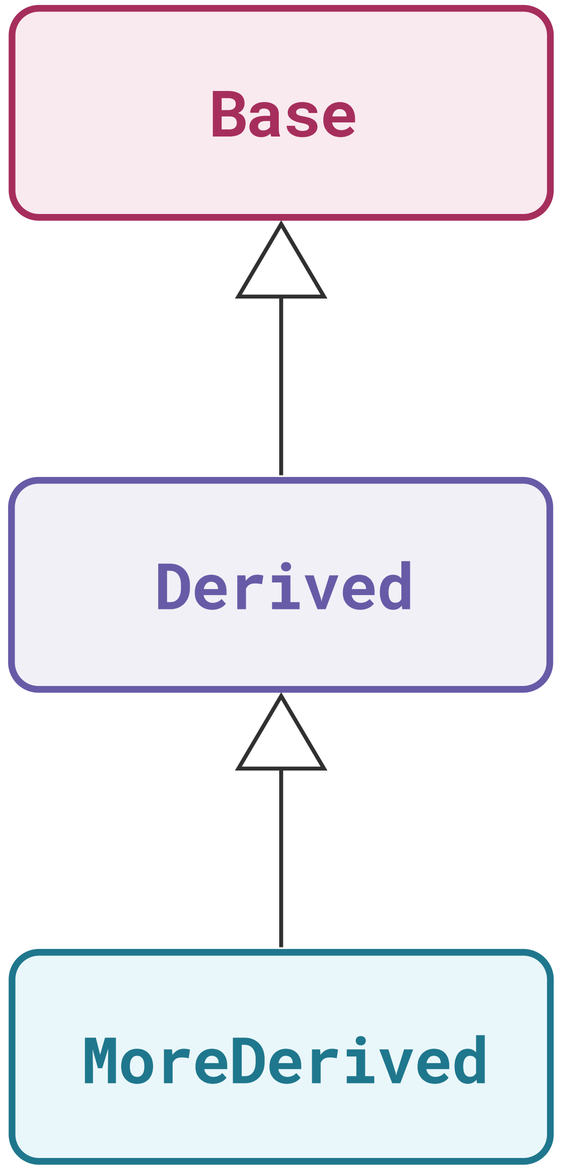
# Using Generic Types in the Object Model

