

# <Grokking Generics>

Daniel Hinojosa

# About Me...

Testing in Scala  
(Book)

Beginning Scala Programming  
(O'Reilly Video)

Scala Beyond the Basics  
(O'Reilly Class)

TDD in Java  
(O'Reilly Class)



@dhinojosa



[dhinojosa@evolutionnext.com](mailto:dhinojosa@evolutionnext.com)



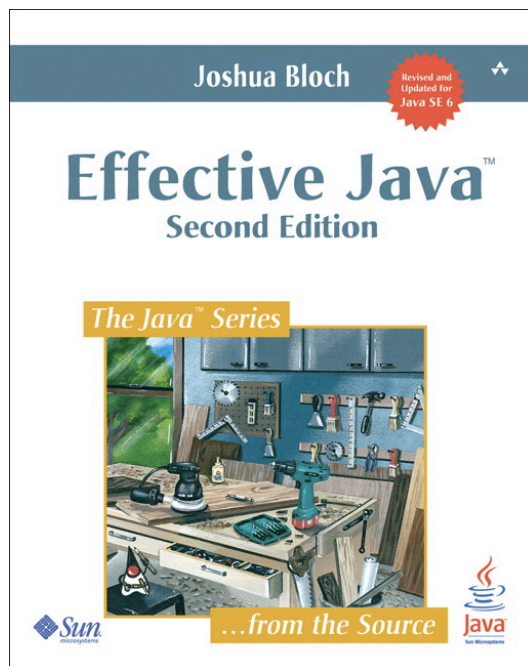
Code & Slides Available @

[https://github.com/dhinojosa/  
generics-study](https://github.com/dhinojosa/generics-study)

# Generics

- Add stability to your code by exposing bugs at compile time
- Provides a way for you to re-use the same code with different inputs allowing for less code
- Eliminates casting
- Eliminates the need for explicit boxing
- One of the harder concepts since JDK 5

# Effective Java Item 26



## Item 26: Favor generic types

It is generally not too difficult to parameterize your collection declarations and make use of the generic types and methods provided by the JDK. Writing your own generic types is a bit more difficult, but it's worth the effort to learn how.

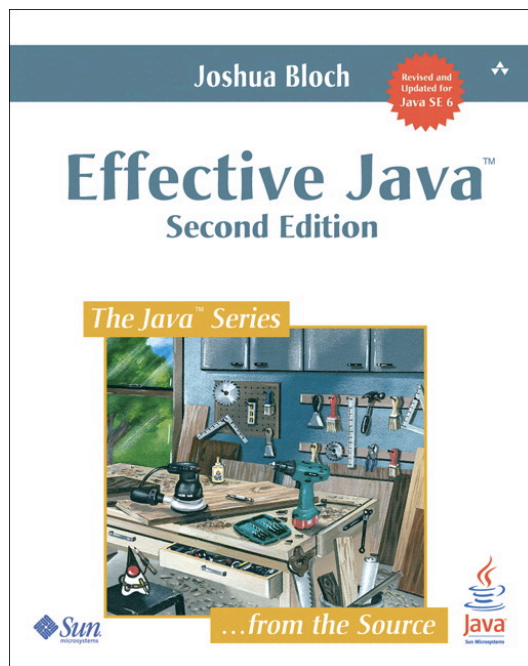
# Generic Terms

Term	Example
Parameterized Type	<code>List&lt;String&gt;</code>
Actual Type Parameter	<code>String</code>
Generic Type	<code>List&lt;E&gt;</code>
Formal Type Parameter	<code>E</code>
Unbounded wildcard type	<code>List&lt;?&gt;</code>
Raw Type	<code>List</code>
Bounded Type Parameter	<code>&lt;E extends Number&gt;</code>
Recursive Type Bound	<code>&lt;T extends Comparable&lt;T&gt;&gt;</code>
Bounded Wildcard Type	<code>List&lt;? extends Number&gt;</code>
Generic Method	<code>static &lt;E&gt; List&lt;E&gt; asList(E[] a)</code>
Type Token	<code>Integer.class</code>

Source: Effective Java 2nd Edition

# Demo: Casting & Boxing

# Effective Java Item 23



## Item 23: Don't use raw types in new code

First, a few terms. A class or interface whose declaration has one or more *type parameters* is a *generic* class or interface [JLS, 8.1.2, 9.1.2]. For example, as of release 1.5, the `List` interface has a single type parameter, `E`, representing the element type of the list. Technically the name of the interface is now `List<E>` (read “list of `E`”), but people often call it `List` for short. Generic classes and interfaces are collectively known as *generic types*.



# Erasure

# Erasure

- Parameterized Types are *erased* at runtime, this is called *type erasure*
- This can trip up the novice and professional alike
- `List<String>` at runtime is purely `List`
- Generic types are non-reifiable, meaning they are not fully available at runtime

# Demo: Erasure

# **Generic Classes/Generic Instance and Static Methods**

# Assignment of Generic Types

# Assignment of Generic Types

- As default, `List<Integer>` can only be assigned to a `List<Integer>`
- Special Consideration for assignments where `List<Integer>` gets assigned a `List<Object>`
- ...or the other way around: `List<Object>` gets assigned a `List<Integer>`

polymorphism

**Number**

←  
<<assignment>>

**Integer**

invariant

**List<Integer>**

←  
<<assignment>>

**List<Integer>**

covariant

**List<Number>**

←  
<<assignment>>

**List<Integer>**

contravariant

**List<Integer>**

←  
<<assignment>>

**List<Number>**

polymorphism

**Number**

←  
<<assignment>>

**Integer**

invariant

**List<Integer>**

←  
<<assignment>>

**List<Integer>**

covariant

**List<? extends Number>**

←  
<<assignment>>

**List<Integer>**

contravariant

**List<? super Integer>**

←  
<<assignment>>

**List<Number>**



What you can take  
out...

