Covariance and Contravariance

Dan S. Wallach and Mack Joyner, Rice University
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* (≤), 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?

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

The Liskov Substitution Principle (LSP) is a concept in Object Oriented Programming that states: Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. Sep 11, 2008

What is the Liskov Substitution Principle? - Stack Overflow
stackoverflow.com/questions/…/what-is-the-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 super type is expected (Porsche is a subtype of Car.)
“Upcasting”

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!
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!
“Upcasting”

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!
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;
    }
}

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

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

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

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

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

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

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

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

class Porsche extends Car: what’s this mean?

1) A Porsche has all the same member variables as a Car (make, model, and year)
2) A Porsche has all the same methods as a Car
3) A Porsche implements all the same interfaces as a Car
4) You can use a Porsche anywhere you’d use a Car

So what’s the benefit?

1) A Porsche can add new members or methods.
2) A Porsche can override methods from Car, specializing it.
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;
    }
}

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

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

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

class Corvette extends Chevy {
    public Corvette(int year) {
        super("Corvette", year);
    }
}
Method overriding
Here's our basic car:

```java
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 void honk() {
        System.out.println("Honk!");
    }

    public void fireGuns() {
        System.out.println("Oddly, nothing happens.");
    }
}
```

Method overriding
Method overriding

Here's our basic car:

```java
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 void honk() {
        System.out.println("Honk!");
    }

    public void fireGuns() {
        System.out.println("Oddly, nothing happens.");
    }
}
```

Here's James Bond's car:

```java
class AstonMartin extends Car {
    public AstonMartin(String model, int year) {
        super("Aston Martin", model, year);
    }
}

class AstonMartinDB5 extends AstonMartin {
    public AstonMartinDB5(int year) {
        super("DB5", year);
    }
}

class BondDB5 extends AstonMartinDB5 {
    public BondDB5() {
        super(1964);
    }

    @Override
    public void fireGuns() {
        System.out.println("BANG BANG BANG");
    }
}
```
**Method overriding**

Here's our basic car:

```java
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 void honk() {
        System.out.println("Honk!");  
    }
    public void fireGuns() {
        System.out.println("Oddly, nothing happens.");
    }
}
```

Here's James Bond's car:

```java
class AstonMartin extends Car {
    public AstonMartin(String model, int year) {
        super("Aston Martin", model, year);
    }
}
class AstonMartinDB5 extends AstonMartin {
    public AstonMartinDB5(int year) {
        super("DB5", year);
    }
}
class BondDB5 extends AstonMartinDB5 {
    public BondDB5() {
        super(1964);
    }
    @Override
    public void fireGuns() {
        System.out.println("BANG BANG BANG!");
    }
}
```

Similar to the "default" methods on an interface: every Car will have these behaviors.
Method overriding

Here’s our basic car:

```java
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 void honk() {
        System.out.println("Honk!");
    }
    public void fireGuns() {
        System.out.println("Oddly, nothing happens.");
    }
}
```

Here’s James Bond’s car:

```java
class AstonMartin extends Car {
    public AstonMartin(String model, int year) {
        super("Aston Martin", model, year);
    }
}
class AstonMartinDB5 extends AstonMartin {
    public AstonMartinDB5(int year) {
        super("DB5", year);
    }
}
class BondDB5 extends AstonMartinDB5 {
    public BondDB5() {
        super(1964);
    }
    @Override
    public void fireGuns() {
        System.out.println("BANG BANG BANG");
    }
}
```

But they can be overridden, giving the new behavior.
Do you need inheritance?

A1: Surprisingly infrequently.
Before Java 8, inheritance was how you avoided repeating code. Now we can use default methods on interfaces.

A2: But still useful on occasion.
When you need to share code and data. And you can live with the restrictions.
Single inheritance!
Java lets you “implement” as many interfaces as you want. But you can only “extend” one class.
- Forces you to shove lots of functionality into the base class.
- “The fragile base class problem”

C++ allows for multiple inheritance
- But method dispatch gets crazy complicated. Java is simpler.
Why single-inheritance?

Every instance has one of its super-type instances “inside” it

If you’re expecting a pointer to a Car, you don’t know if it’s a Porsche911: same memory layout!
How does method dispatch work?

Every object has a pointer to a *vtable* (method dispatch table)

<table>
<thead>
<tr>
<th>Method</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>...</td>
</tr>
<tr>
<td>toString</td>
<td>...</td>
</tr>
</tbody>
</table>
How does method dispatch work?

Every object has a pointer to a *vtable* (method dispatch table)

<table>
<thead>
<tr>
<th>Method</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>fireGuns</td>
<td>...</td>
</tr>
<tr>
<td>honk</td>
<td>...</td>
</tr>
<tr>
<td>hashCode</td>
<td>...</td>
</tr>
<tr>
<td>toString</td>
<td>...</td>
</tr>
</tbody>
</table>
How does method dispatch work?

