Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 11 + Tutorial 11
Java Generics

Eugene Zouev
Fall Semester 2020
Innopolis University

What We Have Learnt Before

- Classes, class instances
- Value and reference types
- **Encapsulation, Inheritance, Polymorphism**
- Class `Object`
- Casts
- Abstract classes
- Interfaces
- Exceptions

The Plan for Today

- The idea of genericity
- The life without genericity
- Boxing & unboxing
- Generics in Java: type parametrization
- Requirements on actual types
- The notion of **variance**
- Variance & wildcards

Templates/Generics

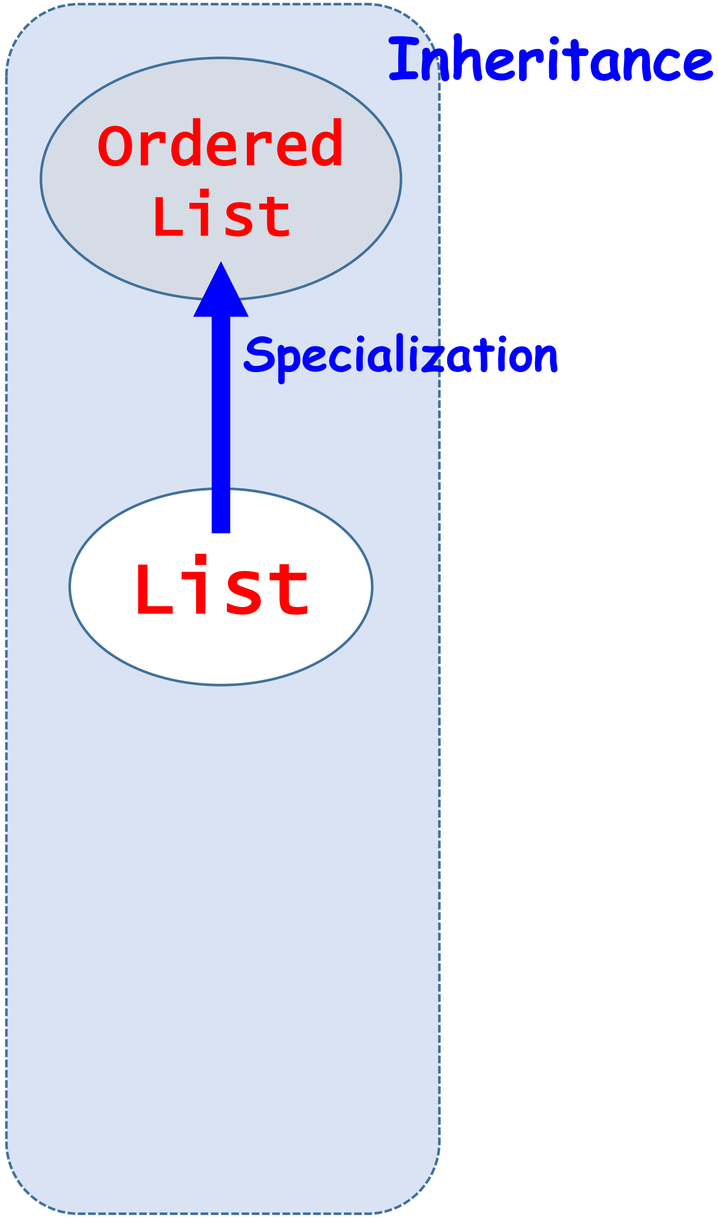
Template/generic mechanism is not an exotic language feature; almost each well-known programming language has it.- NOT only OOP languages.

- **Generics** in Ada, Delphi, Eiffel, Java, Scala, C#, Swift, Rust.
- **Templates** in C++ and D
- **Parametric polymorphism** in ML, Scala, Haskell