What you can  
put in...



```
public A foo(A a) {...}
```

# PECS

Producer-extends  
Consumer-super


producer

consumer



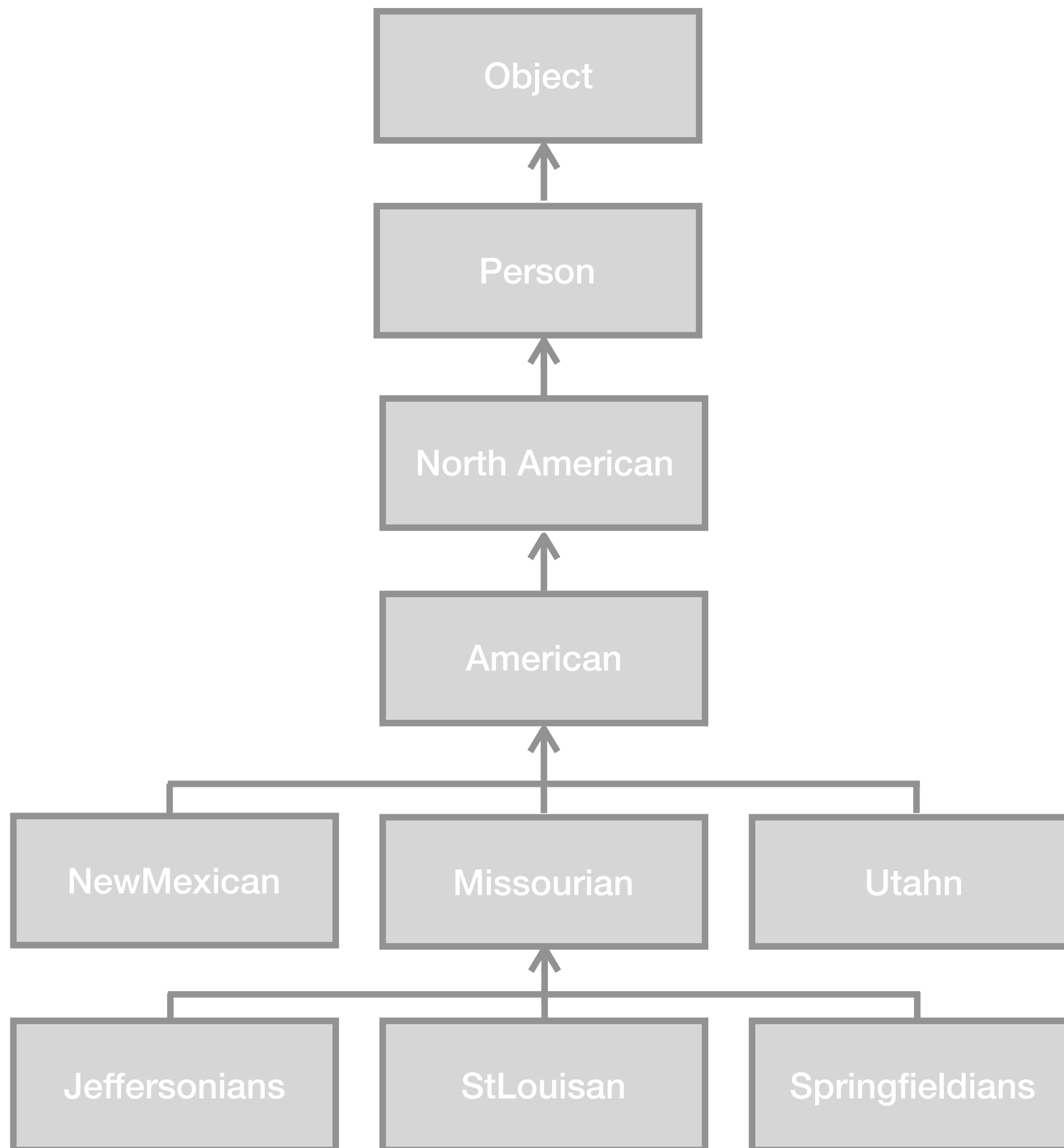
```
public A foo(A a){...}
```

Consumer      Producer



```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

