Formal definitions aren’t immediately helpful

Within the type system of a programming language, a typing rule or a type constructor is:

- **covariant** if it preserves the ordering of types (\(\leq\)), which orders types from more specific to more generic;
- **contravariant** if it reverses this ordering;
- **bivariant** if both of these apply (i.e., both \(I<A> \leq I<B>\) and \(I<B> \leq I<A>\) at the same time);
- **invariant** or **nonvariant** if neither of these applies.

https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
Instead, how about an example

```java
public static class Car {
    public final String make;
    public final String model;
    public final int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}

public static class Porsche extends Car {
    public Porsche(String model, int year) {
        super("Porsche", model, year);
    }
}

public static class Porsche911 extends Porsche {
    public Porsche911(int year) {
        super("911", year);
    }
}

public static class Chevy extends Car {
    public Chevy(String model, int year) {
        super("Chevy", model, year);
    }
}

public static class Corvette extends Chevy {
    public Corvette(int year) {
        super("Corvette", year);
    }
}
```

Every *Porsche* is a *Car*. Not every *Car* is a *Porsche*. (Formally: *Porsche* ⊆ *Car*)
Instead, how about an example

Every *Porsche* is a *Car*. Not every *Car* is a *Porsche*. (Formally: $\text{Porsche} \leq \text{Car}$)
Liskov Substitution Principle

Liskov substitution principle - Wikipedia, the free encyclopedia

https://en.wikipedia.org/wiki/Liskov_substitution_principle

More formally, the Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address entitled Data abstraction and hierarchy.

Principle - Origins - A typical violation - See also
Liskov Substitution Principle
Liskov Substitution Principle

If somebody is expecting a Car, you can give them a Porsche.
But if somebody is expecting a Porsche, you can’t just give them any Car.

Subtypes can always be passed as arguments where the supertype is expected. (Porsche is a subtype of Car.)
If you have a Corvette, you can treat it as if it’s a Chevy. Or a Car. Or an Object.

If you have a Car, you can’t just pretend it’s a Corvette.

Java does not require explicit type casting when you’re upcasting
Because of the Liskov Substitution Principle!
What about lists of cars?

(To keep things simple, we’re going back to GList from week 1)

GList<Car> carList = GList.of(
    new Car("Dodge", "Diplomat", 1985),
    new Car("Ford", "Pinto", 1977));

GList<Car> carList2 = GList.of(
    new Porsche("928S4", 1988), // allowed because Porsche can go anywhere a Car can go
    new Car("Ford", "Pinto", 1977));

GList<Porsche> porscheList = GList.of(
    new Porsche("944", 1987), // a Ford cannot be substituted for a Porsche
    new Porsche("356", 1964));

GList<Porsche911> porsche911List = GList.of(
    new Porsche911(1991),
    new Porsche911(2015));
Covariant relationships

If Porsche911 ≤ Porsche ≤ Car, then what about lists of them?
(List of Porsche911) ≤ (List of Porsche) ≤ (List of Car), right?

If somebody is expecting a list of cars and you give them a list of Porsches, then that’s still fine. If they’re expecting a list of Porsches and you give them a list of other cars, that’s not fine. So we can say:

Lists of T are covariant with T.
Covariance in Java

Our existing list classes are limited in how they’ll accept covariance

```java
GList<Car> carList = GList.of(
    new Car("Dodge", "Diplomat", 1985),
    new Car("Ford", "Pinto", 1977));

GList<Porsche> porscheList = GList.of(
    new Porsche("944", 1987),
    new Porsche("356", 1964));

GList<Car> manyCars = carList.concat(porscheList); // error!
```

As-is, we can't put a GList<Porsche> where a GList<Car> is expected
Mutation lets you violate covariance
Amazingly, this code compiles.
(It fails at runtime.)