**Zoran Horvat**
CEO at Coding Helmet

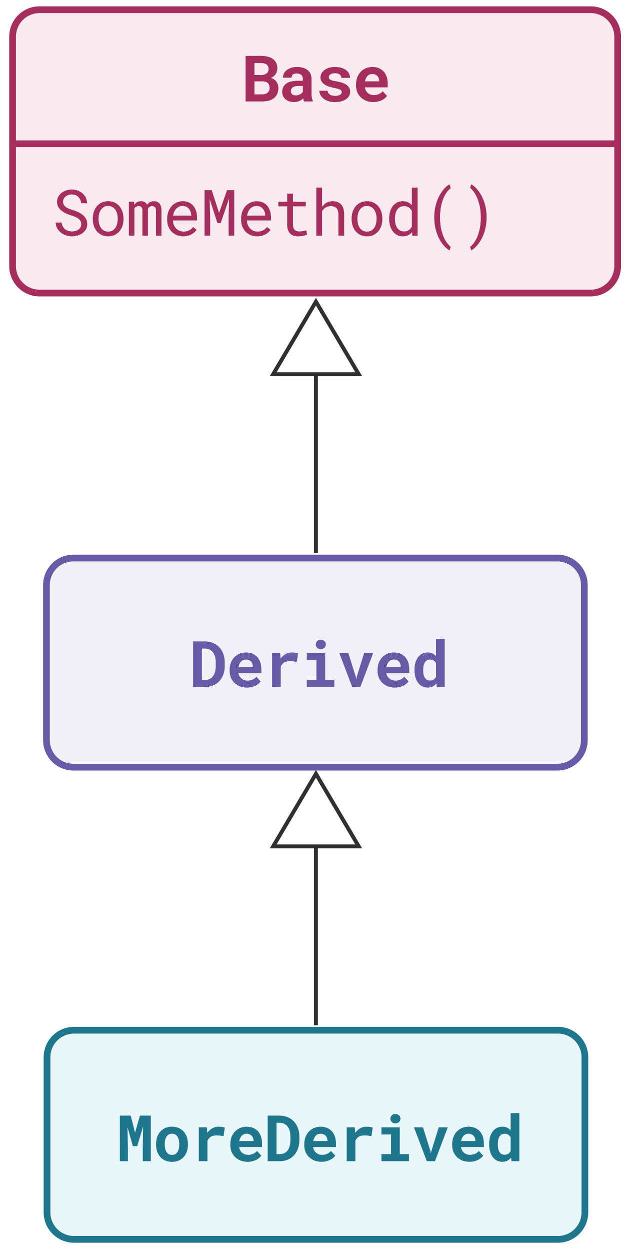
@zoranh75 https://codinghelmet.com

# Augmenting Object Substitution



```
Base a = new Base();
Base b = new Derived();
Base c = new MoreDerived();
```

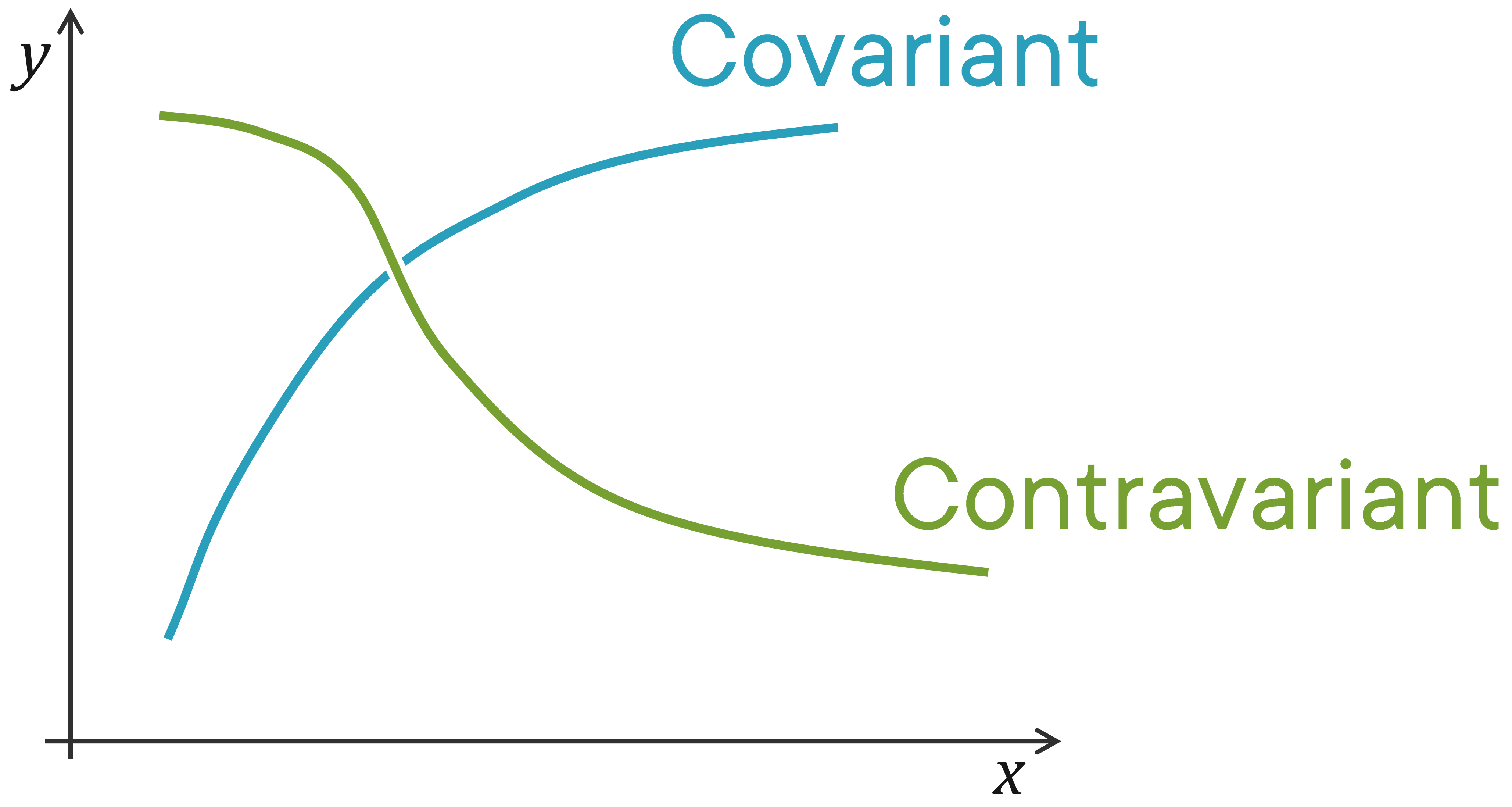
# Augmenting Object Substitution



```
Base a = new Base();
Base b = new Derived();
Base c = new MoreDerived();

c.SomeMethod();
```

