



Generics

You never know what you're gonna get.

Bram Janssens





by Bram Janssens



**“I'm lost
in a forest”**

<https://www.nporadio2.nl/nieuws/top2000/47902840-10d7-4bb8-96fd-509faa351cce/met-a-forest-vond-the-cure-zijn-identiteit>



Forrest Gump (1994) - Paramount Pictures — all screenshots in these slides are taken from Netflix.

My mama always told me...

```
void main() {  
    life = like.a(box).of(chocolates);  
}
```

```
> java nl.bramjanssens.RunForrest
```

You never know what you're gonna get.

– Forrest Gump





Generics – You Never Know What You're Gonna Get.

– Bram Janssens

What could go wrong?

Required type: `capture of ? extends Candy`

Provided: `GummyBear`

Required type: `capture of ? extends Chocolate`

Provided: `capture of ? extends Chocolate`

both methods have same erasure

Cannot access class object of a type parameter

Type parameter 'T' cannot be instantiated directly

```
List<byte> bites;
```

Type argument cannot be of a primitive type

Dealing with smart colleagues



🧐 “You should make it covariant”.

🤔 “Generics are implemented with type erasure.”

🤔 “This is use site variance.”



SH T HAPPENS

A close-up shot of the rear of a green truck. The license plate area is a silver metal plate with a diamond plate texture. The text 'SH T HAPPENS' is printed in bold, black, sans-serif capital letters. The truck is parked on a wet street, and a yellow car is visible in the background.



Actually, it's quite
simple!



Bram Janssens
Trainer/consultant

infoSupport
Solid Innovator



Now we ain't strangers anymore.

– F. Gump

Topics



- Types
 - Type systems
 - Classes and types
 - Generics
 - Subtypes
- Type erasure
- Variance
- Breaking type safety





Types



- Why?
- Benefits
 - Reliability
 - Adds complexity

Type systems



- Weak and dynamic typing
- Strong and static typing

Weak and dynamic typing



- Data *may* have a type (weak)
- The type can change (dynamic)

JavaScript, PHP, Python, ...

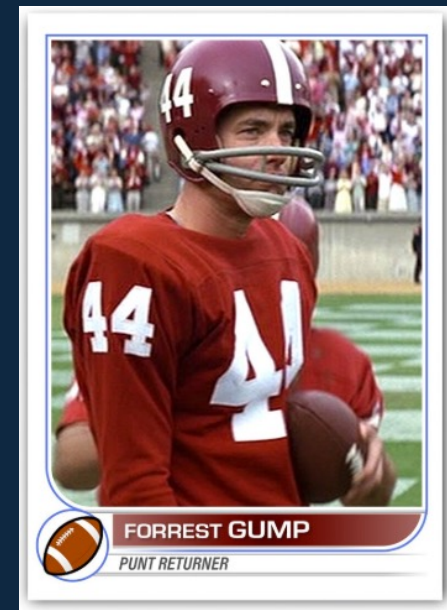
Weak and dynamic typing

JavaScript



```
let myVar = 44;  
console.log(`myVar is a ${typeof myVar}: ${myVar}`);  
  
myVar = "I am a string now";  
console.log(`myVar is a ${typeof myVar}: ${myVar}`);
```

```
myVar is a number: 44  
myVar is a string: I am a string now
```



Weak and dynamic typing

JavaScript



```
console.log("44" - 2)  
console.log("42" + 2)
```

```
42  
422
```

```
console.log("44" - null + 2)  
console.log("42" + null + 2)
```

```
46  
42null2
```

Duck typing

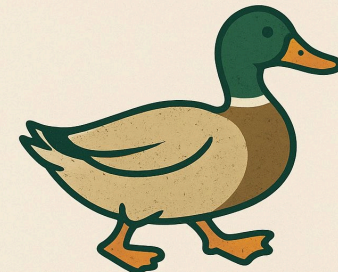
JavaScript

```
class Duck { sound = "Quack!" }  
class Cat { age = 44 }  
  
function makeSomeNoise(maybeDuck) {  
  console.log("Make some noise: " + maybeDuck.sound);  
}  
  
makeSomeNoise(new Duck())  
makeSomeNoise({sound: "I'm a shrimp..."})  
makeSomeNoise({sound: "Is this nonsense?", nonsense: true})  
makeSomeNoise(new Cat())
```

```
Make some noise: Quack!  
Make some noise: I'm a shrimp...  
Make some noise: Is this nonsense?  
Make some noise: undefined
```



**IF IT WALKS
LIKE A DUCK
AND QUACKS
LIKE A DUCK**



**IT'S PROBABLY
A DUCK**

... could lead to runtime errors



```
Uncaught TypeError: Cannot read properties of undefined (reading 'street')
```

```
Uncaught TypeError: doSomething is not a function
```



Strong and static typing



- Data *must* have a type (strong)
- The type cannot change (static)

Java, C#, Kotlin, ...