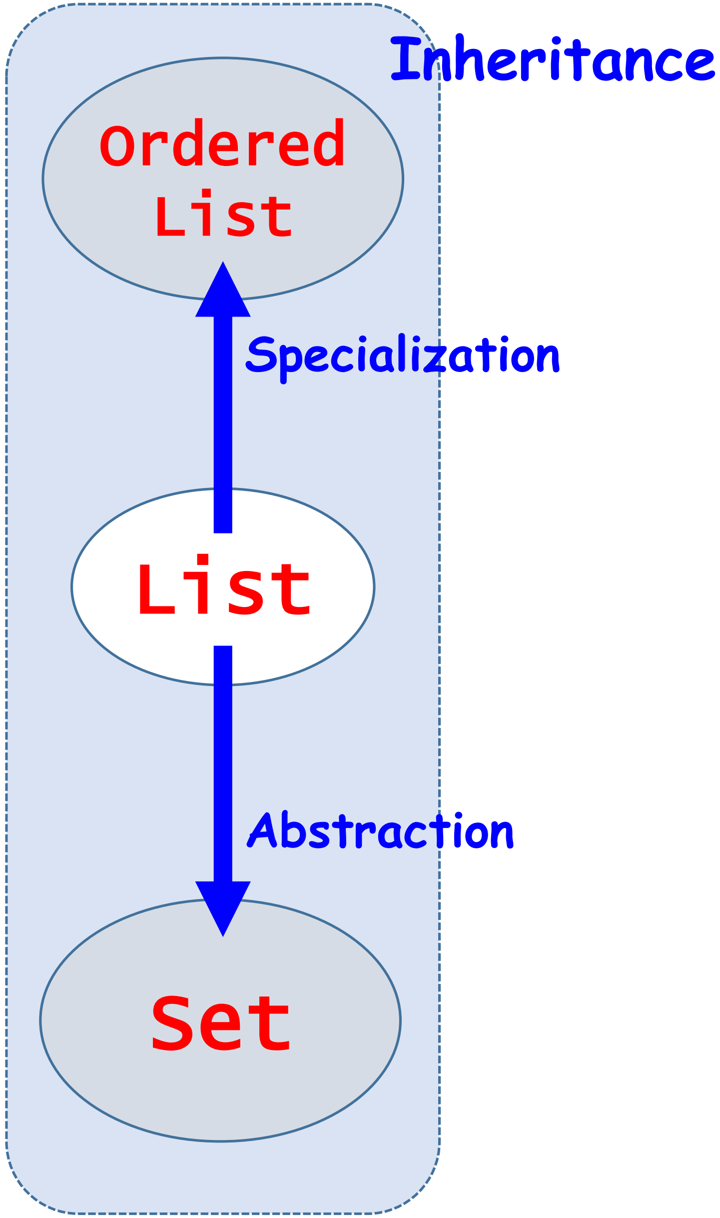
Genericity: the Idea

List

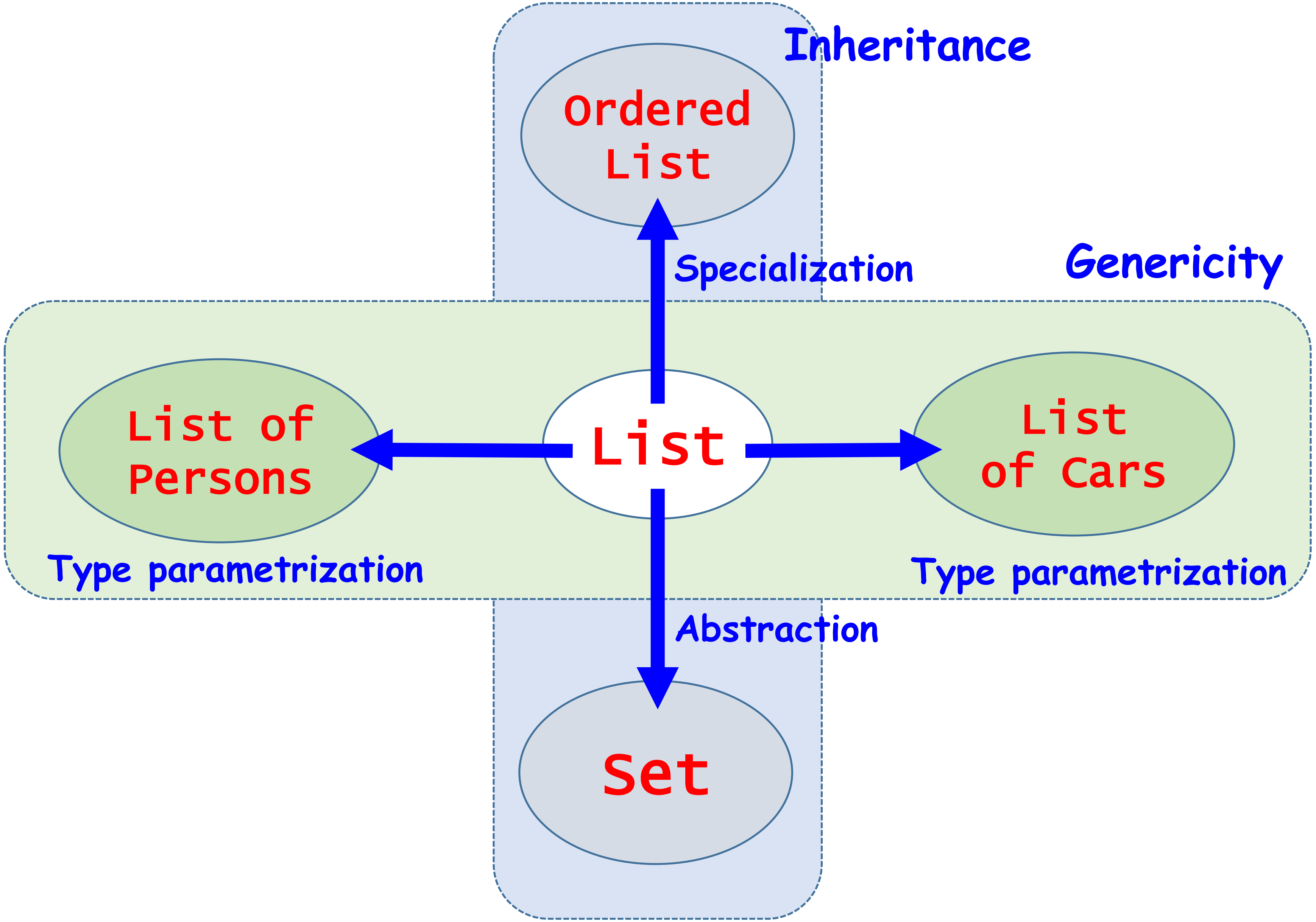
Genericity: the Idea



Genericity: the Idea

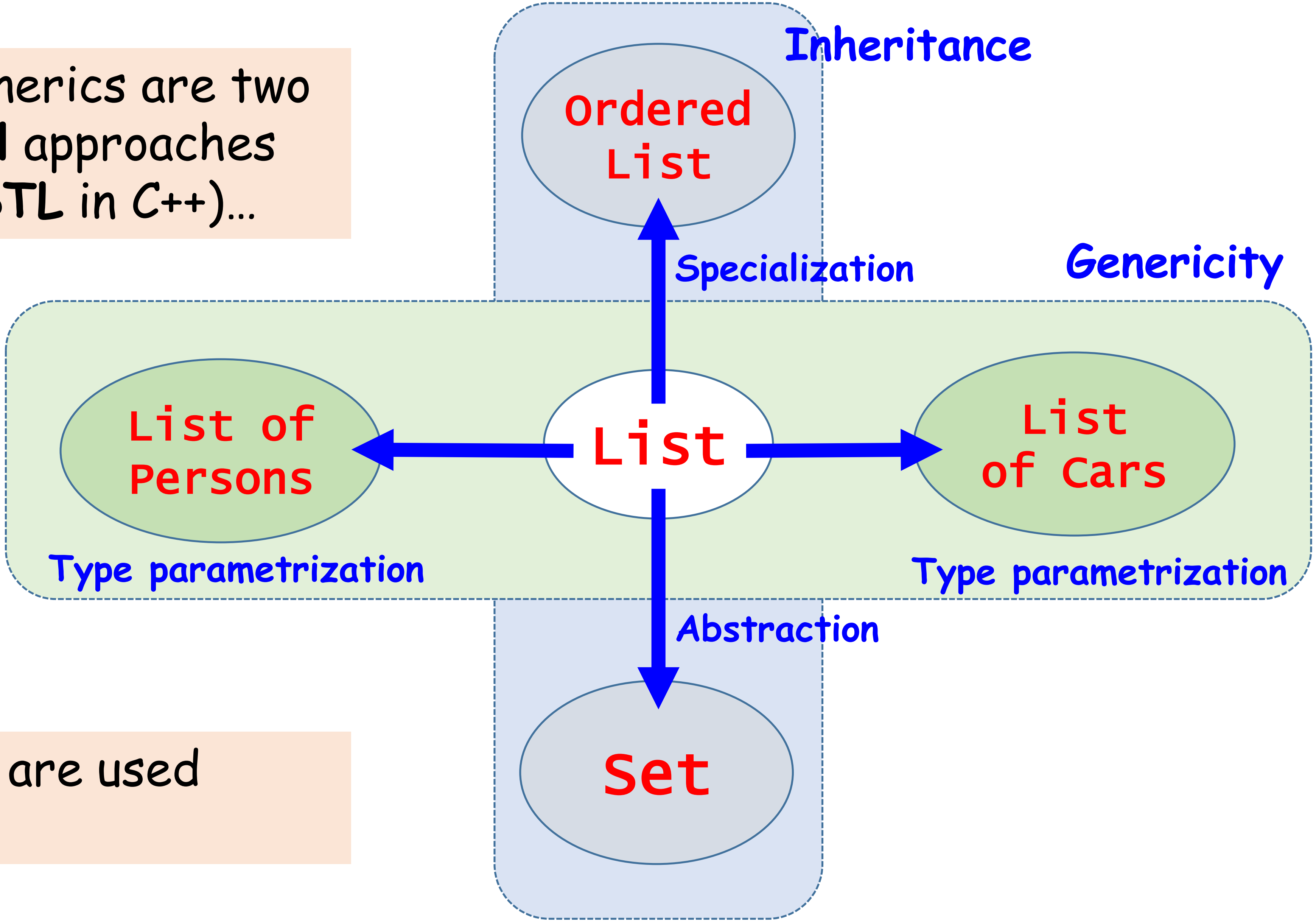


Genericity: the Idea



Genericity: the Idea

OOP & Generics are two **orthogonal** approaches (see, eg, **STL** in C++)...



...but they are used together.