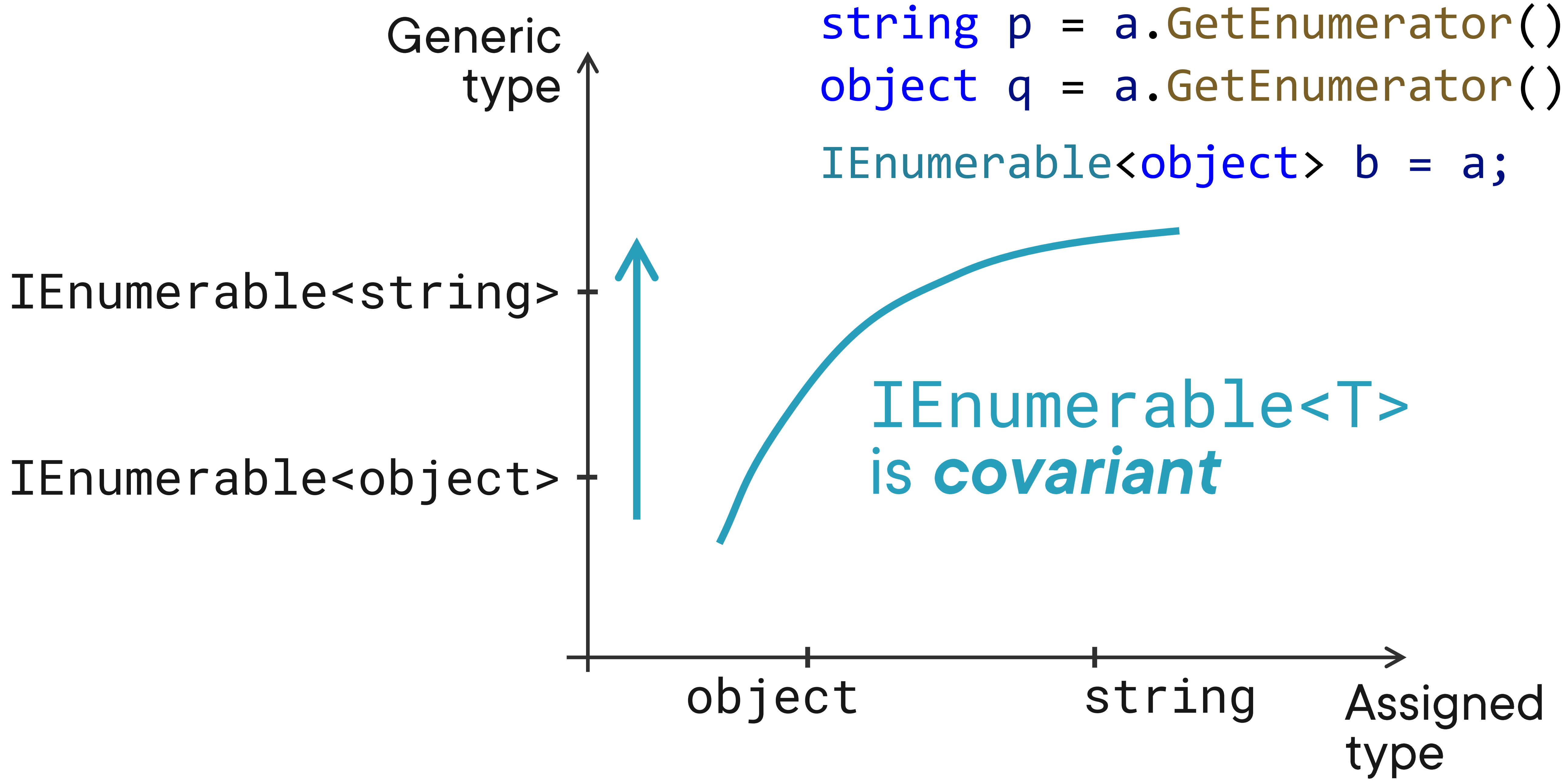
Compiler verifies that the method exists