Create a type and use it



```
// 1. Declaration site 📍  
public class Chocolate { }  
//      class ^^^^^
```

```
// 2. Use site 📍  
void smell(Chocolate c) { }  
//      type  ^^^^^
```

Class → type



<u>Class</u>	<u>yields type</u>	<u>Relationship</u>
Chocolate	Chocolate	1 -> 1
Box<T>	Box<Candy> Box<Chocolate> Box<Shrimp>	1 -> *

So class and type are not exactly the same...



Speaking about generics...

Generic



- [juh · **neh** · ruhk]
 - A whole group of similar things, rather than one particular thing.

Generics



- **Generic thingS**
- **Generic typeS** 🧐
 - Type in a type
 - Parameterized types
 - `C<T>` is a container of type `C` with content of type `T`
 - `List<E>` `Optional<T>` `Box<C>`

Benefit – type safety



```
public static void useRawTypes() {
    List naturalNumbers = new ArrayList();

    // We can add anything...
    naturalNumbers.add(1);
    naturalNumbers.add("2"); // allowed
    naturalNumbers.add(3);

    // We get Object, not int
    Object first = naturalNumbers.getFirst();
    Object middle = naturalNumbers.get(1);
    Object last = naturalNumbers.getLast();

    // We have to cast them...
    int sum = (int) first + (int) last;

    // Could be unsafe...
    int kaboom = (int) middle; // ***
}
```

```
public static void useGenericTypes() {
    List<Integer> naturalNumbers = new ArrayList<>();
    // ^^^^^ = type argument

    // We can add Integers only...
    naturalNumbers.add(1);
    // naturalNumbers.add("2"); // not allowed
    naturalNumbers.add(3);

    // We get Integers, not objects:
    Integer first = naturalNumbers.getFirst();

    Integer last = naturalNumbers.getLast();

    // We don't have to cast them.
    int sum = first + last;

    // Can't be unsafe!
}
```

Benefit – reusability



To make `List` reusable and type safe...

- Instead of *copying* class `List`
 - `ListOfInteger`, `ListOfPerson`, `ListOfChocolate`, ...
- ... we *reuse* the class `List<T>`
 - `List<Integer>`, `List<Person>`, `List<Chocolate>`, ...
 - where `T` is the *type parameter*
 `<...>` are *type arguments*



Subtypes

Subtypes



- > Child is a subtype of Parent if...
 - ... you can **safely** pass a Child into a method where a Parent is expected.

```
void method(Parent p) { ... }  
method(aChild);
```

It's about *substitutability*.



Create and use subtype



```
class Chocolate extends Candy { /* ... */ }
```

Sub class creates a sub type

```
void inspect(Candy c) {  
    c.smell();  
}
```

What type can we *pass* **safely**?

What can we *do with* c in this **context**?

```
forrest.inspect(candy);  
forrest.inspect(chocolate);
```

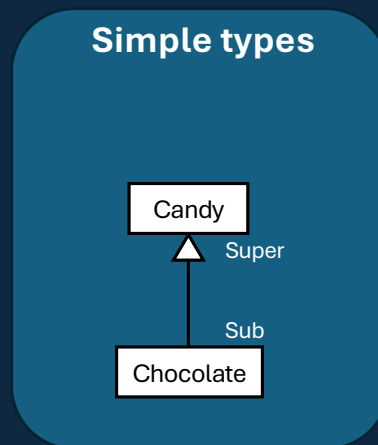
Allowed?



Conclusion



Chocolate is a subtype of Candy



Here, subclass and subtype are the same.

What about generics?



```
// 1. Declaration site
public class Box<T> { } // with type parameter T
//      class ^^^^^^
```

```
// 2. Use site
void inspect(Box<Candy> box) { } // with type argument Candy
//      type  ^^^^^^^^^
```

```
forrest.inspect(candies);    : Box<Candy>
forrest.inspect(chocolates); : Box<Chocolate>
```

What can we *pass safely*?