The Life Without Genericity

```
class ListOfPersons {
    void extend(Person v) { ... }
    void remove(Person v) { ... }
}
```

The Life Without Genericity

```
class ListOfPersons {
    void extend(Person v) { ... }
    void remove(Person v) { ... }
}
```

```
class ListOfCars {
    void extend(Car v) { ... }
    void remove(Car v) { ... }
}
```

```
...
```

The Life Without Genericity

```
class ListOfPersons {
    void extend(Person v) { ... }
    void remove(Person v) { ... }
}
```

extend and remove algorithms **are the same**!

```
class ListOfCars {
    void extend(Car v) { ... }
    void remove(Car v) { ... }
}
```

```
...
```

DRY principle:
Don't **R**epeat **Y**ourself

Possible Approaches

- **Duplicate code**, manually or with help of a macro processor.
- Convert ("cast") all values to **a universal type**, such as "pointer to void" in C/C++, or to `Object` class in Java/C#.
- **Parametrize** the class **giving an explicit name** which is to denote **any type** of container element.

C++ Approach: the "Universal" Type

```
class ListOfAnything {
    void extend(void* v) { ... }
    void remove(void* v) { ... }
};
```

C++

Any pointer can be converted to `void*`

```
ListOfAnything lst;
...
lst.extend(new Car());
```

Yeah, this seems to be OK...

C++ Approach: the "Universal" Type

```
class ListOfAnything {
    void extend(void* v) { ... }
    void remove(void* v) { ... }
};
```

C++

Any pointer can be converted to void*

```
ListOfAnything lst;
...
lst.extend(new Car());
lst.extend(new Person());
```

Yeah, this seems to be OK…

Oops… Compiler doesn't complain. But what does it mean semantically?

C++ Approach: the "Universal" Type

```
class ListOfAnything {
     void extend(void* v) { ... }
     void remove(void* v) { ... }
};
```

C++

Any pointer can be converted to `void*`

```
ListOfAnything lst;
...
lst.extend(new Car());
lst.extend(new Person());
...
City* London = new City();
lst.remove(London);
```

Yeah, this seems to be OK…

Oops… Compiler doesn't complain. But what does it mean <u>semantically</u>?

This is even more strange, but compiler doesn't complain either.

Java Approach: Common Base Type

Generic list implementation <u>without</u> generics

```
public class List
{
    // Internal List implementation
    ...
    // Public List interface
    public void extend ( Object item ) { ... }
    public Object elem ( int i ) { ... }
}
```

`Object` in Java (and *C#*, BTW) is a common base to all class types: both library- and user-defined. This means that it is always possible to convert:

Any Type → `object`

```
List lst = new List();
lst.extend(new MyType());
MyType v = (MyType)lst.elem(5);
```

Java: Boxing & Unboxing

Java types

Value types | Reference types

The problem:

`int` type **is not a reference type**; therefore, it's not derived from `Object`. How could we put the value of `int` type to the list of `Object`s??

```
List lst = new List();
lst.extend(new MyType());   // correct
Lst.extend(777);            // what's this??
```

Java Solution: Wrapper Classes

Value types

```
byte
short
int
long
float
double
boolean
char
```

Eight value types…

Java Solution: Wrapper Classes

Value types

```
byte
short
int
long
float
double
boolean
char
```

Eight value types…

…and each of these has a corresponding library class

Reference types

```
class Byte
class Short
class Integer
class Long
class Float
class Double
class Boolean
class Character
```

package `java.lang`

Java Approach: Wrapper Classes

"Class equivalents " for value types:

`int` ⟺ `class Integer`

An instance of the "wrapper" class `Integer`

A value

A value

```
Integer i =
        new Integer(1);
```

Java Approach: Wrapper Classes

"Class equivalents " for value types:

`int` ⟷ `class Integer`

An instance of the "wrapper" class `Integer`

A value

A value

```
Integer i =
        new Integer(1);
```

`double` ⟷ `class Double`

An instance of the "wrapper" class `Double`

A value

A value

```
Double d =
       new Double(0.5);
```

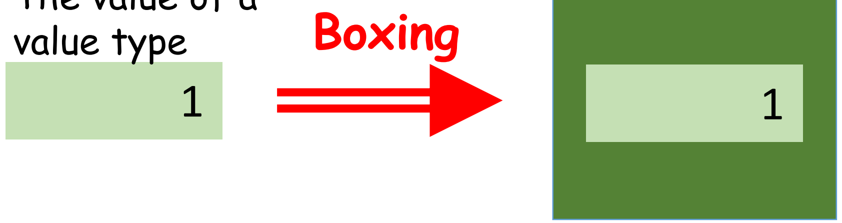
Java: Boxing & Unboxing

```
List lst = new List();
lst.extend(1);      // int->Object: boxing
```

The same as
`new Integer(1)`

The value of a
value type

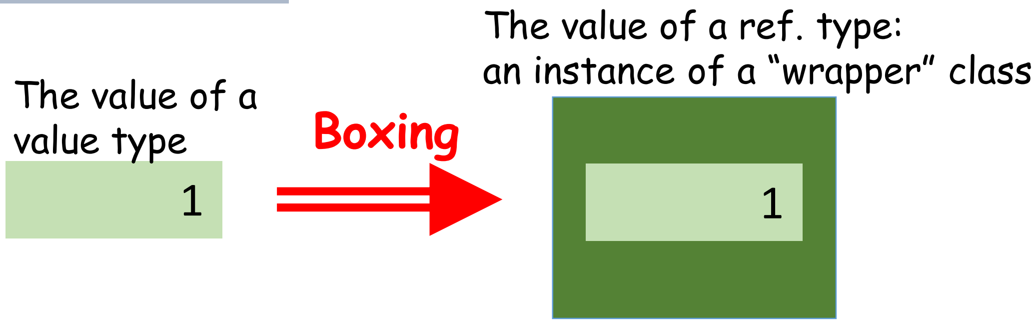
The value of a ref. type:
an instance of a “wrapper” class



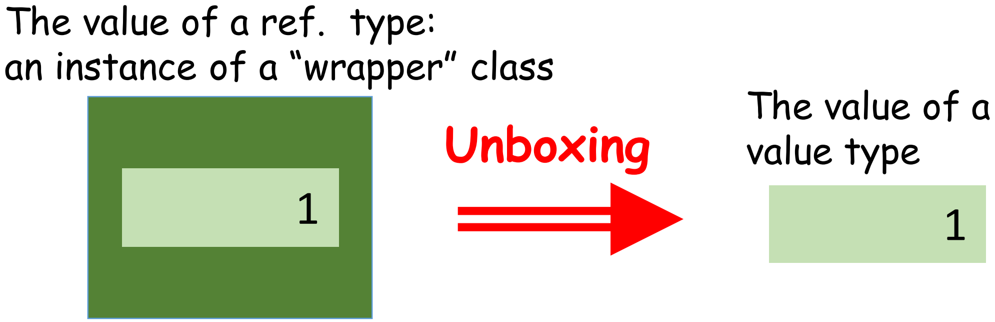
Java: Boxing & Unboxing

```
List lst = new List();
lst.extend(1);       // int->Object: boxing
```

The same as
`new Integer(1)`




```
int i = (int)lst2.elem(1);
                // Object->int: unboxing
```



Java Approach: Common Base Type

```
List lst1 = new List();
lst1.extend(new MyType());
MyType v = (MyType)lst1.elem(0);

List lst2 = new List();
lst2.extend(1);                // int->Object: boxing
int i = (int)lst2.elem(1);// Object->int: unboxing

List lst3 = new List();
lst3.extend(new MyType());
int j = (int)lst3.elem(2);  // Run-time error!
```

Java Approach: Common Base Type

```
List lst1 = new List();
lst1.extend(new MyType());
MyType v = (MyType)lst1.elem(0);

List lst2 = new List();
lst2.extend(1);                // int->Object: boxing
int i = (int)lst2.elem(1);// Object->int: unboxing

List lst3 = new List();
lst3.extend(new MyType());
int j = (int)lst3.elem(2);  // Run-time error!
```

Problems (disadvantages):

- Cannot specify the type of list elements
- Compiler cannot check type consistency
- **Boxing/unboxing** necessary for value types

Approach with Type Parametrization

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

C# too!