Every object has a pointer to a vtable (method dispatch table)

When you override a method, you’re changing the entry in the vtable.
Multiple interface implementation?

No impact on memory layout: just on method dispatch
Different internal mechanism to dispatch methods
  • Slower? Not necessarily.
If there is exactly one possible target for a call:
  • The compiler will figure it out.

Default methods in interfaces are preferable in Java8
The only downside: interfaces can’t add instance variables.
What about *lists* of objects?

(To keep things simple, we’re going back to GList from week 2.)

```java
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 BondDB5(),
    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));
```
If Porsche911 ≤ Porsche ≤ Car, then what about lists of them? 
(List of Porsche911) ≤ (List of Porsche) ≤ (List of Car), right?

If somebody expects a list of Cars, a list of Porsches is fine.
If somebody expects a list of Porsches, a list of Cars is not fine.

Definition: *Lists of T (as described) are covariant with T.*
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.
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);
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!
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!
All the built-in collections classes (java.util.*) use mutation. When they added Java generics (Java5), they kept the mutation.

If you wanted covariance, i.e.:
(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
Covariance and functional lists

If you have a list of Porsches, you cannot add a Ford.

GList<Porsche> porsches = 🚗🚗🚗
Covariance and functional lists

If you have a list of Porsches, you cannot add a Ford.

GList<Porsche> porsches = 🚗🚗🚗

Your list of Porsches is also a list of Cars.
If you have a list of Porsches, you cannot add a Ford.

GList<Porsche> porsches ← Porsches

Your list of Porsches is also a list of Cars.

If you treat it as a list of Cars, then add a Ford, you get a list of Cars.

But the original list of Porsches is still all Porsches. No mutation!

GList<Car> cars ← Cars
Java type parameters and covariance

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

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

GList<Car> castPorsches = porscheList; // fails because types are not compatible

GList<? extends Car> wildcardPorschesList = porscheList;
GList<? extends Car> wildcardPorsche911List = porsche911List;

GList<Car> mappedPorsches = porscheList.map(x -> x); // works, but inefficient
GList<Car> narrowPorsches = narrow(porscheList); // much nicer
GList<Porsche> narrowPorsches2 = narrow(porsche911List); // also works
Java type parameters and covariance

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

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

GList<Car> castPorsches = porscheList;
// fails because types are not compatible

GList<? extends Car> wildcardPorscheList = porscheList;
GList<? extends Car> wildcardPorsche911List = porsche911List;

GList<Car> mappedPorsches = porscheList.map(x -> x);
// works, but inefficient

GList<Car> narrowPorsches = narrow(porscheList);
// much nicer

GList<Porsche> narrowPorsches2 = narrow(porsche911List);
// also works

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

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

GList<Porsche911> porsche911list = GList.of(
    new Porsche911(1991),
    new Porsche911(2015));

GList<Car> castPorsches = porscheList;  // fails because types are not compatible

GList<? extends Car> wildcardPorscheList = porscheList;
GList<? extends Car> wildcardPorsche911list = porsche911list;

GList<Car> m   Wildcard type: matches Car and subtypes of Car  \( x \rightarrow x \);  // works, but inefficient
GList<Car> narrowPorsches = narrow(porscheList);  // much nicer
GList<Porsche> narrowPorsches2 = narrow(porsche911list);  // also works
```
Java type parameters and covariance

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

GList\<Porsche911\> \texttt{porsche911list} = GList.of(
    new Porsche911(1991),
    new Porsche911(2015));

GList\<Car\> \texttt{castPorsches} = \texttt{porscheList}; // fails because types are not compatible

GList\?<\ extends \ Car\> \texttt{wildcardPorscheList} = \texttt{porscheList};
GList\?<\ extends \ Car\> \texttt{wildcardPorsche911list} = \texttt{porsche911list};

GList\<Car\> \texttt{mappedPorsches} = \texttt{porscheList}.map(x -> x); // works, but inefficient
GList\<Car\> \texttt{narrowPorsches} = \texttt{narrow(\texttt{porscheList})}; // much nicer
GList\<Porsche\> \texttt{narrowPorsches2} = \texttt{narrow(\texttt{porsche911list})}; // also works
Explain `map(x->x)` please?

Here's the original code:
```java
gList<Porsche> porscheList = ...
gList<Car> mappedPorsches = porscheList.map(x -> x);
```

Map can change types (e.g., map `String::length` onto a list of strings)
```java
Q IList<Q> map(Function<T, Q> f);
```

In this example, Java infers that `T` is `Porsche`, `Q` is `Car`

`x->x` is a lambda of type `Function<Porsche, Car>`

And, via the Liskov Substitution Principle, this code works.
Here's the original code:

```java
GList<Porsche> porscheList = ...
GList<Car> narrowPorsches = narrow(porscheList);
```

Narrow is just a method on the list type (imported statically):

```java
static <T> GList<T> narrow(GList<? extends T> list) {
    @SuppressWarnings("unchecked")
    GList<T> result = (GList<T>) list;
    return result;
}
```

We accept a broader type (GList of anything that extends `T`)

We return a narrower type (GList of `T`)

And, via the Liskov Substitution Principle, it's safe

If the “map” version works, then it’s safe to do this instead (constant time!)

But only on a functional list, no mutation!
Explain “narrow” please?

Here’s the original code:

```java
GList<Porsche> porscheList = ...
GList<Car> narrowPorsches = narrow(porscheList);
```

Narrow is just a method on the list type (imported statically):

```java
static <T> GList<T> narrow(GList<? extends T> list) {
    @SuppressWarnings("unchecked")
    GList<T> result = (GList<T>) list;
    return result;
}
```

We accept a broader type (GList of anything that extends `T`)
We return a narrower type (GList of `T`)
And, via the Liskov Substitution Principle, it’s safe
If the “map” version works, then it’s safe to do this instead (constant time!)
But only on a functional list, no mutation!

At runtime, Java doesn’t know the type parameters. If the map version is safe, so is the fast “narrow” version.
Adding covariance to GList

Beforehand (constructor):
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<T> tailList) {
        this.head = value;
        this.tail = tailList;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<? extends T> tailList) {
        this.head = value;
        this.tail = narrow(tailList);
    }
}
Adding covariance to GList

Beforehand (constructor):
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<T> tailList) {
        this.headVal = value;
        this.tailVal = tailList;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<? extends T> tailList) {
        this.headVal = value;
        this.tailVal = narrow(tailList);
    }
}
```

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

Beforehand (constructor):
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<T> tailList) {
        this.headVal = value;
        this.tailVal = tailList;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<? extends T> tailList) {
        this.headVal = value;
        this.tailVal = narrow(tailList);
    }
}
```

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

Beforehand (constructor):
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<T> tailList) {
        this.headVal = value;
        this.tailVal = tailList;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    private final T headVal;
    private final GList<T> tailVal;

    private Cons(T value, GList<? extends T> tailList) {
        this.headVal = value;
        this.tailVal = narrow(tailList);
    }
}
```

Safe because we’re doing functional lists.
Beforehand (list concatenation):
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return other;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return narrow(other);
    }
}
```
Input parameters should be broad

Beforehand (list concatenation):
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return other;
    }
}
```

Afterward:
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return narrow(other);
    }
}
```

Wildcard type parameters to accept subtypes.
Beforehand (list concatenation):

```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return other;
    }
}
```

Afterward:

```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return narrow(other);
    }
}
```

Convert `GList<? extends T>` to `GList<T>` (Which we know is safe.)
Beforehand (list concatenation):
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<T> other) {
        return other;
    }
}
```

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

Afterward:
```java
class Cons<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return tailVal.concat(other).add(headVal);
    }
}
class Empty<T> implements GList<T> {
    public GList<T> concat(GList<? extends T> other) {
        return narrow(other);
    }
}
```
What about contravariance?

For “read” types (e.g., the arguments of concat) we want covariance

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

For “write” types (e.g., the return value) we want contravariance

You can assign a list of Porsche to a variable typed “list of Cars”

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

For “read” types (e.g., the arguments of concat) we want covariance

\[
\text{GList}\langle\text{Porsche}\rangle \quad \text{morePorsches} = \text{porscheList}.\text{concat(}\text{porsche911list}\text{)};
\]

For “write” types (e.g., the return value) we want contravariance

You can assign a list of Porsche to a variable typed “list of Cars”

\[
\text{GList}\langle\text{Car}\rangle \quad \text{moreCars} = \text{narrow(}\text{porscheList}.\text{concat(}\text{porsche911list}\text{)});
\]
What about contravariance?

For “read” types (e.g., the arguments of concat) we want covariance

\[
\text{GList<Porsche> morePorsches = porscheList.concat(porsche911list);} \\
\]

For “write” types (e.g., the return value) we want contravariance

You can assign a list of Porsche to a variable typed “list of Cars”

\[
\text{GList<Car> moreCars = narrow(porscheList.concat(porsche911list));} \\
\]

Contravariance on the output: we can use a broader type
What about contravariance?