How can we *use* box in this **context**?

Allowed?

Let's see...

Generic types

Box<Candy>

?

Box<Chocolate>

Introducing...



Forrest



Jenny



=



=



=



Bubba



=



I'm not a
smart man,
but I know
what love is...



The story



- Forrest ❤️ Jenny.
- He buys her a 📦 of stuff she likes.
- Before handing it over, he inspects the box to:
 - 🔍 Check if each candy is fresh.
 - 📦 Safely fill the box.

First try



```
List<byte> bites = new ArrayList<>();
```

```
bites.add(bite1);  
bites.add(bite2);  
bites.add(bite3);  
bites.add(bite4);
```

Type argument cannot be of a primitive type

Replace 'byte' with 'java.lang.Byte'

More actions...

What 🤔?

Why?



- Generics introduced in Java 5 (2004)
 - Java 4 compiled code should still run on JVM 5
 - Java 4 compiled code should still run on JVM 6
 - Java 4 compiled code should still run on JVM 7
 - ... 🥁 ...
 - Java 4 compiled code should still run on JVM 27!
- The JVM is backwards compatible.

Demo

```
void main(String[] args) {  
    System.out.println("I'm running on Java " +  
        System.getProperties().getProperty("java.specification.version"));  
}
```

```
$ /opt/j2sdk1.4.2_19/bin/javac example/Example.java
```

```
$ java -cp . example.Example  
I'm running on Java 25
```

... let me tell you a secret



The JVM doesn't know
generics...



How?



Compile time

```
List<Integer> box = ...  
  
boolean add(T e) { ... }  
  
T getFirst() {  
    return (T) get(0);  
}
```

Run time

```
List box = ...  
  
boolean add(Object e) { ... }  
  
Object getFirst() {  
    return (Object) get(0);  
}
```

byte != Object



That's called Type Erasure.

Implications type erasure



- T must be *at least* an Object
 - i.e. a reference type
 - primitives are not allowed
- T cannot be used at runtime
- ? are "captured" during compilation

Compile time only



- In Java, T is gone at runtime: non-reifiable

```
public <T> void materialize() {  
    var clz = T.class; // ✗  
}
```

Cannot access class object of a type parameter

- In Kotlin, you *can* have a reified T

```
inline fun <reified T> materialize() {  
    val clz = T::class  
}
```

fun must be inline

- In .NET, T is not erased at all.

Capturing



When compiling code like `void opens (List<?> box) ...`

? is the "wild card type": any generic type argument

? must be temporarily "captured" during compilation

- wild card capturing = inferring the type of a wildcard, i.e.:
"what type is ? in this case"
- capture of ? = the inferred type (compiler internal)

And when something is wrong with ? you get errors like:

incompatible types: Integer cannot be converted to capture#1 of ?

Type erasure wrap up



- Java generics are a **compile time construct**.
 - T must be a reference type
 - ? is captured during compilation
- Generic types are **not available at run time**.
 - T cannot be used at runtime



Variance

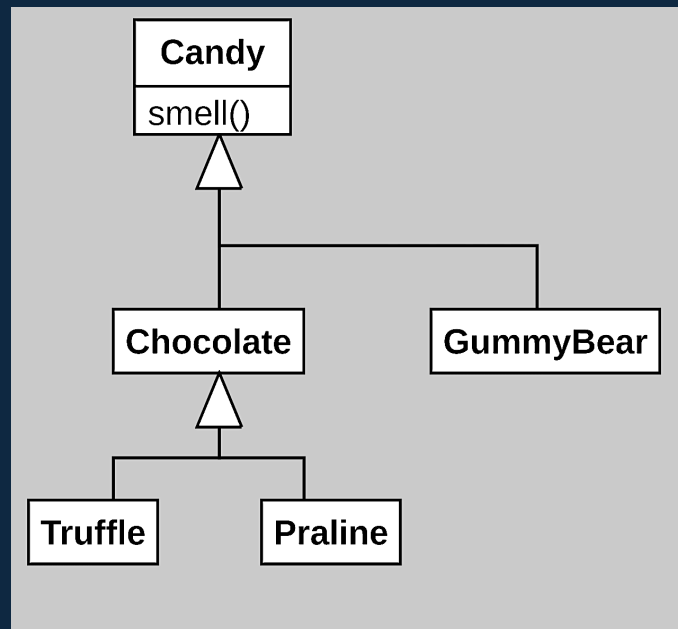
Back to subtyping with generics.