Approach with Type Parametrization

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

C# too!

- T denotes something like "**any type**". It's called **universal parameter**.
- The whole `List<T>` declaration specifies a list whose all elements are of *some type* T.
- The `List<T>` declaration is (still) an abstraction ("generic", or "template"): in order to use it we have to **instantiate** it specifying a particular ("actual") type.
- The result of instantiating is a "real" class and it can be used exactly as a usual (non-generic) class.

Approach with Type Parametrization

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem( int i) { ... }
}
```

Here, the compiler **instantiates** `List<T>` generic replacing `T` for `Car` and producing the new type `List<Car>`

```
List<Car> garage = new List<Car>();
...
lst.extend(new Car());    // OK
lst.extend(new Person()); // Error!
```

Approach with Type Parametrization

```
List<MyType> lst1 = new List<MyType>();
lst1.extend(new MyType());
MyType v = lst1.elem();    // no need to convert

List<Integer> lst2 = new List<>();
lst2.extend(1);            // no implicit conversion; no boxing
int i = lst2.elem(1);      // no expl.conversion:no unboxing
Lst2.extend(new MyType()); // compile-time error

List<MyType> lst3 = new List();
lst3.extend(new MyType());
int j = (int)lst3.elem(3); // Compile-time error: illegal conversion
```

Approach with Type Parametrization

```
List<MyType> lst1 = new List<MyType>();
lst1.extend(new MyType());
MyType v = lst1.elem();     // no need to convert

List<Integer> lst2 = new List<>();
lst2.extend(1);             // no implicit conversion; no boxing
int i = lst2.elem(1);       // no expl.conversion:no unboxing
Lst2.extend(new MyType()); // compile-time error

List<MyType> lst3 = new List();
lst3.extend(new MyType());
int j = (int)lst3.elem(3); // Compile-time error: illegal conversion
```

Advantages

- Type of list elements is explicitly and statically specified
- **No boxing and unboxing** conversions
- Compiler is always able to check type consistency

Approach with Type Parametrization

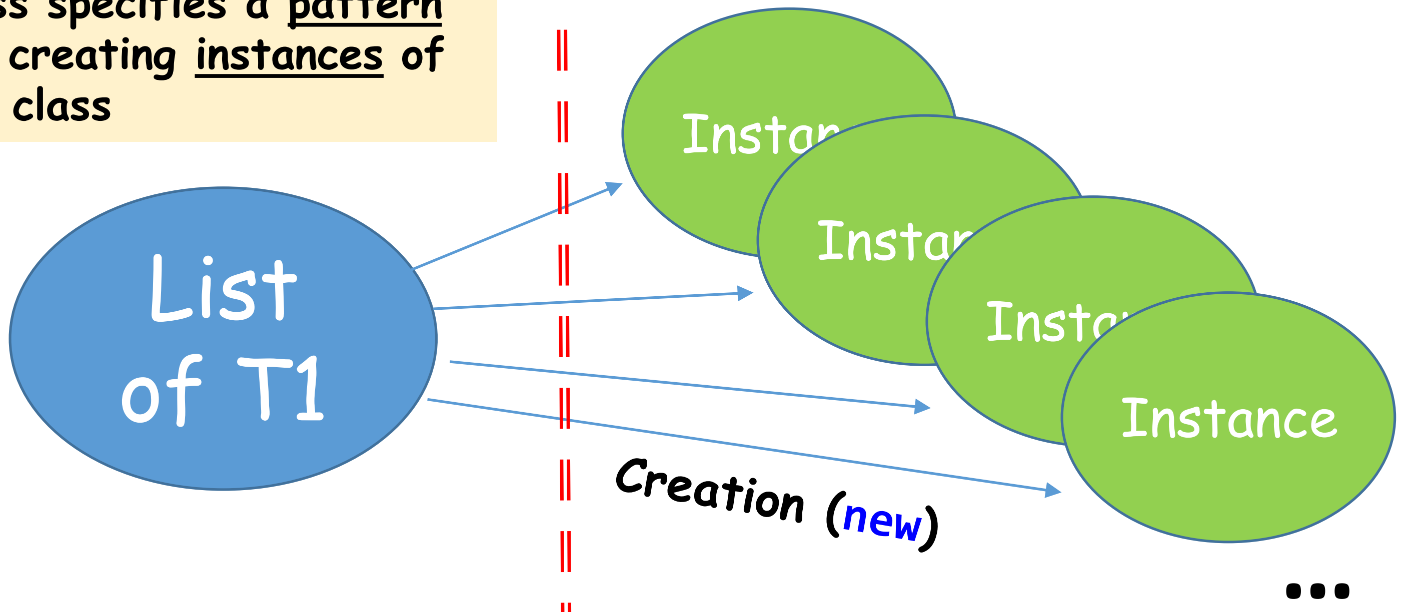
Benefits:

- We have **type safety**: we cannot put an element of type `Person` into the list consisting of `Car`s.
- We don't need to write a **new list implementation** for every type of things we want to put into it.
- This solution doesn't lead to any kind of inefficient performance because instantiating is done entirely **at compile-time**.

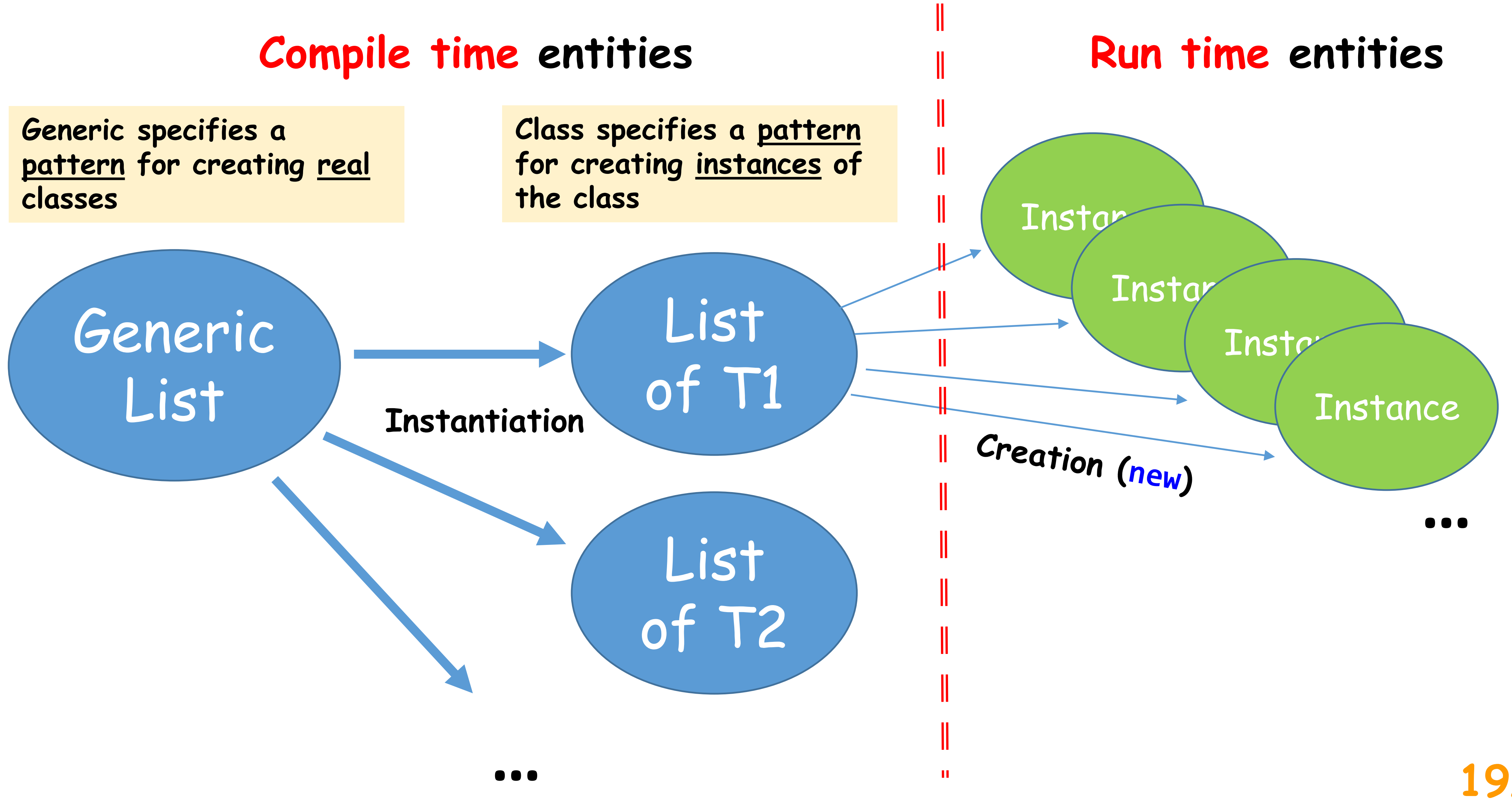
Generics, Classes & Instances

Compile time entities | **Run time entities**

Class specifies a <u>pattern</u> for creating <u>instances</u> of the class



Generics, Classes & Instances



Generics & Classes: C++ vs Java

Implementation approaches

C++: **expansion**
For each instantiation, the new copy of the class is generated, with type parameters replaced for actual ones

"Macroprocessing on steroids" ☺

☹ Code bloat
☺ Better optimization

Generics & Classes: C++ vs Java

Implementation approaches

C++: expansion
For each instantiation, the new copy of the class is generated, with type parameters replaced for actual ones

"Macroprocessing on steroids" ☺

☹ Code bloat
☺ Better optimization

Java: erasure
For each instantiation, the same copy of the class is used.

Information about types is removed ("type erasure" method), and boxing & unboxing is used internally.

☺ More compact code
☹ Slower execution

Type Parametrization for Methods

Generic class

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

Indicates that the method is generic

Generic method