# Introducing Generic Variance



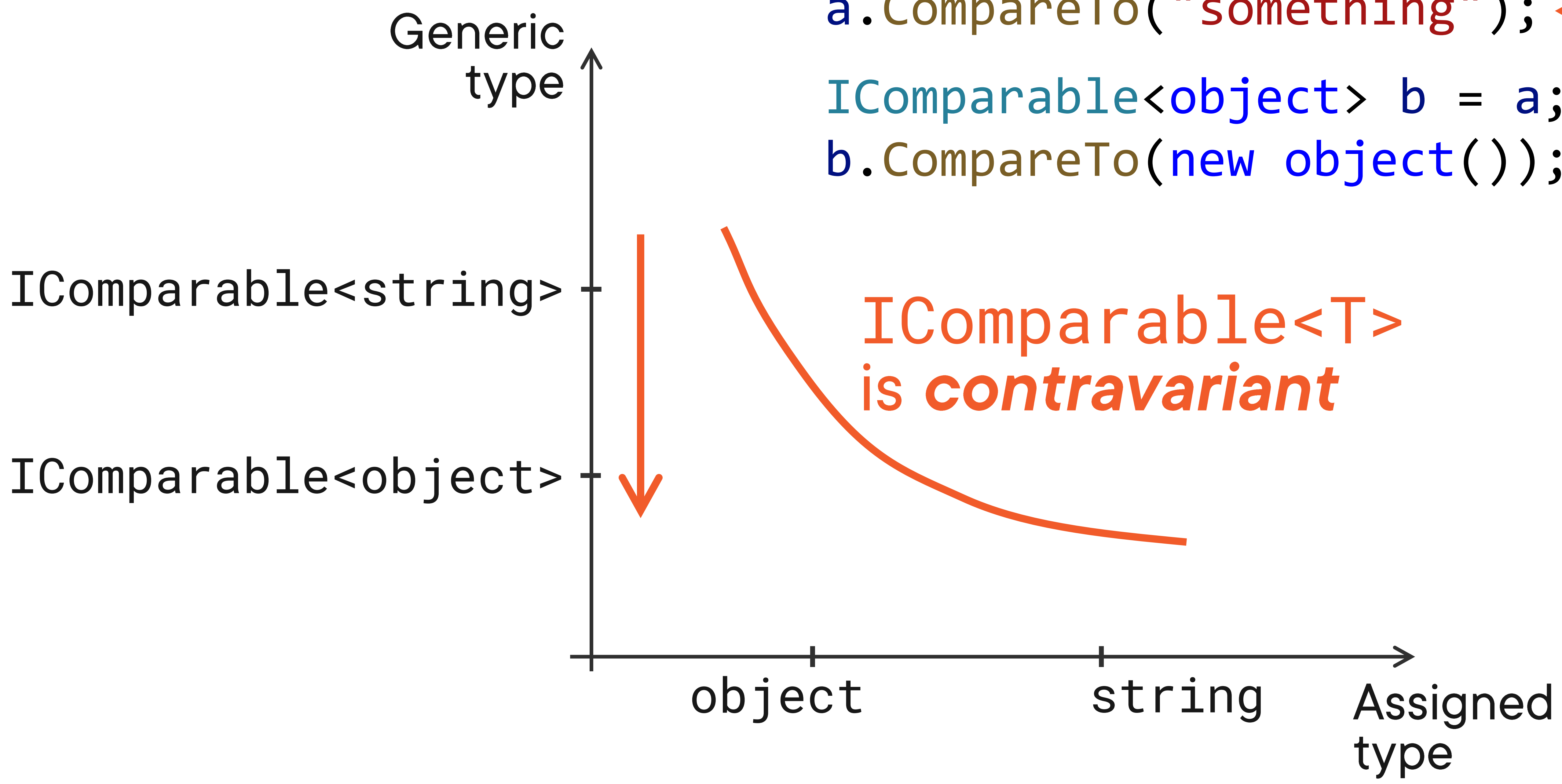
# Introducing Generic Variance

```
IEnumerable<string> a = ...
string p = a.GetEnumerator().Current;
object q = a.GetEnumerator().Current;
IEnumerable<object> b = a;
```



# Introducing Generic Variance

```
IComparable<string> a = ...
a.CompareTo("something");

IComparable<object> b = a;
b.CompareTo(new object());
```



IComparable<T> is ***contravariant***

# Introducing Generic Variance

Variance only applies to interfaces and delegate types

`IEnumerable<out T>`

Covariant

`IComparable<in T>`

Contravariant

Invariant

`IList<T>`

## The Array Covariance Problem

```
object[] a = new string[10];
a[0] = new DateTime(2022, 1, 1);
IList<object> b = new List<string>();
```

`ArrayTypeMismatchException`

Attempted to access an element
as a type incompatible with the array.

# The Array Covariance Problem

```
object[] a = new string[10];
a[0] = new DateTime(2022, 1, 1);
IList<object> b = new List<string>();
```

Cannot implicitly convert type 'List<string>' to 'IList<object>'

# The Array Covariance Problem

```
object[] a = new string[10];
a[0] = new DateTime(2022, 1, 1);
IList<object> b = new List<string>(); ✗
IEnumerable<object> c = new List<string>(); ✓
```

# Understanding the Limitation of Contravariance

```
public interface IContravariant<in T>
{
    void DoSomething(T obj);
}

Worker worker = ...;
IContravariant<Employee> contravariant = ...;

contravariant.DoSomething(worker);
```