I don't wanna be called stupid.

– Forrest Gump

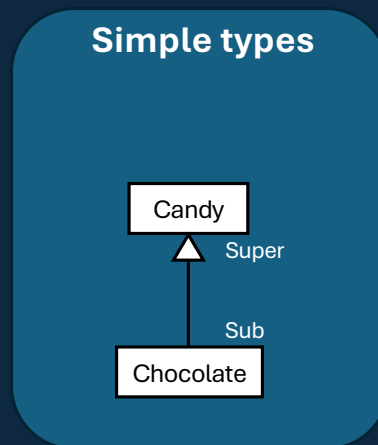
Let's help him out...



Recall



Chocolate is a subtype of Candy



Here, subclass and subtype are the same.

What about generics?



Generic types

Box<Candy>



Box<Chocolate>

Subtypes and generics?



```
// 1. Declaration site  
public class Box<T> { } // with type parameter T  
//      class ^^^^^^
```

```
// 2. Use site  
void inspect(Box<Candy> box) { } // with type argument Candy  
//      type  ^^^^^^^^^^^
```

How can we *use* box in this context?

```
forrest.inspect(candies);  
forrest.inspect(chocolates);
```

What can we *pass safely*?

```
Cannot resolve method 'inspect(Box<Chocolate>)'
```

Why not?



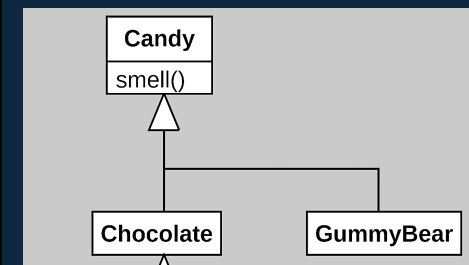
Type **unsafe** from *this* context!

chocolates + 🐻 = 💔

```
public void inspect(Box<Candy> box) {  
    box.forEach(Candy::smell);    // read ✅  
    if (box.isNotSquare()) {  
        box.add(new GummyBear()); // write ✅  
    }  
}
```



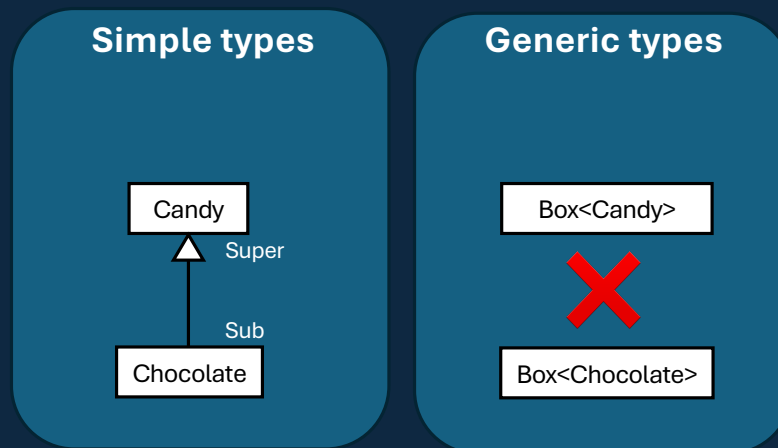
Type **safe** in *this* context.



Conclusion



Box<Chocolate> is NOT a subtype of Box<Candy>



Invariant = “no direction”

Let's try overloading...



```
void inspect(Box<Candy> box) { }    // with type argument Candy  
void inspect(Box<Chocolate> box) { } // with type argument Chocolate
```

'inspect(Box<Candy>)' clashes with 'inspect(Box<Chocolate>)'; both methods have same erasure

- Aha! 💡
 - `Box<Candy> == Box<Chocolate> == Box`

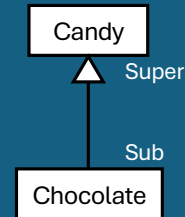
Next attempt...



```
// 2. Use site
public void inspectSafely(Box<? extends Candy> box) {
//           type ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
}
```

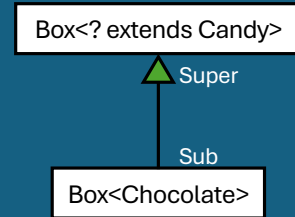
```
forrest.inspectSafely(candies);    // ok
forrest.inspectSafely(chocolates); // ok
```