```
class Lists
{
    public static <T> T sort(List<T> lst)
    {
        ...
    }
}
```

Type Parametrization for Methods

Generic class

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

Indicates that the method is generic

Generic method

```
class Lists
{
    public static <T> T sort(List<T> lst)
    {
        ...
    }
}
```

?

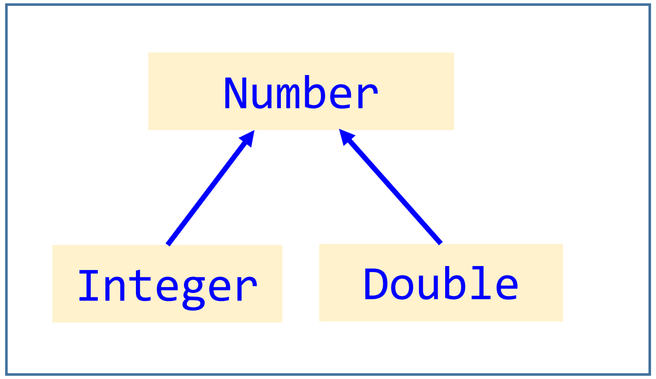
```
public static T sort<T>(...)
```

Ambiguity!
That's why not `sort<T>`

Liskov Substitution Principle

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

As mentioned before, here T denotes something like "**any type**". It's called **universal parameter**.

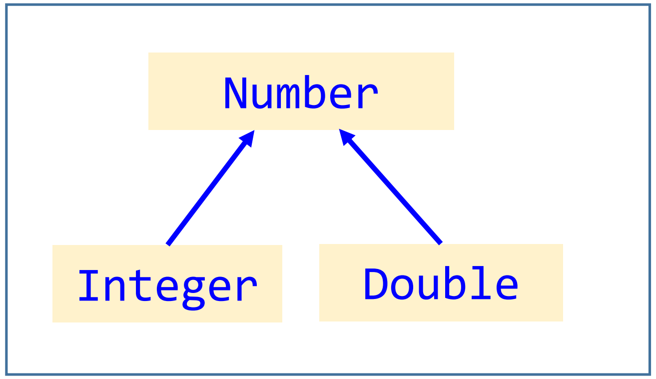


Liskov Substitution Principle

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

As mentioned before, here T denotes something like "**any type**". It's called **universal parameter**.

```
List<Integer> ints = new List<Integer>();
ints.extend(new Integer(1));   // correct
ints.extend(2);                // correct (boxing)
ints.extend(3.14);             // compile-time error!
Ints.extend(new Double(3.14)); // compile-time error!
```

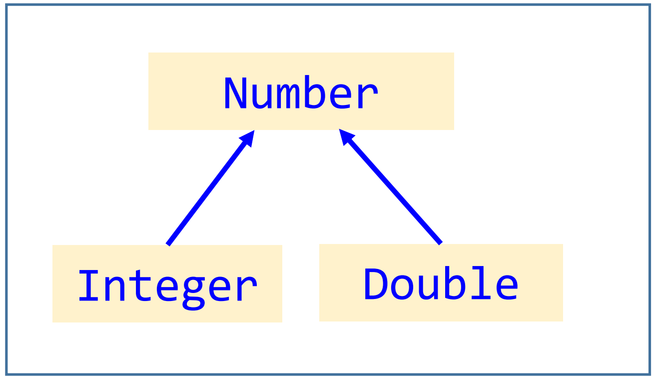


Liskov Substitution Principle

```
class List<T> {
    void extend(T v) { ... }
    void remove(T v) { ... }
    T elem(int i) { ... }
}
```

As mentioned before, here T denotes something like "**any type**". It's called **universal parameter**.