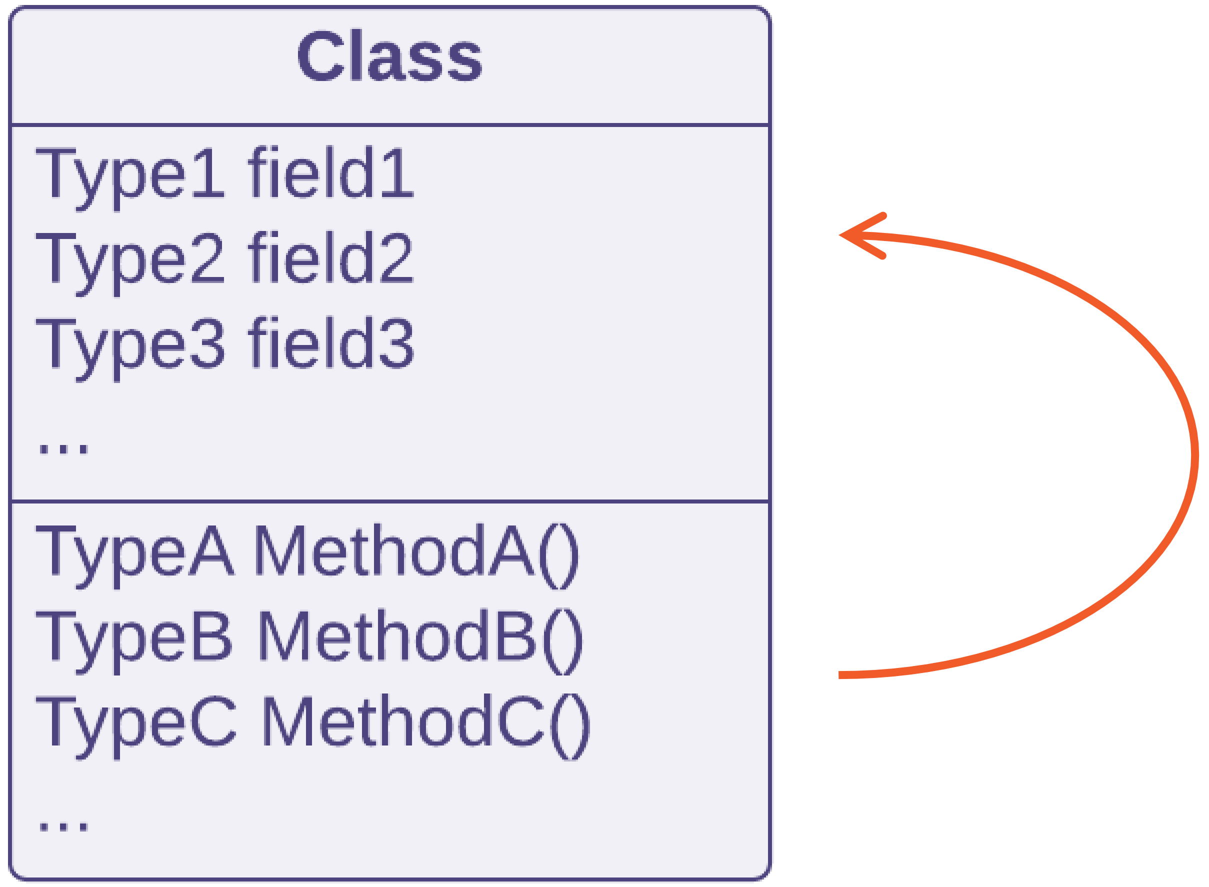
```
public interface IInvariant<T>
{
    void DoSomething(T obj);
}

Worker worker = ...;
IInvariant<Employee> invariant = ...;

invariant.DoSomething(worker);
```

We can pass a derived instance to an invariant method, too!

```
interface IOrderedList<out T>
  : IReadOnlyCollection<T>
{
}

class Implementer<T> : IOrderedList<T>
{
}

class Consumer<T>
{
  public void Do(IOrderedList<T> list) ...
}

Consumer<string> consumer = new();

consumer.Do(new Implementer<string>());

consumer.Do(new List<string>());
```