Simple types



Generic types

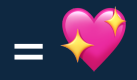
Covariant



Covariant = “**same** direction”

Implications covariant



candies
chocolates +  = 
= 

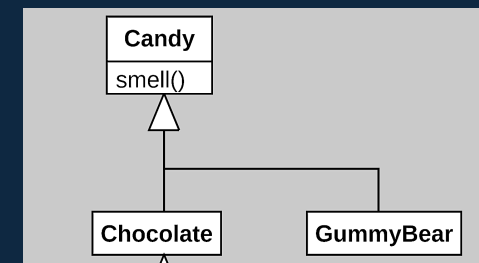


```
public void inspectSafely(Box<? extends Candy> box) {
```

box is a producer

Required type: `Capture of : extends Candy`

Provided: `GummyBear`



Not done yet...



Backlog

 Check if each candy is fresh.

 Safely fill the box.

Done?



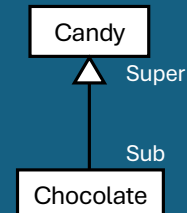
Final attempt



```
// 2. Use site
public void inspectSafely(Box<? super Chocolate> box) {
//           type ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
}
```

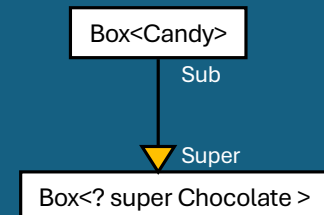
```
forrest.inspectSafely(candies);    // ok
forrest.inspectSafely(chocolates); // ok
```

Simple types



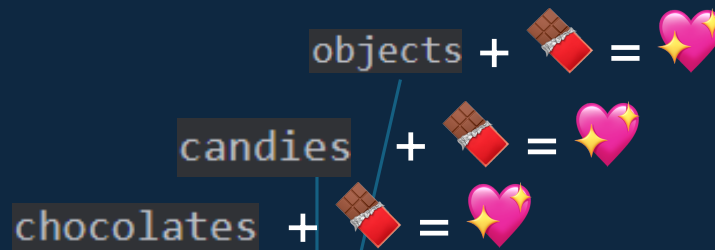
Generic types

Contravariant



Contravariant = “**opposite** direction”

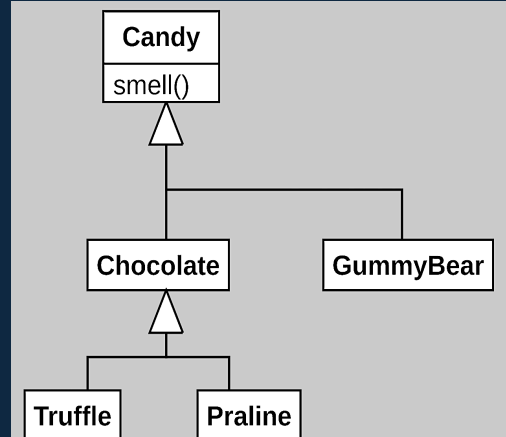
Implications



```
public void inspectSafely(Box<? super Chocolate> box) {
```

```
}
```

box is a *consumer*



Conclusion



Backlog



Check if each candy is fresh.



Safely fill the box.

Done?



with covariant, consumer



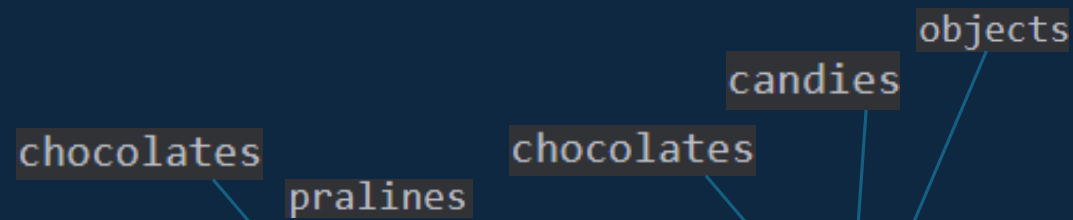
with contravariant, producer

The saga continues...



- Forrest ❤️ nature too.
- He wants to recycle boxes.
 - Move any chocolate from one box into another box.