Naftalin and Wadler call it the *Get and Put Principle*

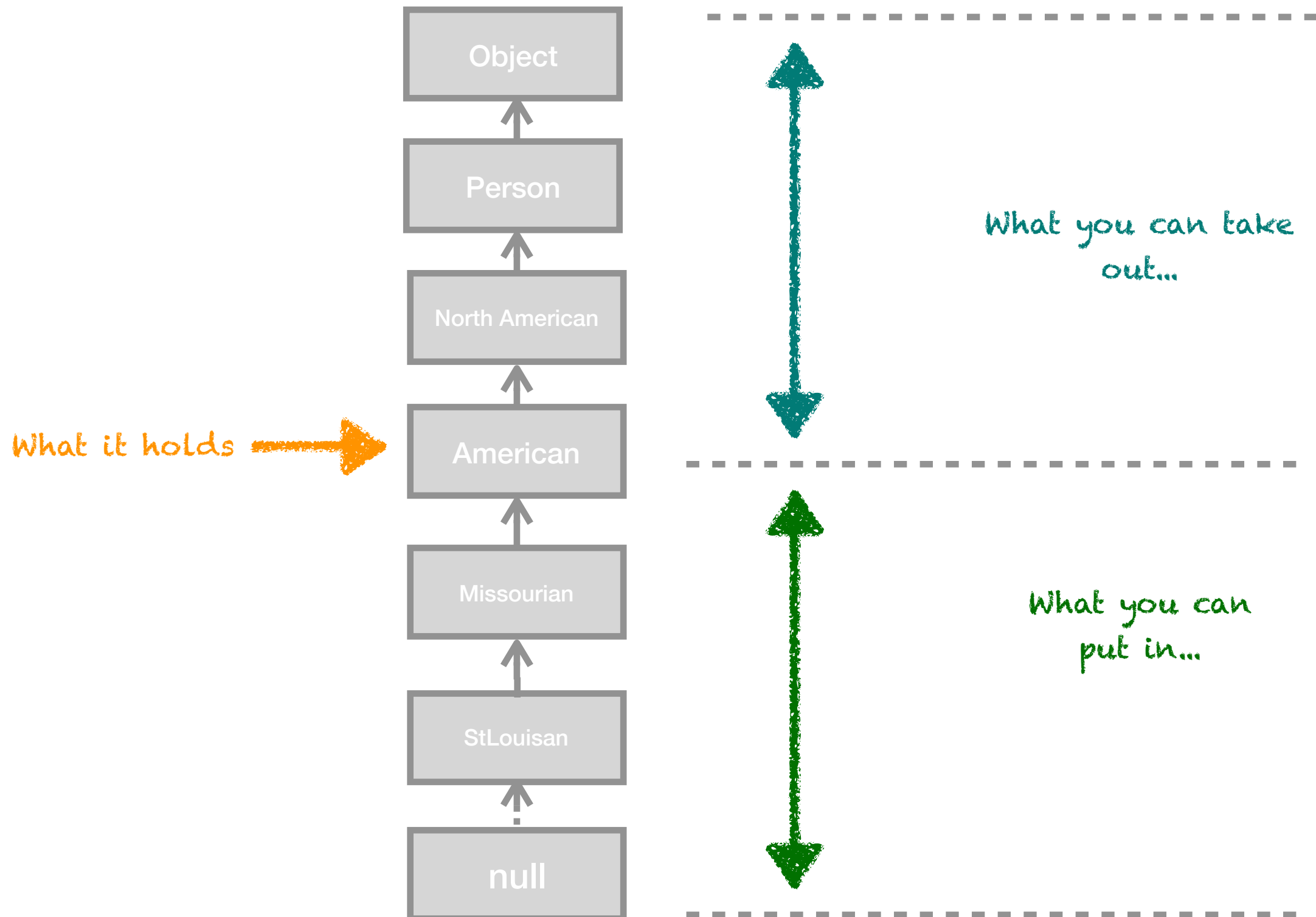


# Invariant

invariant

**List<Integer>**  $\xleftarrow{\text{assignment}}$  **List<Integer>**

**Container<American> c = new