransportWorker> trainingBus(BusDriver trainee, List<...> trainers)
```

where `...` is replaced with something that allows *any of the `List` types mentioned above* to be passed in. The method returns a new bus (capacity 10) with the trainee driver and all the trainers on board. Write a Javadoc comment with appropriate `@require` and `@ensure` tags.

> **Hint.** If a `List<A>` is required, a `List<B>` cannot be passed in, even if `B extends A`. So `List<TransportWorker>` alone will not do — you need wildcards.

i Working

```
/**
 * Populate a bus to be used to train a new driver.
 *
 * @param trainee   New driver to be trained.
 * @param trainers  TransportWorkers who will train the new driver.
 * @require trainee != null && trainers.size() < 10
 * @ensure  \result contains trainee and all trainers
 */
public static Bus<TransportWorker> trainingBus(BusDriver trainee,
        List<? extends TransportWorker> trainers) {

    Bus<TransportWorker> bus = new Bus<>(10);
    bus.passengerOn(trainee);
    for (TransportWorker trainer : trainers) {
        bus.passengerOn(trainer);
    }
    return bus;
}
```

The hint is the whole point: generic types are **invariant**, so `List<BusDriver>` is *not* a subtype of `List<TransportWorker>`. `List<? extends TransportWorker>` is the supertype of all of them.
The precondition `trainers.size() < 10` is what keeps the postcondition honest — `passengerOn` silently does nothing once the bus is full, so without a bound on the crew size the bus would not actually contain all the trainers.

Transfer Students

At bus stops, transfers happen between school buses — used exclusively for school children on concession fares, i.e. `Bus<Concession>` — and ordinary buses. When school children depart their bus they can only get onto buses that allow concession holders on board (not, for example, a bus meant exclusively for transport workers). So they can move either onto another concession-only bus, or onto a bus that allows *superclasses* of concession holders on board.

Task 4. Write the `transferStudents` method:

```
static void transferStudents(Bus<Concession> students, Bus<...> bus)
```

where `...` is replaced with something that ensures concession holders are allowed onto `bus`. All students on the `students` bus should be transferred to `bus`. Write a Javadoc comment with appropriate `@require` and `@ensure` tags.

> **Hint.** Valid bus types are `Bus<Concession>`, `Bus<GeneralPublic>`, `Bus<Person>` and `Bus<Passenger>`. Wildcards will be useful here too.

Working

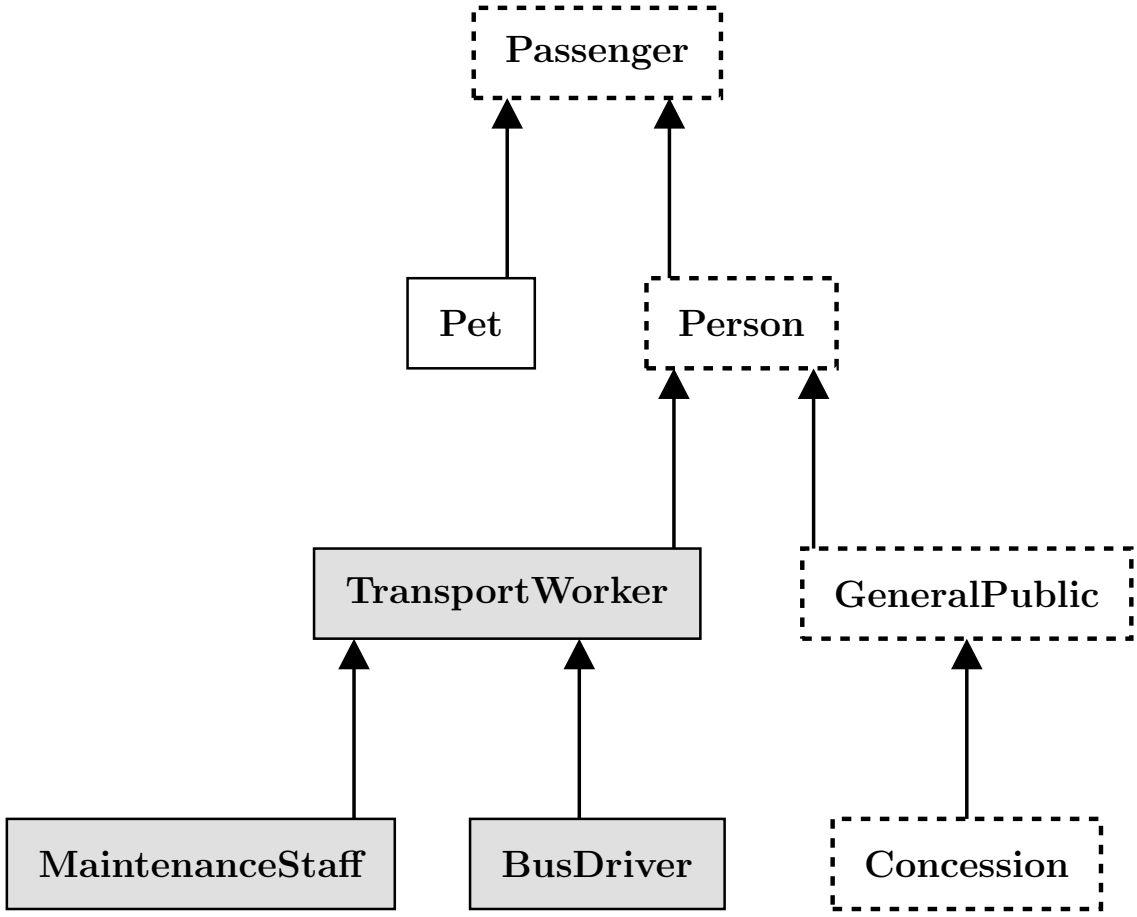
```
/**
 * Transfer students to the new bus.
 *
 * @param students Bus only containing Concession fare riders.
 * @param bus      Bus to which students will transfer.
 * @require students != null
 *          && bus.getCapacity() >= students.getCurrentCount()
 * @ensure  students.getCurrentCount() == 0
 *          && bus.getCurrentCount >= \old(students).getCurrentCount()
 *          && bus contains all passengers from \old(students)
 */
public static void transferStudents(Bus<Concession> students,
        Bus<? super Concession> bus) {