```
List<Integer> ints = new List<Integer>();
ints.extend(new Integer(1));   // correct
ints.extend(2);                // correct (boxing)
ints.extend(3.14);             // compile-time error!
Ints.extend(new Double(3.14)); // compile-time error!
```



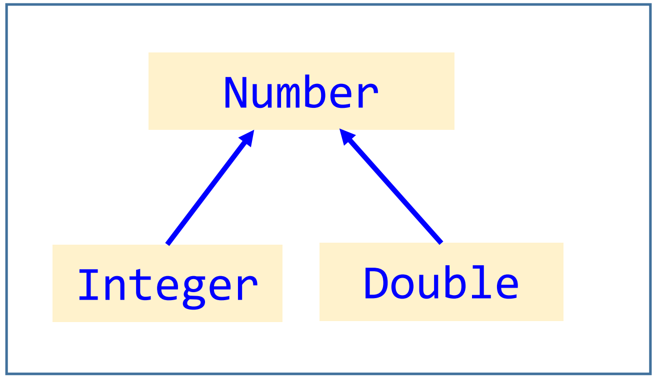

```
List<Number> nums = new List<Number>();
nums.extend(2);                // correct (boxing)
nums.extend(3.14);             // correct!!
```

WHY??

Liskov Substitution Principle

Subtyping definition

One type is a **subtype** of another type if they are related by an **`extends`** of **`implements`** clause

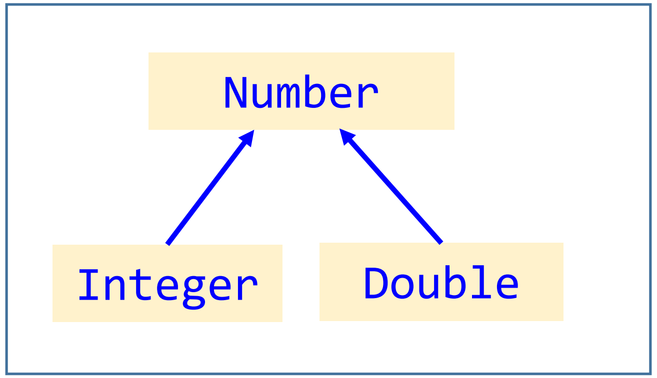


`Integer` and `Double` are subtypes of `Number`

Liskov Substitution Principle

Subtyping definition

One type is a **subtype** of another type if they are related by an **`extends`** of **`implements`** clause



`Integer` and `Double` are <u>subtypes</u> of `Number`

Liskov Substitution Principle (LSP)

Barbara Liskov

- A variable of a given type may be assigned a value of any subtype of that type.

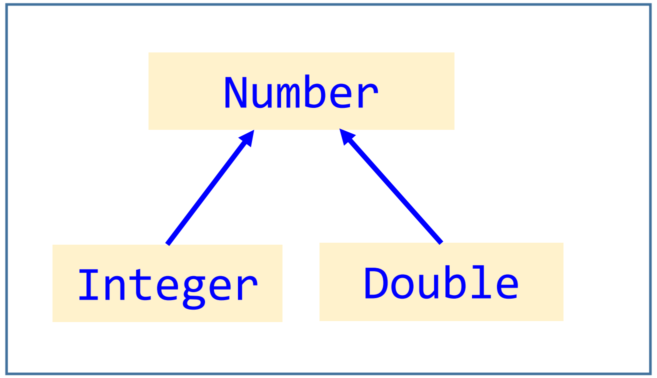
- A method of a parameter of a given type may be invoked with an argument of any subtype of that type.

If a method has a parameter of type `Animal`, it can be successfully called for any animal (`Lion`, `Frog` etc.)

Liskov Substitution Principle

Subtyping definition

One type is a **subtype** of another type if they are related by an **`extends`** of **`implements`** clause



`Integer` and `Double` are <u>subtypes</u> of `Number`

Barbara Liskov

Liskov Substitution Principle (LSP)

- A variable of a given type may be assigned a value of any subtype of that type.

- A method of a parameter of a given type may be invoked with an argument of any subtype of that type.

Sounds familiar, isn't it? ☺

Recall the notion of <u>dynamic types</u> introduced before!

If a method has a parameter of type `Animal`, it can be successfully called for any animal (`Lion`, `Frog` etc.)

A side-off remark: shorthands

Are you happy with such an awkward notation?

```
List<Integer> ints = new List<Integer>();
```

The same fragment is duplicated!

A side-off remark: shorthands

Are you happy with such an awkward notation?

```
List<Integer> ints = new List<Integer>();
```

The same fragment is duplicated!

This one looks much more concise, elegant, and easier to read

```
var ints = new List<Integer>();
```

A side-off remark: shorthands

Are you happy with such an awkward notation?

```
List<Integer> ints = new List<Integer>();
```

The same fragment is duplicated!

This one looks much more concise, elegant, and easier to read

```
var ints = new List<Integer>();
```

What's the type of `ints`?

- The compiler takes the type of `ints` from its initializer

Type inference

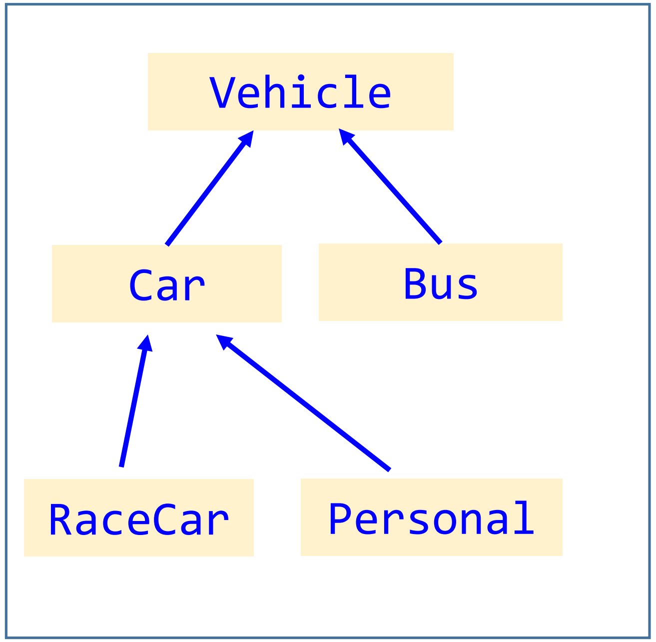
Requirements on Types

A problem

Suppose we want to develop a (generic) class that represents various kinds of **garages**.

```
class Garage<T> {
  // implementation:
  // a list (or array, or set) of vehicles
  // with some functionality (methods)
  void repair(T vehicle) { ... }
}
```

T represents a vehicle class



Requirements on Types

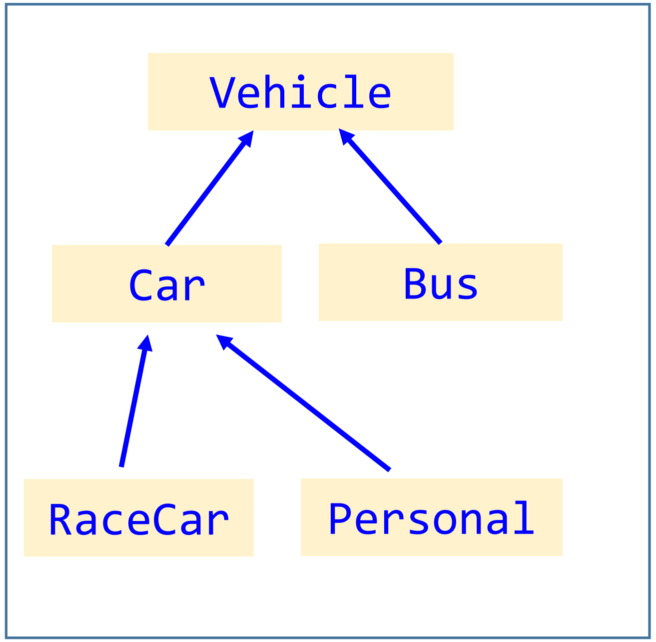
A problem

Suppose we want to develop a (generic) class that represents various kinds of **garages**.