Container<American>**




# Demo: Invariant



# Covariant

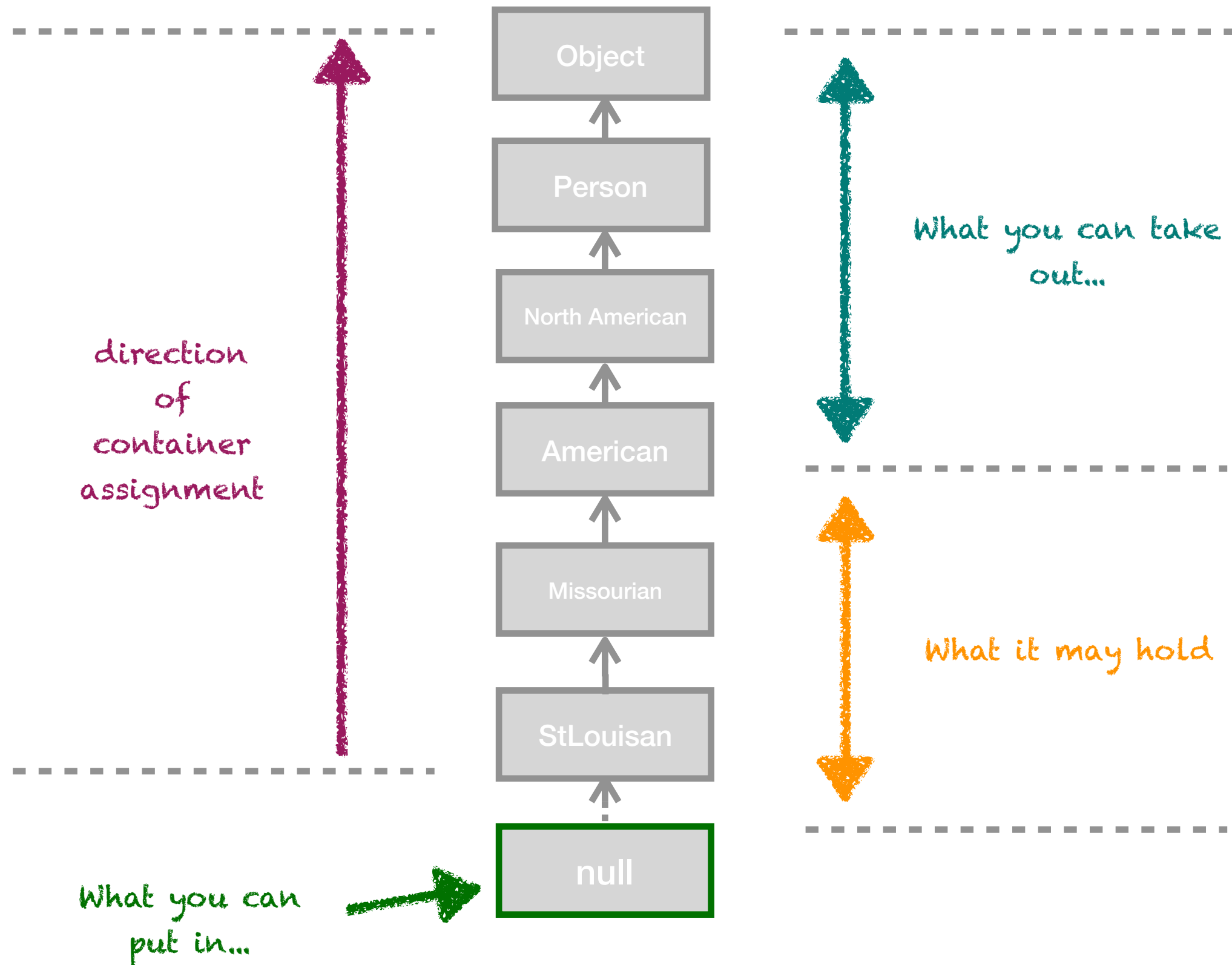
covariant

**List<? extends Number>**  **List<Integer>**  
<<assignment>>

StLouisian

? extends American

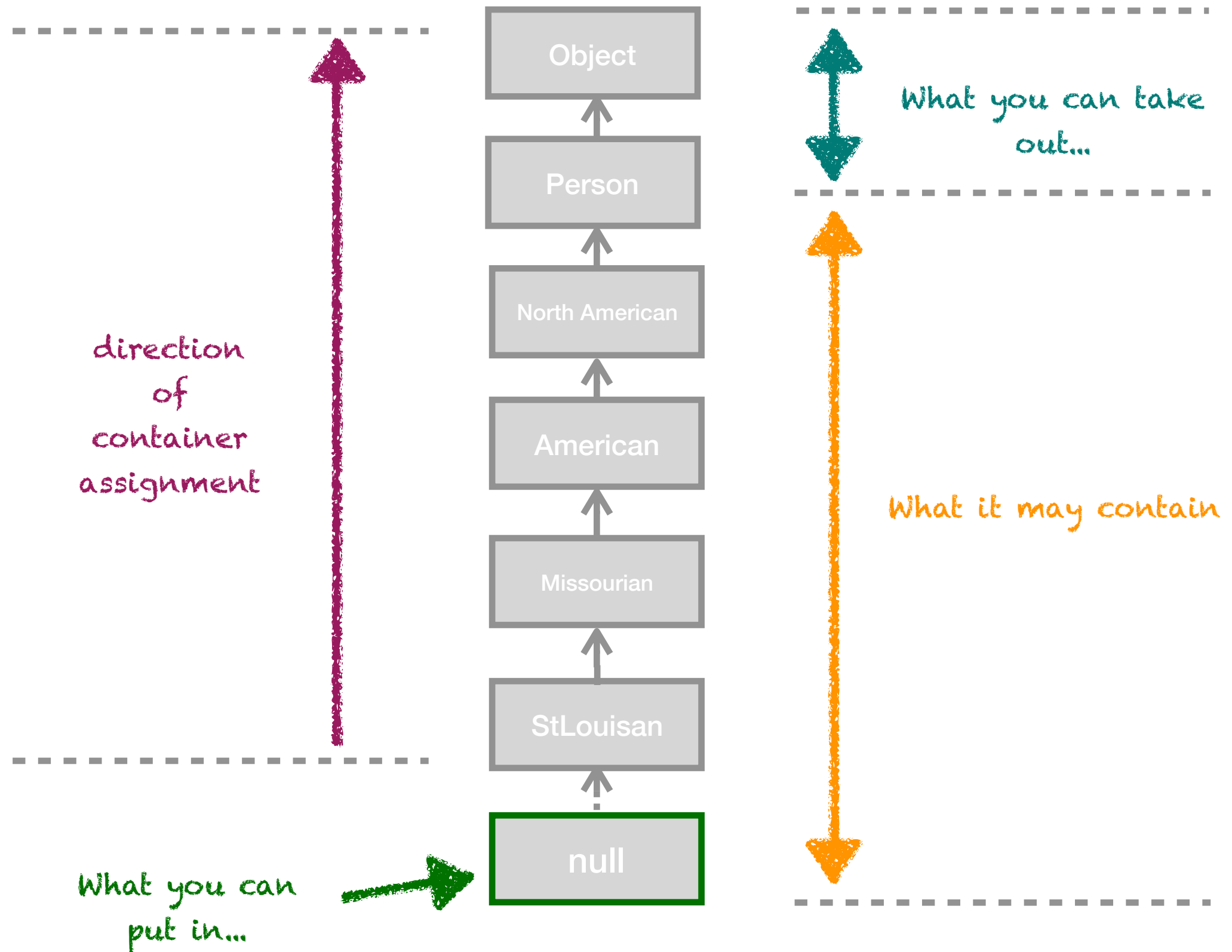