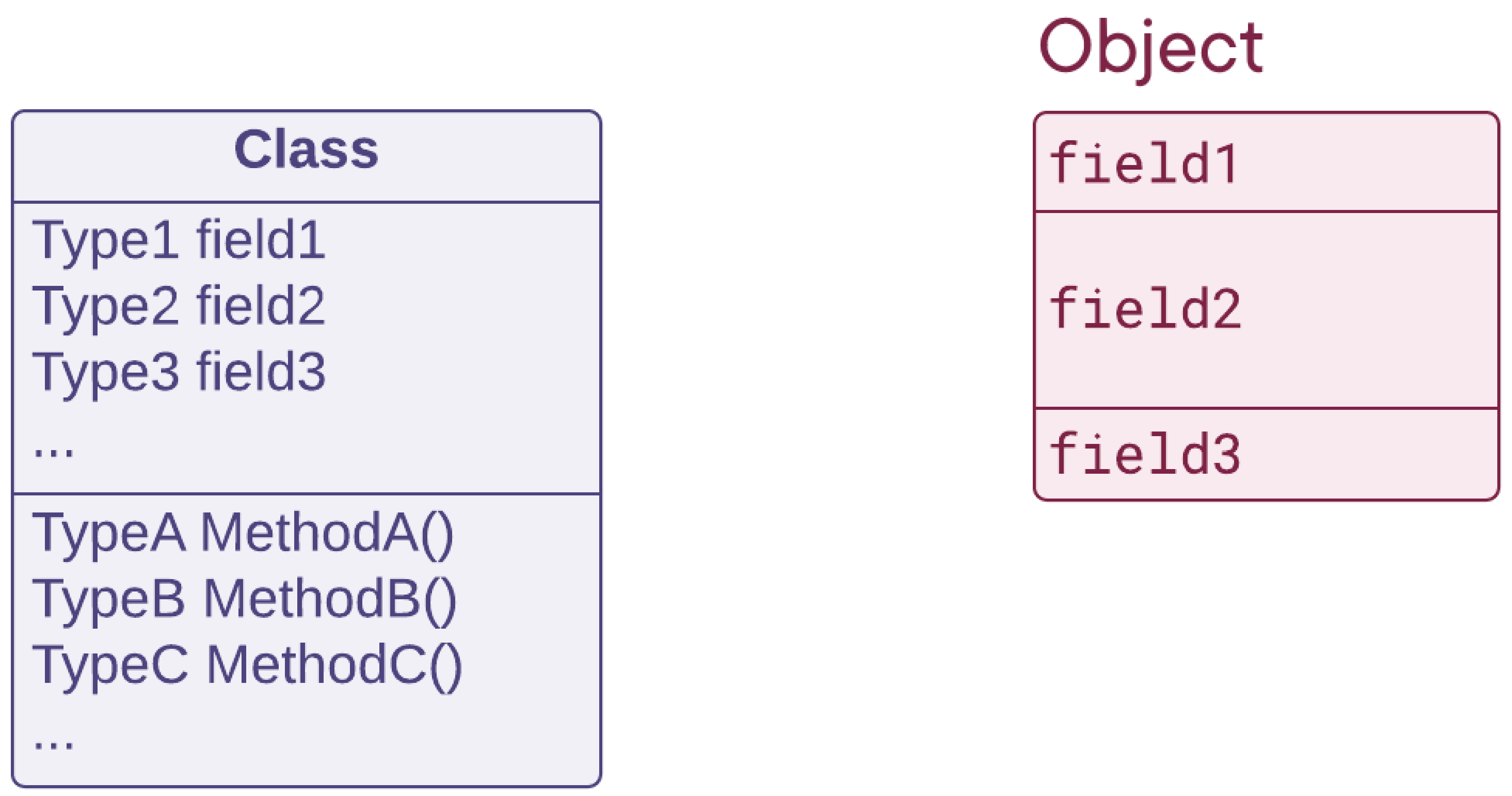
- Marker interface
- A concrete implementation does nothing
- But it is now carrying a semantical meaning
- A method expects the interface
- Works fine because instance is `IOrderedList`
- Fails because `List` is only `IReadOnly`
- Also useful in constrained generics (comes in the next module)