```
class Garage<T> {
  // implementation:
  // a list (or array, or set) of vehicles
  // with some functionality (methods)
  void repair(T vehicle) { ... }
}
```

T represents a vehicle class

```
Garage<Personal> myCars = new Garage<Personal>();
Garage<Bus> BusStation = new Garage<Bus>();
...
```



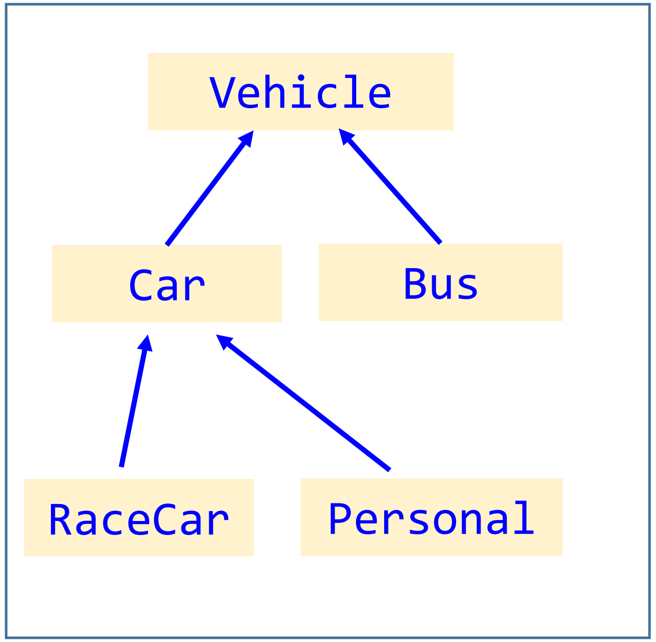
Requirements on Types

A problem

Suppose we want to develop a (generic) class that represents various kinds of **garages**.

```
class Garage<T> {
  // implementation:
  // a list (or array, or set) of vehicles
  // with some functionality (methods)
  void repair(T vehicle) { ... }
}
```

T represents a vehicle class




```
Garage<Personal> myCars = new Garage<Personal>();
Garage<Bus> BusStation = new Garage<Bus>();
...
Garage<Frog> lake = new Garage<Frog>();
```

Formally correct but definitely doesn't make sense **semantically**.

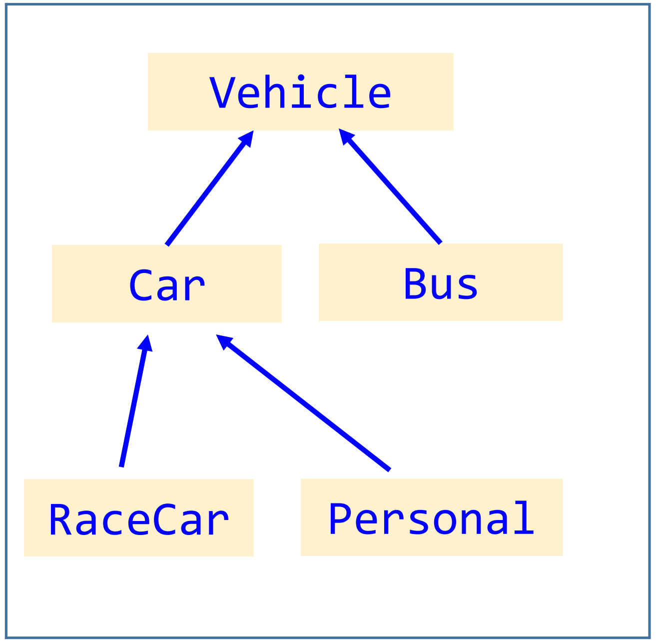
Moreover, the call `lake.repair()` can cause unpredictable runtime error.

Requirements on Types

The solution

To <u>restrict</u> possible set of type parameters

```
class Garage<T extends Vehicle> {
  // implementation:
  // a list (or array, or set) of vehicles
  // with some functionality (methods)
  void repair(T vehicle) { ... }
}
```




```
Garage<Personal> myCars = new Garage<Personal>();
Garage<Bus> BusStation = new Garage<bus>();
...
Garage<Frog> lake = new Garage<Frog>();
```

Compile-time error

Requirement on actual type:

Generic class Garage can be instantiated only by class Vehicle or its any subclass.

Requirements on Types

```
class Bank<T extends iAccount>
{
  T[] accounts;
  public Bank(T[] accs) { this.accounts = accs; }
    ...
}
```

```
interface iAccount
{
   int getId();
}
class Account implements iAccount
{
   ...
}
```

```
...
Account[] accounts = { new Account(), new Account(), ... };
Bank<Account> bank = new Bank(accounts);
...
Bank<Something> bank = new Bank(...);  // ERROR
```

The solution

To restrict possible set of type parameters

Requirement on actual type:

- Actual type for T formal type must be a class implementing iAccount interface.

Requirements on Types

```
class Bank<T1, T2 extends Person & iAccount>
{
    ...
}
```

Requirements on actual type:

- **No** requirements on `T1` formal type;
- Actual type for `T2` formal type must be a class implementing `iAccount` interface, and, at the same time, must be derived from class `Person`.

C#: Requirements on Types

For comparison

```
public class MyTemplate<Type1,Type2>
                     where Type1 : IComparable,
                           Type2 : MyInterface,
                           Type2 : MyBaseClass
{
    ...
}
```

Requirements on actual types:

- Actual type for `Type1` formal type must implement `IComparable` interface;
- Actual type for `Type2` formal type
  (a) must implement `MyInterface` interface, and
  (b) must be derived from `MyBaseClass` type.

Several interfaces can be specified as constraints for a certain type but **only one base class**.

C++: Requirements on Types

For comparison

The solution is:

- very powerful, very flexible, very detailed, and…

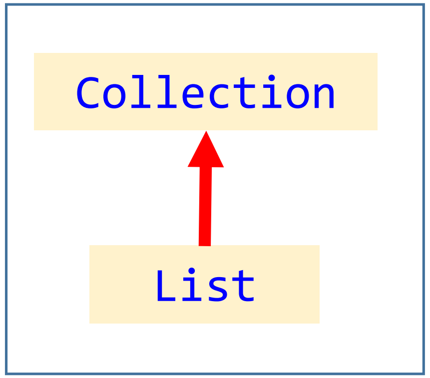
- **extremely complicated:**

The notion of concept

Wait for the next semester ☺

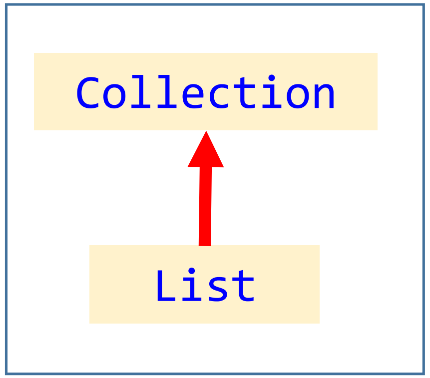
Variance: Preliminary Example

```
// Common features for various collections
class Collection<T> { ... }

// Features specific for lists
class List<T> extends Collection<T> { ... }
```

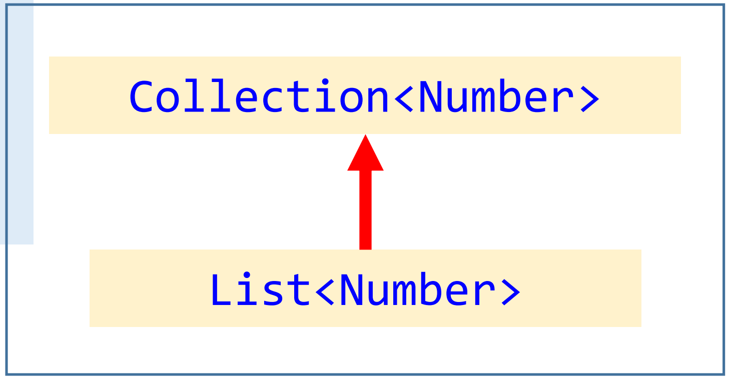


Variance: Preliminary Example

```
// Common features for various collections
class Collection<T> { ... }

// Features specific for lists
class List<T> extends Collection<T> { ... }
```




```
Collection<Integer> col = new Collection<Integer>();
...
List<Integer> lst = new List<Integer>();
...
col = lst;    // Substitution OR upcasting!
```



Variance: The Problem

Suppose there are two related classes:

```
class Base { ... }
class Derived extends Base { ... }
```

…and a collection: an array, a list, a set etc.