**Container<? extends American> c = new Container<StLouisian>**



Missourian

? extends Person

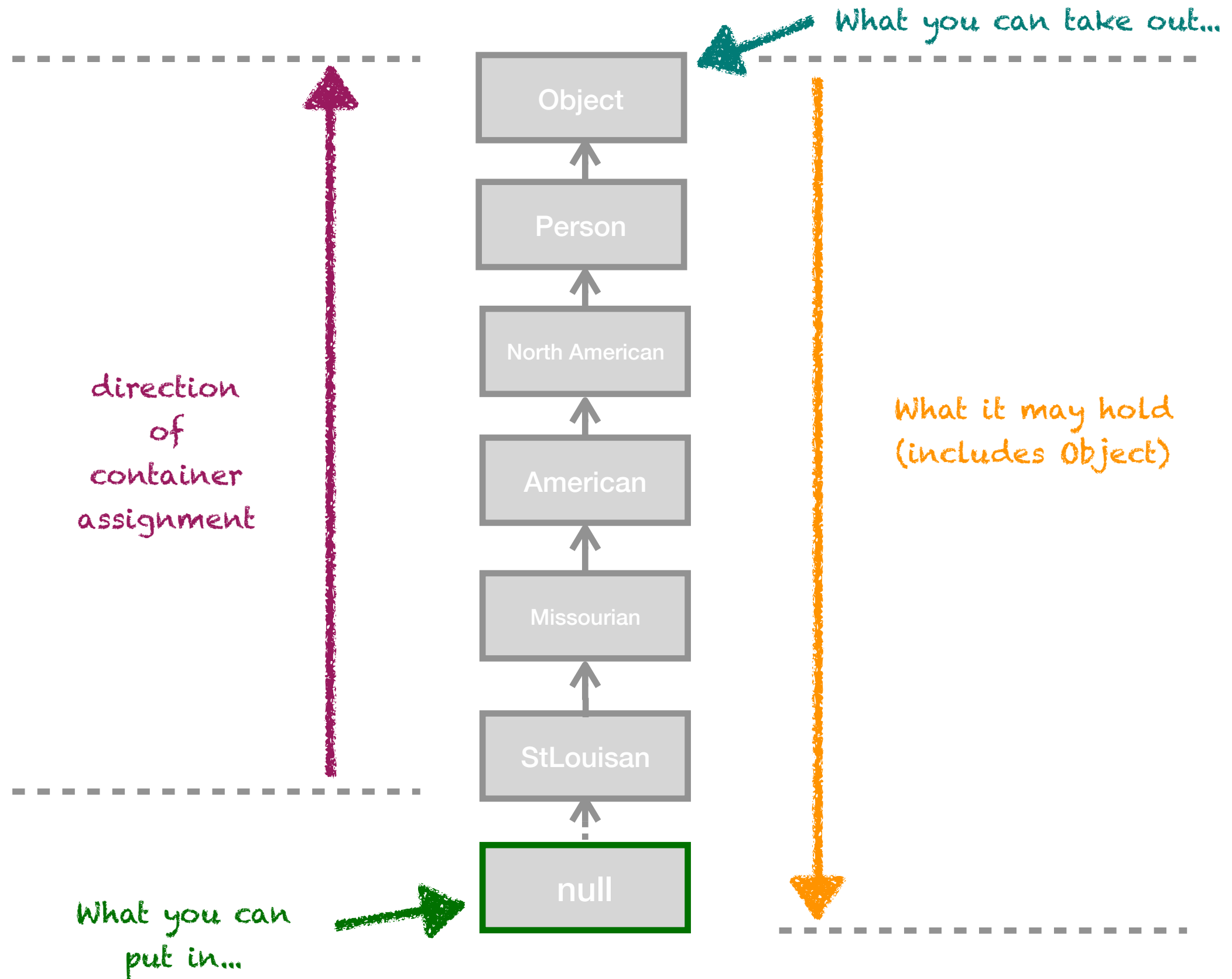
```
Container<? extends Person> c = new Container<Missourian>();
```



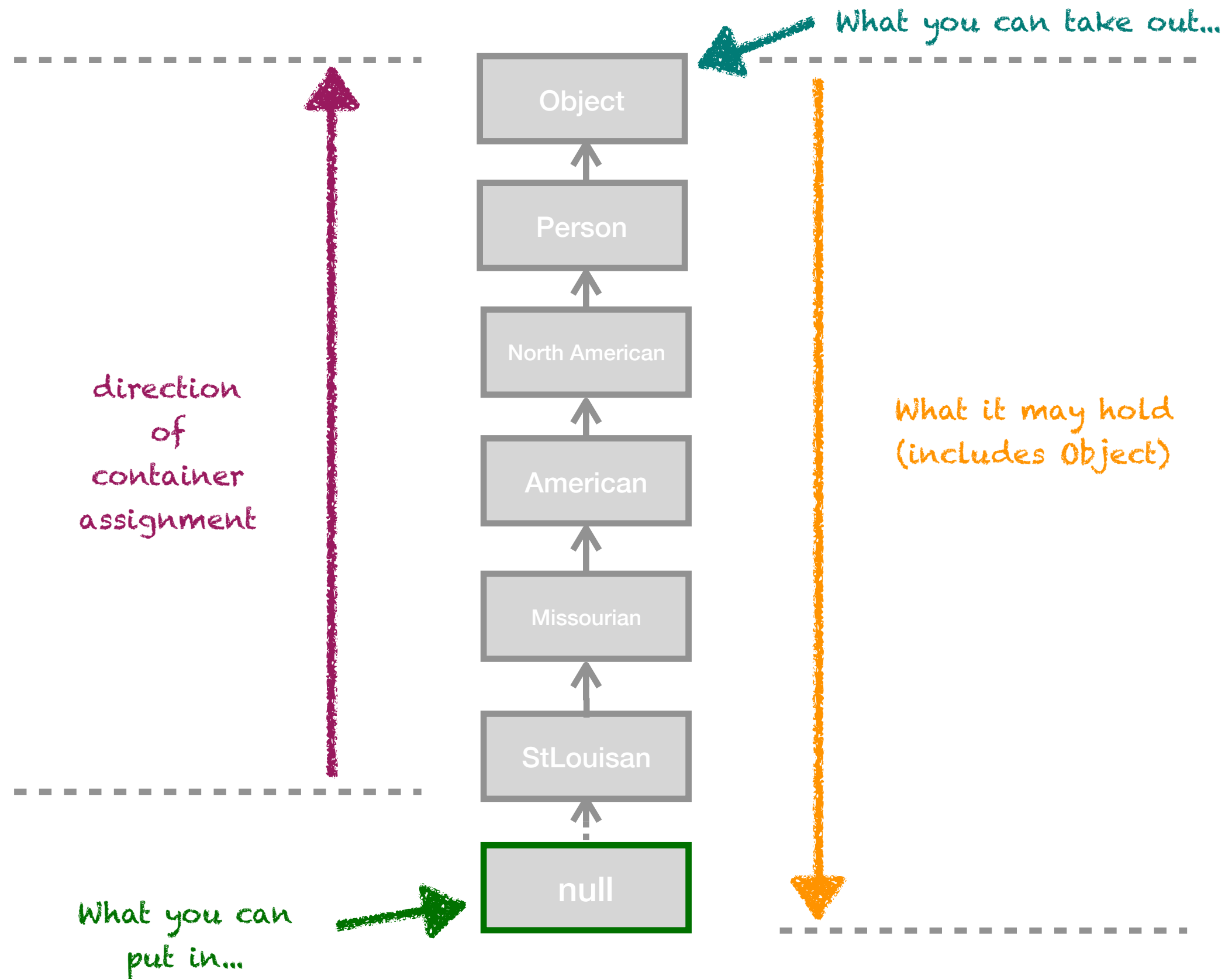
StLouisian

? extends Object