PECS



```
public void recycle(Box<? extends Chocolate> prod, Box<? super Chocolate> cons) {  
    prod.stream()           // read from source ✓  
    .forEach(cons::add);    // write to destination ✓  
}
```

PECS =
Producer extends
Consumer super

Some observations...



Where to declare variance?



- Java
 - only at the use site of a type:
in method declarations
- Kotlin
 - also at the declaration site of a type:
in method declarations and class declarations

Example Kotlin variance declaration



Declaration site variance

```
class AnyBox<T> {    // Invariant by default
}

class OutBox<out T> { // Covariant always
}

class InBox<in T> {  // Contravariant always
}
```

Use site variance

```
fun <T> add(cons: AnyBox<in T>) {
}

fun <T> get(prod: AnyBox<out T>) {
}
```

Focus?



- Java: what you can *pass*

`Box<? extends T> = co`

`Box<? super T> = contra`

- Kotlin: what you can *do*

`Box<out T> = producer`

`Box<in T> = consumer`

Breaking type safety 🧨

The final chapter

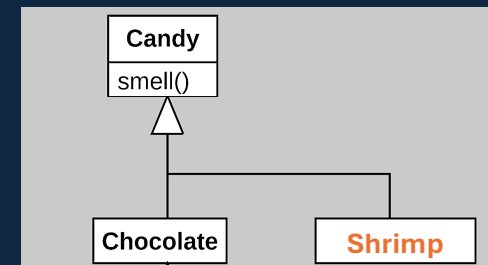


```
public void inspect(Candy[] candies) {  
    stream(candies).forEach(Candy::smell);  
    candies[candies.length - 1] = new Shrimp();  
}
```

```
bubba.inspect(arrayOfChocolates);
```

Does this compile..? 

Runtime: `chocolates` +  = ...



The background image depicts a powerful volcanic eruption. A massive, bright orange and yellow fireball of molten lava is being ejected from a volcano, filling much of the upper half of the frame. In the foreground, two people are standing in a lush, green tropical environment with banana trees and other foliage. They appear to be looking up at the eruption. The overall scene is dramatic and intense.

```
Exception in thread "main" java.lang.ArrayStoreException: nl.bramjanssens.domain.Shrimp
```

Arrays are covariant but NOT type safe, Bubba!

Wrap up

Let's drive this story home



Credits: Boxoffice Movie Scenes

The Importance of Variance



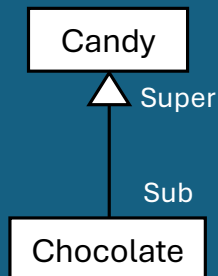
1. Increases code Flexibility and Reusability
 - Allows using super or subtypes when needed
2. Enables safer and cleaner generic code
 - Safely read/write a container
3. Improves Functional Programming Patterns
 - Monads like Stream, Optional, Try, Maybe, ...
4. Understand 🤔 and impress 💪 your colleagues!

Java Variance Cheat Sheet



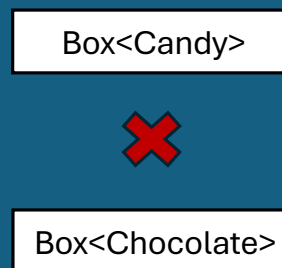
Bridging the gap between inheritance hierarchies and generic type systems.

Simple types

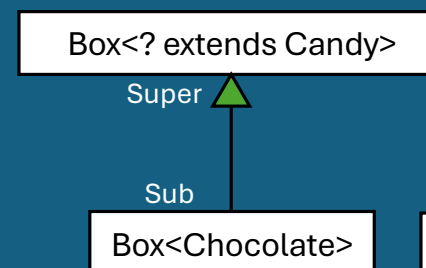


Generic types

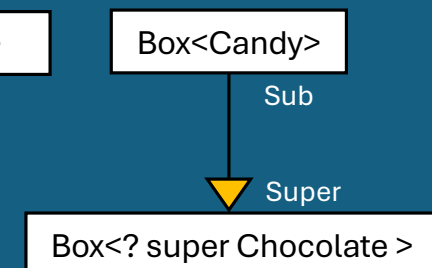
Invariant



Covariant producer



Contravariant consumer



PECS

Producer **extends**
Consumer **super**

Arrays are covariant and not type safe!

Generics are erased during compilation.

Variance is declared at the use site only.



Generics

Now you know what you're gonna get!





slides + cheat sheet



bramjanssens.nl



@sajanssens



/sajanssens



infoSupport
Solid Innovator