For “read” types (e.g., the arguments of concat) we want covariance

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

For “write” types (e.g., the return value) we want contravariance

You can assign a list of Porsche to a variable typed “list of Cars”

```java
GList<Car> moreCars = narrow(porscheList.concat(porsche911list));
```

Annoying: do we need to explicitly `narrow` the type?
What about contravariance?

narrow is a static method on GList

```
static <T> GList<T> narrow(GList<? extends T> list) { ... }
```

In this case, <T> is Car, and <? extends T> is Porsche
Java automatically found a T which makes narrow work!

```
GList<Car> moreCars = narrow(porscheList.concat(porsche911list));
```

Annoyng: do we need to explicitly narrow the type?
Consumers vs. producers

Beforehand (map, filter):

```java
<R> GList<R> map(Function<T, R> f);
GList<T> filter(Predicate<T> predicate);
```

Afterward:

```java
<R> GList<R> map(Function<? super T, ? extends R> f);
GList<T> filter(Predicate<? super T> predicate);
```
Consumers vs. producers

Beforehand (map, filter):

```java
<R> GList<R> map(Function<T, R> f);
GList<T> filter(Predicate<T> predicate);
```

If we’re passing a Porsche and the function can handle any Car, that’s fine. (Liskov Substitution Principle)

Afterward:

```java
<R> GList<R> map(Function<? super T, ? extends R> f);
GList<T> filter(Predicate<? super T> predicate);
```
Consumers vs. producers

Beforehand (map, filter):

```java
<R> GList<R> map(Function<T, R> f);
GList<T> filter(Predicate<T> predicate);
```

If we're expecting the function to return a Car and we're given a Porsche, that's fine as well.

Afterward:

```java
<R> GList<R> map(Function<? super T, ? extends R> f);
GList<T> filter(Predicate<? super T> predicate);
```
Consumers vs. producers

Beforehand (map, filter):
\[
\langle R \rangle \ \text{GList} \langle R \rangle \ \text{map}(\text{Function}\langle T, R \rangle \ f);
\text{GList}\langle T \rangle \ \text{filter}(\text{Predicate}\langle T \rangle \ \text{predicate});
\]

Afterward:
\[
\langle R \rangle \ \text{GList} \langle R \rangle \ \text{map}(\text{Function}\langle ? \text{super} \ T, ? \text{extends} \ R \rangle \ f);
\text{GList}\langle T \rangle \ \text{filter}(\text{Predicate}\langle ? \text{super} \ T \rangle \ \text{predicate});
\]

General rule: “Producer Extends, Consumer Super” (PECS)
Absent wildcards

Last year’s Comp215: wildcards were there from the beginning
We told the students to ignore them until after the midterm.

This year, I rewrote everything ("Comp215 standard style")
No need for class inheritance anywhere!
Notice that every `make`-method has the same signature:

```java
private static final IList<Function<IList<Token<JsonPatterns>>, Option<Result<Value>>> MAKERS = List.of(
    Parser::makeString,
    Parser::makeNumber,
    Parser::makeObject,
    Parser::makeArray,
    Parser::makeBoolean,
    Parser::makeNull);
```

```java
static Option<Result<Value>> makeString(IList<Token<JsonPatterns>> tokenList) { ... }
static Option<Result<Value>> makeNumber(IList<Token<JsonPatterns>> tokenList) { ... }
static Option<Result<Value>> makeObject(IList<Token<JsonPatterns>> tokenList) { ... }
```

Everything takes and returns `Value` rather than `JObject`, `JArray`, etc.

Liskov Substitution Principle still works

No need for shared members in the classes
Rules of thumb: when to wildcard

For a general-purpose library (lists, trees, etc.):
You’ll have users with class inheritance, so you should support it. (But, you’re more likely to see it in “legacy” Java code than new Java8.)

For a closed-world package (like the JSON parser):
You don’t necessarily need class inheritance; keep things simple!

Wildcards belong on type parameters for method arguments

Wildcards should not be used on return values or most variables
Narrow the type, make life easier for your callers.
Postel’s Law

(Also called the “Robustness Principle”)

“Be liberal in what you accept, be conservative in what you send.”

- Computers on networks
- Functions in a program
Further reading

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

When reading the FAQ, note that `java.util` classes support mutation
  · No mutation in `edu.rice.list.*`, so we can have a `narrow` operator.
Further reading

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

When reading the FAQ, note that `java.util` classes support mutation
  • No mutation in `edu.rice.list.*`, so we can have a narrow operator.
This week’s code has wildcards everywhere!
Start off by reading the code in `edu.rice.week9covariance`.
Then, once you’re comfortable, check out `edu.rice.list.*`, etc.

You won’t need wildcards for this week’s RPN project
But you might want to use it starting next week.