```
Container<? extends Object> c = new Container<StLouisian>();
```



```
Container<?> c = new Container<StLouisian>();
```






# Demo: Covariant

# Contravariance

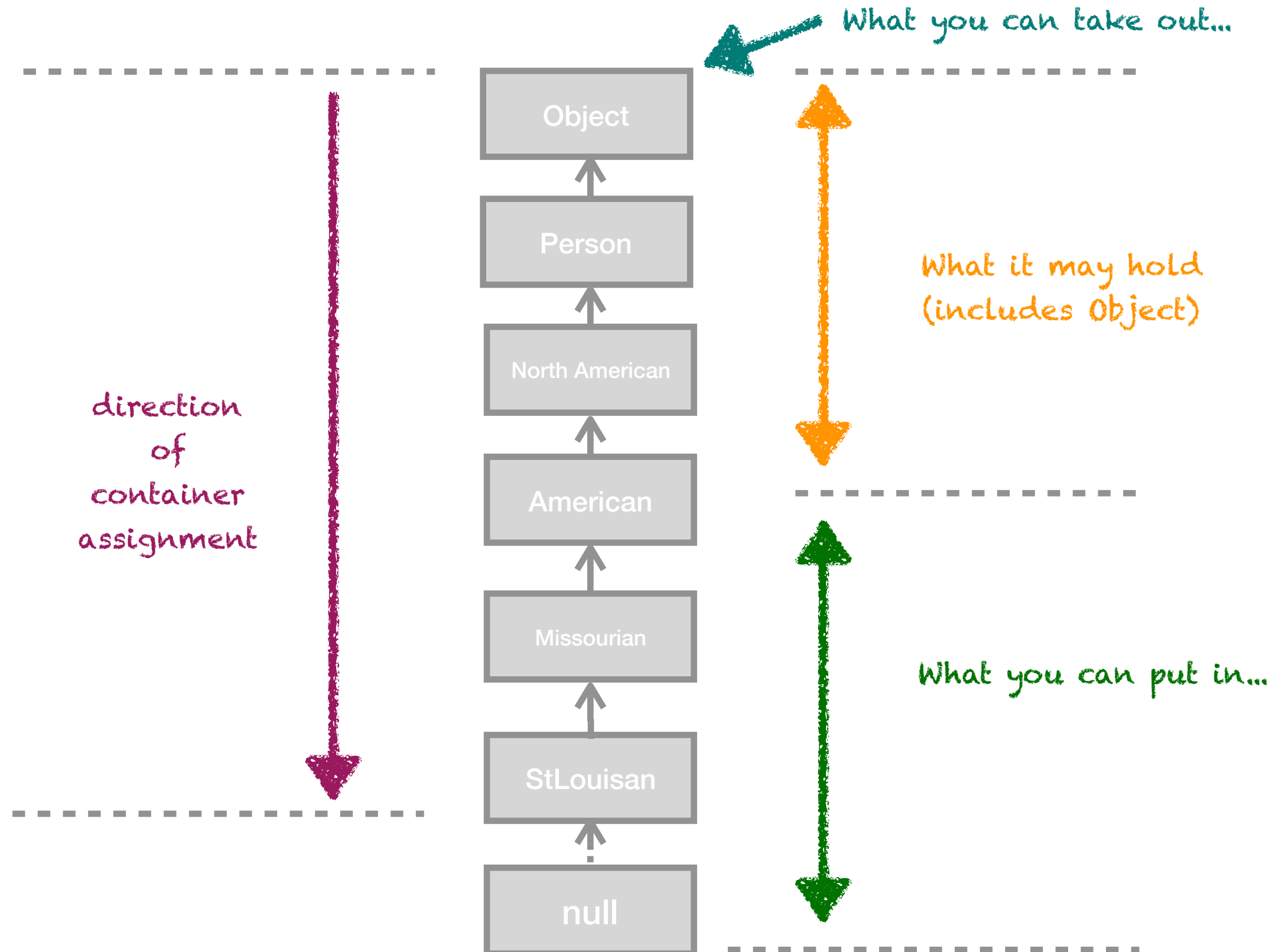
contravariant

**List<? super Integer>**  **List<Number>**  
<<assignment>>

Person