Example (arrays w/ mutation):
Porsche[] porsches = {
    new Porsche("944", 1987),
    new Porsche("356", 1964)
};
Car[] cars = porsches;
cars[0] =
    new Car("Ford", "Pinto", 1977);
Covariance and mutation

Mutation lets you violate covariance
Amazingly, this code compiles.
(It fails at runtime.)

Example (arrays w/ mutation):

```java
Porsche[] porsches = {
    new Porsche("944", 1987),
    new Porsche("356", 1964)
};

Car[] cars = porsches;
cars[0] =
    new Car("Ford", "Pinto", 1977);
```

You just put a Ford into an array of Porsches!
Covariance and mutation

Mutation lets you violate covariance
Amazingly, this code compiles.
(It fails at runtime.)

Example (arrays w/ mutation):
Porsche[] porsches = {
    new Porsche("944", 1987),
    new Porsche("356", 1964)
};
Car[] cars = porsches;
cars[0] =
    new Car("Ford", "Pinto", 1977);

You just put a Ford into an array of Porsches!
Covariance, mutation, and Java generics

All the built-in collections classes (java.util.*) use mutation
When they added Java generics (Java5), they kept the mutation

If you wanted (List of Porsche911) ≤ (List of Porsche) ≤ (List of Car)
and you allowed mutation, then mutation would violate the covariance

Consequently, covariance on generics isn’t “on by default”
Covariance and functional lists

If you have a list of Porsches, it’s also a list of Cars.

If you treat it as a list of Cars, then add a Ford, you’ll get a list of Cars. But the original list of Porsches is still all Porsches. No mutation!
Java type parameters and covariance

```
GList<Car> carList = GList.of(
    new Car("Dodge", "Diplomat", 1985),
    new Car("Ford", "Pinto", 1977));

GList<Porsche> porscheList = GList.of(
    new Porsche("944", 1987),
    new Porsche("356", 1964));

GList<Car> upcastPorsches1 = porscheList; // fails because types are not compatible
GList<Car> upcastPorsches2 = porscheList.map(x->x); // works, but inefficient
GList<? extends Car> wildcardPorscheList = porscheList; // succeeds, but less useful
```
Java type parameters and covariance

GList<Car> carList = GList.of(
  new Car("Dodge", "Diplomat", 1985),
  new Car("Ford", "Pinto", 1977));

GList<Porsche> porscheList = GList.of(
  new Porsche("944", 1987),
  new Porsche("356", 1964));

GList<Car> upcastPorsches1 = porscheList; // fails because types are not compatible
GList<Car> upcastPorsches2 = porscheList.map(x->x); // works, but inefficient
GList<? extends Car> wildcardPorscheList = porscheList; // succeeds, but less useful

Identity function <Porsche to Car> : types correctly because Liskov substitution principle
Java type parameters and covariance

GList<Car> carList = GList.of(
    new Car("Dodge", "Diplomat", 1985),
    new Car("Ford", "Pinto", 1977));

GList<Porsche> porscheList = GList.of(
    new Porsche("944", 1987),
    new Porsche("356", 1964));

GList<Car> upcastPorsches1 = porscheList; // fails because types are not compatible
GList<Car> upcastPorsches2 = porscheList.map(x->x); // works, but inefficient
GList<? extends Car> wildcardPorscheList = porscheList; // succeeds, but less useful

Wildcard type: matches Car and subtypes of Car
Explain `map(x->x)` again please?