# Compiling Generic Types



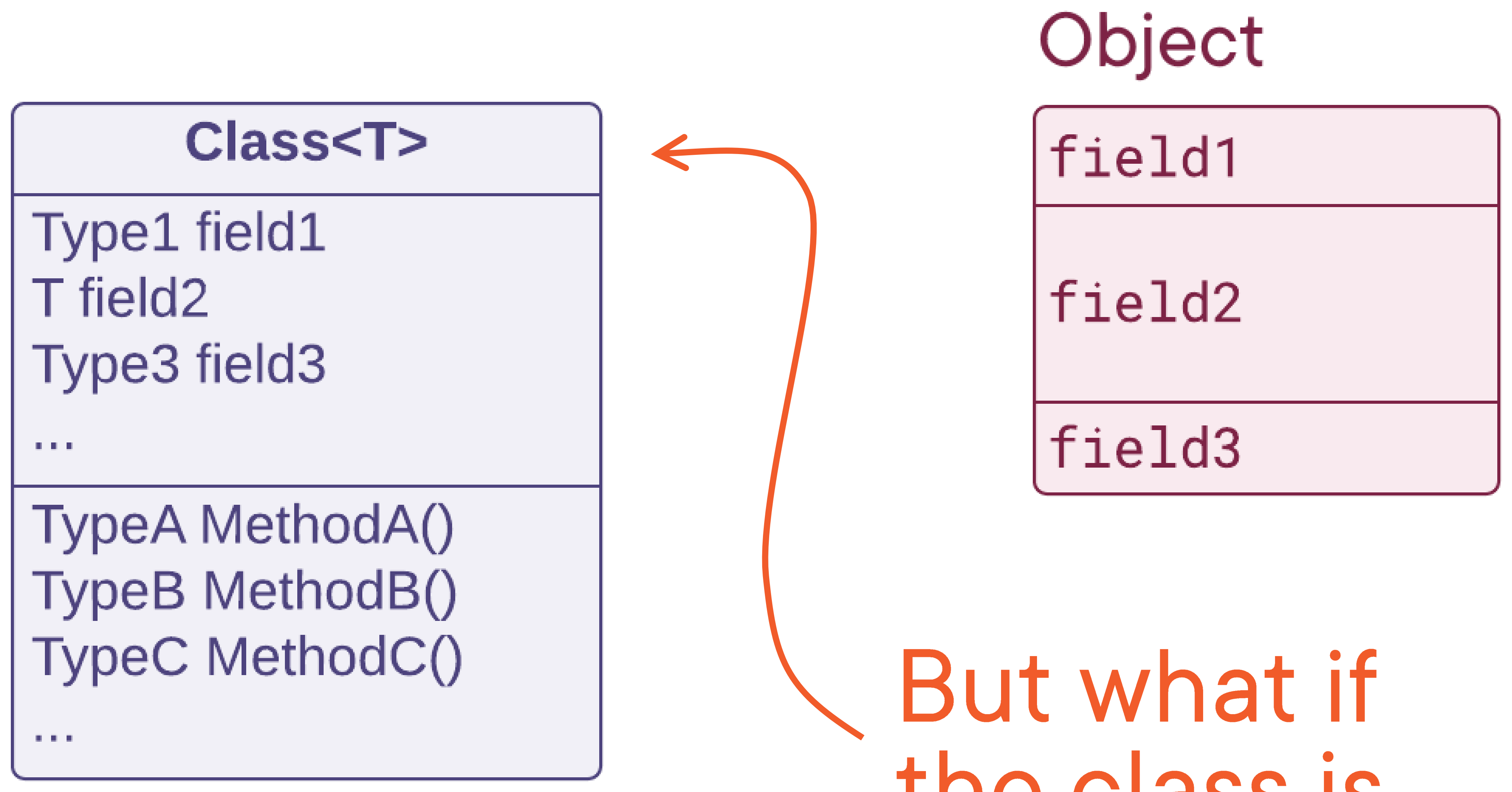
Methods need relative positions of fields