? super American

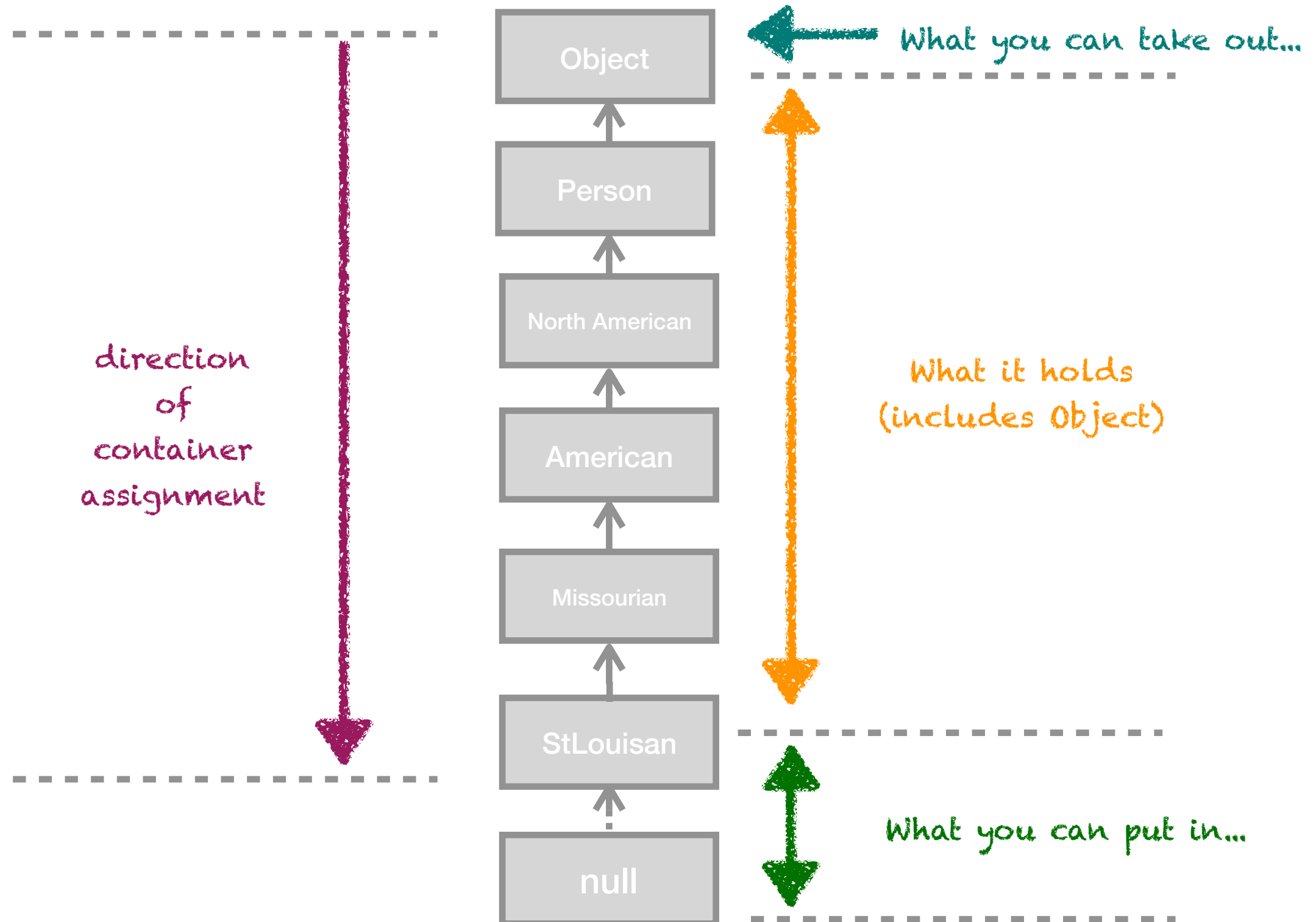
```
Container<? super American> c = new Container<Person>();
```



Person

? super StLouisian

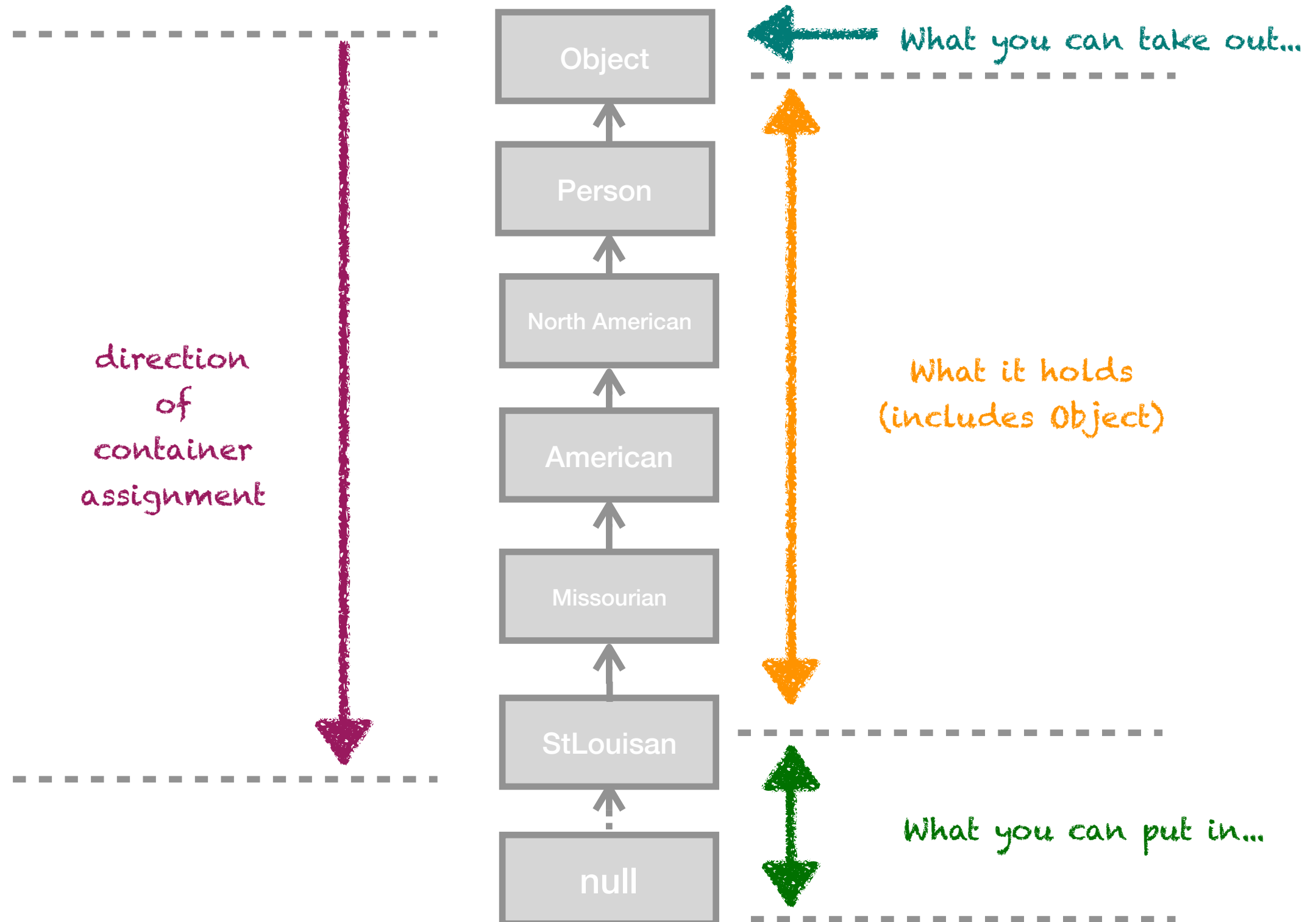
```
Container<? super StLouisian> c = new Container<Person>();
```



American

? super StLouisian