Here’s our original code:
```
GList<Porsche> porscheList = ...;
GList<Car> upcastPorsches2 = porscheList.map(x->x); // works, but inefficient
```

Map can change types (e.g., map list of strings to list of string lengths)
```
public <R> GList<R> map(Function<T,R> func) { ... }
```

In this example, `T` is Porsche, `R` is Car. Java infers that `x->x` is a lambda of type `Function<Porsche, Car>`

And, via the Liskov Substitution Principle, it’s safe to “upcast” a Porsche to a Car
Adding covariance to GList

Beforehand (constructor):
```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<T> tailList) {
    this.value = value;
    this.tailList = tailList;
}
```

Afterward:
```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<? extends T> tailList) {
    this.value = value;
    this.tailList = upcast(tailList);
}

public static <T> GList<T> upcast(GList<? extends T> input) {
    return input.map(x->x);
}
```
Adding covariance to GList

Beforehand (constructor):

```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<T> tailList) {
    this.value = value;
    this.tailList = tailList;
}
```

Afterward:

```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<? extends T> tailList) {
    this.value = value;
    this.tailList = upcast(tailList);
}

public static <T> GList<T> upcast(GList<? extends T> input) {
    return input.map(x->x);
}
```

It’s okay if any subtype of T is in the tail.
Adding covariance to GList

Beforehand (constructor):

private final T value;
private final GList<T> tailList;

protected GList(T value, GList<T> tailList) {
    this.value = value;
    this.tailList = tailList;
}

Afterward:

private final T value;
private final GList<T> tailList;

protected GList(T value, GList<? extends T> tailList) {
    this.value = value;
    this.tailList = upcast(tailList);
}

public static <T> GList<T> upcast(GList<? extends T> input) {
    return input.map(x -> x);
}

But we want to simplify, which we know is safe.
Adding covariance to GList

**Beforehand (constructor):**
```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<T> tailList) {
    this.value = value;
    this.tailList = tailList;
}
```

**Afterward:**
```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<? extends T> tailList) {
    this.value = value;
    this.tailList = upcast(tailList);
}

public static <T> GList<T> upcast(GList<? extends T> input) {
    return input.map(x->x);
}
```

Safe because we can substitute for T. (Slow!)
Adding covariance to GList

Beforehand (constructor):

```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<T> tailList) {
    this.value = value;
    this.tailList = tailList;
}
```

Afterward:

```java
private final T value;
private final GList<T> tailList;

protected GList(T value, GList<? extends T> tailList) {
    this.value = value;
    this.tailList = upcast(tailList);
}

public static <T> GList<T> upcast(GList<? extends T> input) {
    @SuppressWarnings("unchecked")
    GList<T> castInput = (GList<T>) input;
    return castInput;
}
```

Optimization: no copying.
Why is the unchecked typecast safe?

Beforehand (copy the list using map - completely type-safe):

```java
public static <T> GList<T> upcast(GList<? extends T> input) {
    return input.map(x->x);
}
```

Afterward (using an unchecked cast):

```java
public static <T> GList<T> upcast(GList<? extends T> input) {
    @SuppressWarnings("unchecked")
    GList<T> castInput = (GList<T>) input;
    return castInput;
}
```
Answer: Because of type erasure!

Beforehand (after type erasure):

```java
public static <? extends GList<?>> upcast(GList<?> input) {
    return input.map(x->x);
}
```

The “map” just copies the list. Nothing changes.

Afterward (after type erasure):

```java
public static <? extends GList<?>> upcast(GList<?> input) {
    return input;
}
```

With mutation forbidden, we don’t need to make a copy. Just return it.
Answer: Because of type erasure!

Beforehand (after type erasure):
```java
    return input.map(x->x);
}
```

The “map” just copies the list. Nothing changes.

Afterward (after type erasure):
```java
    return input;
}
```

At runtime, Java doesn't know the type parameters. If the map-copy version is safe, so is the fast version. With mutation forbidden, we don't need to make a copy. Just return it.
Beforehand (methods):

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<T> afterTail) {
    return new GList<>(
        value,
        tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ...
    public GList<T> concat(GList<T> afterTail) {
        return afterTail;
    }
    ...
}
```

Afterward:

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<? extends T> afterTail) {
    return new GList<>(
        value,
        tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ...
    public GList<T> concat(GList<? extends T> afterTail) {
        return upcast(afterTail);
    }
    ...
}
```
Input parameters should accept subtypes