# Compiling Generic Types



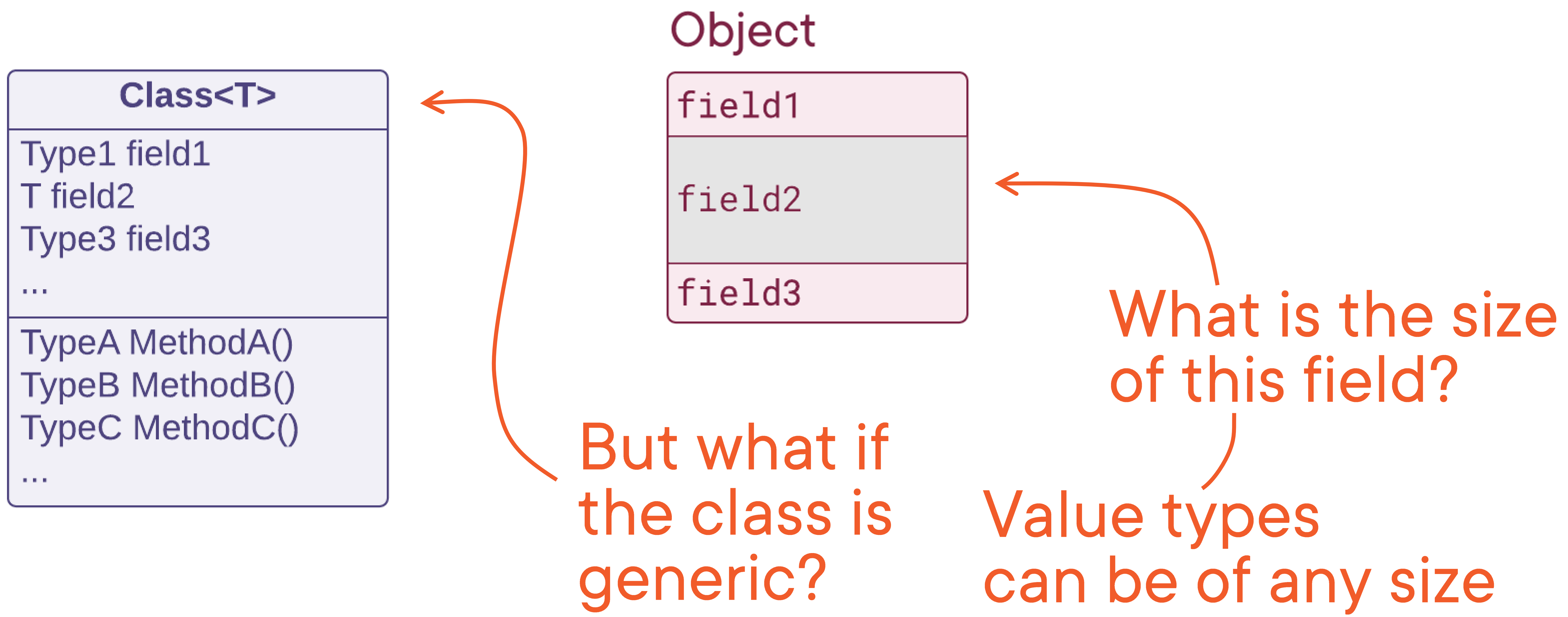
A field's type determines its size

## Compiling Generic Types



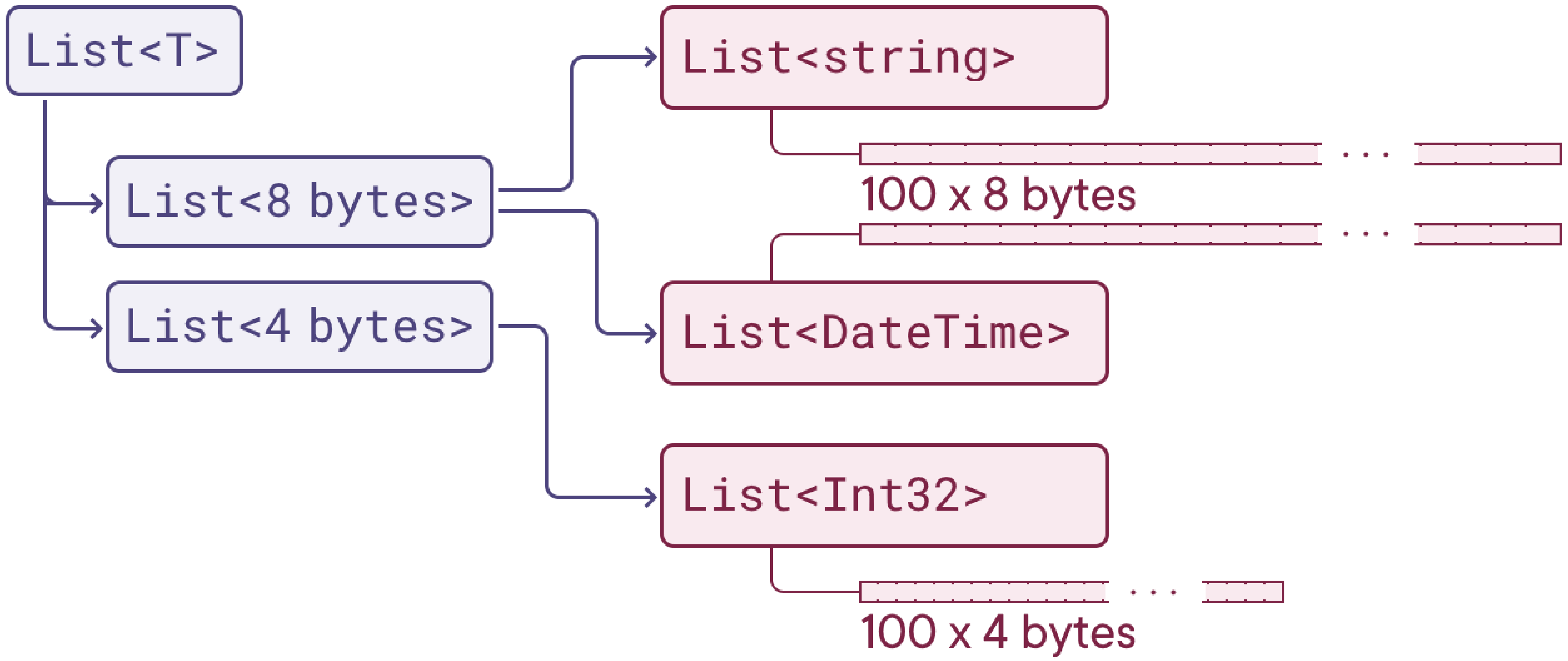
But what if the class is generic?

# Compiling Generic Types



# Compiling Generic Types

```
List<T>

List<string>
List<DateTime>

List<Int32>
```



## Summary

**Principles of generic variance**

- Variance specifies subtype relationship between generic types
- Either covariant, contravariant or invariant
- Variance helps enforce the Object Substitution Principle
- Keywords `in` and `out` specify contravariant and covariant types

## Up Next: Designing Generic Types