    // Ensure loop condition does not change while iterating
    int numStudents = students.getCurrentCount();
    for (int i = 0; i < numStudents; i++) {
        Concession student = students.passengerOff();
        bus.passengerOn(student);
    }
}
```

Two things worth pulling out:

- **The lower-bounded wildcard.** `bus` is only ever *written to*, so `Bus<? super Concession>` is the right bound — it accepts a bus whose type argument is `Concession` or any supertype of it, which covers the four types the hint lists, all of which can legally accept a `Concession` via `passengerOn`. (`Bus<Object>` satisfies the bound too; the hint stops at `Passenger`.)
- **The cached loop bound.** Writing `for (int i = 0; i < students.getCurrentCount(); i++)` would be a bug: every `passengerOff()` decrements the count while `i` increments, so the two meet in the middle and only half the students get transferred. Reading the count once, before the loop, is what the code comment ("Ensure loop condition does not change while iterating") is about.

Reading the two wildcard directions off the hierarchy

Tasks 3 and 4 need wildcards in **opposite directions**. Task 3 accepts `List<BusDriver>`, `List<MaintenanceStaff>` and `List<TransportWorker>`. Task 4 accepts `Bus<Concession>`, `Bus<GeneralPublic>`, `Bus<Person>` and `Bus<Passenger>`. Laid on the same tree, `? extends` reaches *down* from its bound and `? super` reaches *up*.



grey fill: accepted by `List<? extends TransportWorker>` (Task 3)

dashed border: accepted by `Bus<? super Concession>` (Task 4)

Figure 2: The two wildcard bounds this practical needs, mapped onto the hierarchy. Grey fill marks the type arguments accepted by `List<? extends TransportWorker>` (Task 3); a dashed border marks those accepted by `Bus<? super Concession>` (Task 4).

Which direction you need falls out of what the method does with the parameter: `trainers` is only **read** (upper bound, `? extends`), `bus` is only **written** (lower bound, `? super`).

Transfer Concession

Suppose new subclasses of `Concession` were added at a later stage — say `SchoolStudent` and `UniStudent`.

Task 5. Write the `transferAllConcession` method:

```
static ... void transferAllConcession(Bus<...> students, Bus<...> bus)
```

which does the same thing as `transferStudents`, except it works for `Concession` **or** any potential subclass of it. That is, if `SchoolStudent` were a subclass of `Concession`, this should work without error:

```
Bus<SchoolStudent> students = new Bus<>(30);
Bus<GeneralPublic> bus = new Bus<>(50);
transferAllConcession(students, bus);
```

Hint. You will need a generic *method* for this one.

Working

```
public static <T extends Concession> void transferAllConcession(
        Bus<T> students,
        Bus<? super T> bus) {

    int numStudents = students.getCurrentCount();
    for (int i = 0; i < numStudents; i++) {
        T student = students.passengerOff();
        bus.passengerOn(student);
    }
}
```

Declaring `<T extends Concession>` on the **method** (before the return type) is what makes the two parameters move together: `T` is inferred per call site from `students`, and `bus`'s bound `? super T` is then resolved *relative to that same T*. With `Bus<SchoolStudent>` and `Bus<GeneralPublic>`, `T` infers as `SchoolStudent`, and `GeneralPublic` is indeed a supertype of it.
Task 4's `transferStudents` is exactly the special case where `T` is pinned to `Concession`.

A note on the spec tags

This practical writes `@require` / `@ensure` (no trailing `s`), and uses `\result` for the return value and `\old(x)` for a parameter's pre-call state. The course uses both spellings: `@requires` / `@ensures` in the specification tag table and the exam preparation material, and the `-s`-less forms in the SOLID contracts and this prac. `\result` is used in the SOLID contracts too. Follow whatever the sheet in front of you uses; see java-specification[3] for the tags themselves.

Task 4's `@ensure` clause writes `bus.getCurrentCount` with the parentheses missing. That is a slip in the practical solution.

[3] courses/csse2002/java-specification.pdf