Beforehand (methods):

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<T> afterTail) {
    return new GList<>(value,
        tailList.concat(afterTail));
}

public static class Empty<T> ...
    ...
    public GList<T>
    concat(GList<T> afterTail) {
        return afterTail;
    }
    ...
```

Afterward:

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<? extends T> afterTail) {
    return new GList<>(
        value,
        tailList.concat(afterTail));
}

public static class Empty<T> ...
    ...
    public GList<? extends T>
    concat(GList<? extends T> afterTail) {
        return upcast(afterTail);
    }
    ...
```

No change: `add()` already accepts subtypes of `T` (Liskov Substitution Principle!)
Input parameters should accept subtypes

Wildcard type parameters to accept subtypes.

```java
public GList<T> concat(GList<T> afterTail) {
    return new GList<>(value,
        tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ... public GList<T>
        concat(GList<T> afterTail) {
            return afterTail;
        }
    ...
}

public GList<T> concat(GList<? extends T> afterTail) {
    return new GList<>(value,
        tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ... public GList<T>
        concat(GList<? extends T> afterTail) {
            return upcast(afterTail);
        }
    ...
}
```
Beforehand (methods):

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<T> afterTail) {
    return new GList<>
        (value,
         tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ... public GList<T>
    concat(GList<T> afterTail) {
        return afterTail;
    }
}
```

Afterward:

```java
public GList<T> add(T value) {
    return new GList<>(value, this);
}

public GList<T> concat(GList<? extends T> afterTail) {
    return new GList<>
        (value,
         tailList.concat(afterTail));
}

public static class Empty<T> ... {
    ... public GList<T>
    concat(GList<? extends T> afterTail) {
        return upcast(afterTail);
    }
}
```

Convert `GList<? extends T>` to `GList<T>` (Which we know is safe.)
What about contravariance?

For “in” types (e.g., the argument of `concat()` ) we have covariance
Example: You can concatenate a list of Porsche911’s to a list of Porsches and get a list of Porsches

For “out” types (e.g., the return value) we have contravariance
You can assign a “list of Porsches” to a variable typed “list of Cars”

```java
GList<Porsche> morePorsches = porscheList.concat(porsche911list);
GList<Car> moreCars = upcast(porscheList.concat(porsche911list));
```
What about contravariance?

For “in” types (e.g., the argument of `concat()`) we have covariance
Example: You can concatenate a list of Porsche911’s to a list of Porsches and get a list of Porsches

For “out” types (e.g., the return value) we have contravariance
You can assign a “list of Porsches” to a variable typed “list of Cars”

```java
GList<Porsche> morePorsches = porscheList.concat(porsche911list);
GList<Car> moreCars = upcast(porscheList.concat(porsche911list));
```

Covariance on the input: GList<Porsche911> is accepted
What about contravariance?

For “in” types (e.g., the argument of `concat()`) we have covariance.
Example: You can concatenate a list of Porsche911’s to a list of Porsches and get a list of Porsches.

For “out” types (e.g., the return value) we have contravariance.
You can assign a “list of Porsches” to a variable typed “list of Cars.”

```java
GList<Porsche> morePorsches = porscheList.concat(porsche911list);
GList<Car> moreCars = upcast(porscheList.concat(porsche911list));
```
What about contravariance?

For “in” types (e.g., the argument of `concat()`) we have covariance
Example: You can concatenate a list of Porsche911’s to a list of Porsches and get a list of Porsches

For “out” types (e.g., the return value) we have contravariance
You can assign a “list of Porsches” to a variable typed “list of Cars”

```java
GList<Porsche> morePorsches = porscheList.concat(porsche911list);
GList<Car> moreCars = upcast(porscheList.concat(porsche911list));
```

Annoying: do we need to explicitly upcast the type?
Sadly, yes, but, ... uggh.

We do have Java syntax that’s relevant: `GList<? super T>`
If we had `GList<? super Porsche911>`, then `GList<Car>` would match

But if we declare it like this:

```java
public GList<? super T> concat(GList<? extends T> afterTail)
```

Then this otherwise sensible code breaks:

```java
GList<Car> manyCars = carList.concat(porscheList);
```

Java’s unhelpful error:

```
GList<? super Car> cannot be automatically converted to GList<Car>
```
Sadly, yes, but, ... uggh.

We do have Java syntax that’s relevant: `GList<? super T>`
If we had `GList<? super Porsche911>`, then `GList<Car>` would match

But if we declare it like this:
```java
public GList<? super T> concat(GList<? extends T> afterTail)
```

Then this otherwise sensible code breaks:
```java
GList<Car> manyCars = carList.concat(porscheList);
```

Java’s unhelpful error:
```
GList<? super Car> cannot be automatically converted to GList<Car>
```

Don’t return `<? super T>`. Instead, have a upcast method.
So what’s `<? super T>` good for?

Beforehand (lambda-using methods):

```java
public <R> GList<R> map(Function<T, R> func) {
    return new GList<>(
        func.apply(value),
        tailList.map(func));
}

public void foreach(Consumer<T> func) {
    func.accept(value);
    tailList.foreach(func);
}
```

Afterward:

```java
public <R> GList<R> map(Function<<? super T, R> func) {
    return new GList<>(
        func.apply(value),
        tailList.map(func));
}

public void foreach(Consumer<<? super T> func) {
    func.accept(value);
    tailList.foreach(func);
}
If the function wants a Car and we give it a Porsche, then that’s fine. (Liskov Substitution Principle)
So what’s `<? super T>` good for?

**Beforehand (lambda-using methods):**

```java
public <R> GList<R> map(Function<T,R> func) {
    return new GList<>(
        func.apply(value),
        tailList.map(func));
}

public void foreach(Consumer<T> func) {
    func.accept(value);
    tailList.foreach(func);
}
```

**Afterward:**

```java
public <R> GList<R> map(Function<? super T,R> func) {
    return new GList<>(
        func.apply(value),
        tailList.map(func));
}

public void foreach(Consumer<? super T> func) {
    func.accept(value);
    tailList.foreach(func);
}
```

General rule: “Producer Extends, Consumer Super” (PECS)
Further reading

Very detailed FAQ:
http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#Wildcards
Wildcards in prior Comp215 code

We did this in a few places, like ITree:

default ITree<T> insertList(@NotNull IList<? extends T> values) {
    return values.foldl(ITree::insert, this);
}

So if you had a tree of Cars, you can fold in a list of Porsches.
Even list singletons...

Simpler wildcards, without type boundaries

```
private final static Empty<?> singleton = new Empty<>();

public static <T> IList<T> create() {
    @SuppressWarnings("unchecked")
    IList<T> typedEmptyList = (IList<T>)singleton;
    return typedEmptyList;
}
```

`singleton` is a list of... something... but we don’t care what.
Avoiding wildcards: the JSON parser

Did you notice that every builder has the same type signature?

```java
public static Optional<Pair<IValue, IList<Token<Scanner.TokenType>>> build(IList<Token<Scanner.TokenType>> tokenList) { ... }
```

Alternatively, each could have returned the specific type, not the IValue
But then you’d need wildcards to store these things
And you’d still have to cast to concrete types to use them

Our solution let’s you ask what you’ve got then helps you cast it
Wildcards in the wild

Here's how Java's Optional class does it

```java
public static <T> Optional<T> of(T value) { ... }

public Optional<T> filter(Predicate<? super T> predicate) { ... }

public<U> Optional<U> map(Function<? super T, ? extends U> mapper) { ... }

public T orElse(T other) { ... }
```

No explicit typecasts (except the `Optional.empty` singleton).
Wildcards in the argument type parameters, not in the return types.
Live coding

Making GList support covariance / contravariance