```
Container<? super StLouisian> c = new Container<American>();
```





# Demo:

# Contravariance

# Array Covariance

# Array Covariance

- Arrays are covariant
- `Array<Integer>` can be assigned to `Array<Number>` reference.
- Any attempt to do put in something wrong will be met with an `ArrayStoreException`
- An array of `List<String>` at runtime is purely an array of `List`

# Demo: Array Covariance

# SafeVargs & HeapPollution

# Heap Pollution

- A situation where a variable of a parameterized type refers to an object that is not of that parameterized type.
- This leads likely to a `ClassCastException`
- Occurs by
  - Mixing Raw Types and Parameterized Types
  - Performing Unchecked Casts
  - Separate Compilation of Transaction Units

# Demo: Heap Pollution

# SafeVargs

- Annotation that marks that the developer is not performing an unsafe operation using varargs
- Particularly committing heap pollution
- Reminder! Arrays are covariant



# Demo: SafeVarArgs

# Subclassing Generics & The Bridge Method

# Subclassing Generics

- A class that is generic `MyClass<T>` can be extended
- The Generic Type can either be
  - Fulfilled by the subclass
  - Kept Generic

# Demo: Subclassing

# Bridge Method

- At times, the compiler may create a synthetic bridge method to add support for the super classes method type.
- This is used to avoid unsafe method calls and `ClassCastException`

# Demo: The Bridge Method

# Multiple Bounds

# Multiple Bounds

- A type parameter can have multiple bounds:  
`<T extends B1 & B2 & B3>`
- If one of the bounds is a class, then it should be first:  
`class A {...}`  
`interface B {...}`  
`interface C {...}`  
  
`class D <T extends A & B & C> {...}`



# Demo: Multiple Bounds

# **Recursive Type Bound**

# Recursive Type Bound

- Bound by an expression involving the type parameter itself
- Usually involves `Comparable<T>` or `Class<T>`
- For example: `String extends Comparable<String>`

# Demo: Recursive Type Bound



- Declaration Site Variance using – and +
- Reification of types using TypeTag universe
- Use of type classes to change behavior
- No raw type collections

# Demo: Scala Parameterized Types



- Declaration Site Variance using `in` and `out`
- Reification of types using `inline` and `reified`
- Type projections to limit behavior
- No raw type collections

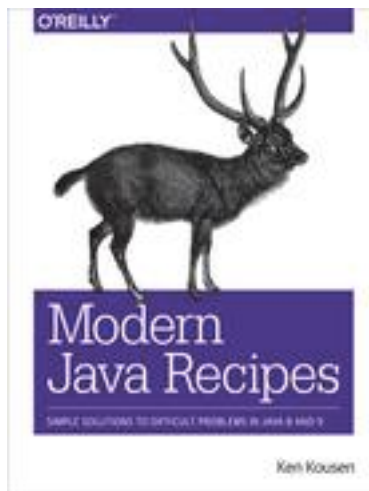
# **Demo: Kotlin Parameterized Types**



# References/Credit



Java Generics and Collections  
by Maurice Naftalin; Philip Wadler  
Published by O'Reilly Media, Inc., 2006

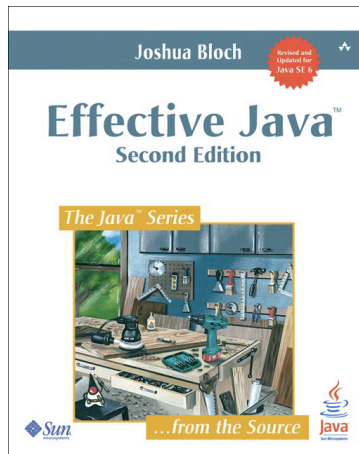


Modern Java Recipes  
by Ken Kousen  
Published by O'Reilly Media, Inc., 2017



Angelika Langer  
Java Generics FAQs - Frequently Asked Questions  
<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>

# References/Credit



Java Generics and Collections  
by Maurice Naftalin; Philip Wadler  
Published by O'Reilly Media, Inc., 2006



Programming Kotlin  
by Stephen Samuel; Stefan Bocutiu  
Published by Packt Publishing, 2017



**Java Generics Tutorial**

**<https://docs.oracle.com/javase/tutorial/java/generics/>**

**Thanks**