```
class Collection<T>
{
    ...
}
```

Variance: The Problem

Suppose there are two related classes:

```
class Base { ... }
class Derived extends Base { ... }
```

…and a collection: an array, a list, a set etc.

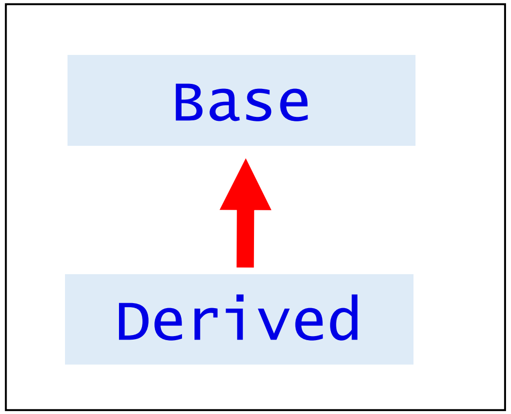
```
class Collection<T>
{
    ...
}
```

…And we have instantiated two classes out of `Collection`:

```
Collection<Base>
Collection<Derived>
```

The question:
What is relationship between these two collections?

Variance: Explanation



Variance: Explanation

`Base`

↑

`Derived`

Covariance

`Collection<Base>`

↑

`Collection<Derived>`

Typical for most cases;
intuitively obvious.

Variance: Explanation

Base

↑

Derived

Covariance

Collection<Base>

↑

Collection<Derived>

Typical for most cases;
intuitively obvious.

Contravariance

Collection<Base>

↓

Collection<Derived>

Seems to be bit artificial case.
However, sometimes it does
make sense.

Variance: Explanation

`Base`

$\uparrow$

`Derived`

Invariance

`Collection<Base>`

✗

`Collection<Derived>`

Typical (but **not ubitiqous**) for C++.

Contravariance

`Collection<Base>`

$\downarrow$

`Collection<Derived>`

Seems to be bit artificial case.
However, sometimes it does make sense.

Covariance

`Collection<Base>`

$\uparrow$

`Collection<Derived>`

Typical for most cases; intuitively obvious.

Variance: Example 1

Number

Integer

Let's **assume** that List<Integer> is a subtype of List<Number>: **covariance**

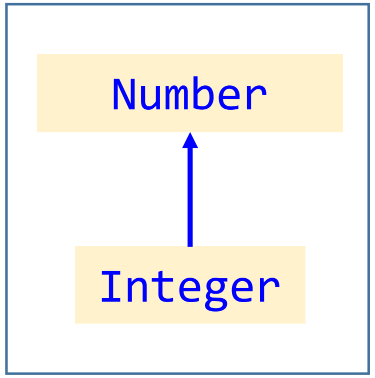
List<Number>

?

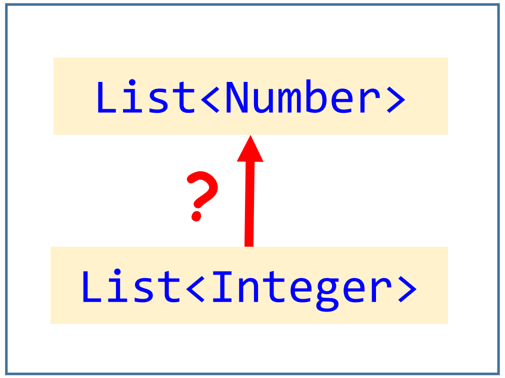
List<Integer>

```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Integer> ints = new List<Integer>();
ints.extend(1);
ints.extend(2);
List<Number> nums = ints;
nums.extend(3.14);
```

Variance: Example 1



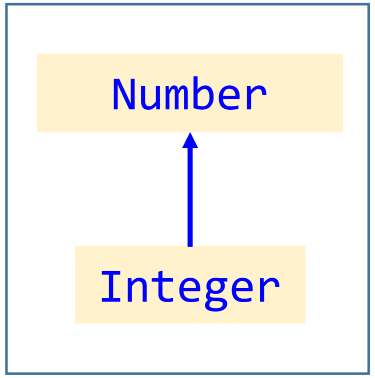
Let's **assume** that List<Integer> is a subtype of List<Number>: **covariance**



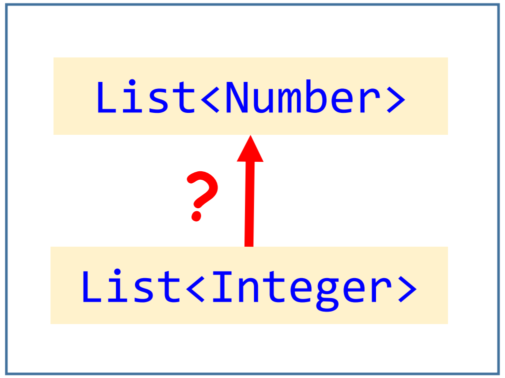

```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Integer> ints = new List<Integer>();
ints.extend(1);
ints.extend(2);
List<Number> nums = ints;
nums.extend(3.14);
```

If **covariance** then it's legal: List<Integer> is a subtype of List<Number>

Variance: Example 1



Let’s **assume** that List<Integer> is a subtype of List<Number>: **covariance**




```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Integer> ints = new List<Integer>();
ints.extend(1);
ints.extend(2);
List<Number> nums = ints;
nums.extend(3.14);
```

If **covariance** then it’s legal: List<Integer> is a subtype of List<Number>

If **covariance** then **it’s legal as well**. However, it’s **a nonsense**: we try to add Number to <u>the same</u> list of Integers.

Variance: Example 1

Number

↑

Integer

Let's **assume** that List<Integer> is a subtype of List<Number>: **covariance**

List<Number>

? ↑

List<Integer>

Conclusion:
List<Integer>
is <u>not a subtype</u> of
List<Number>
OR: LSP doesn't apply

```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Integer> ints = new List<Integer>();
ints.extend(1);
ints.extend(2);
List<Number> nums = ints;
nums.extend(3.14);
```

If **covariance** then it's legal: List<Integer> is a subtype of List<Number>

If **covariance** then **it's legal as well**. However, it's **a nonsense**: we try to add Number to <u>the same</u> list of Integers.

Variance: Example 2

Number

Integer

Let's **assume** that `List<Integer>` is a supertype of `List<Number>`: **contravariance**

List<Number>

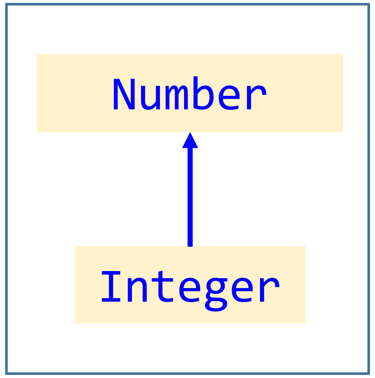
?

List<Integer>

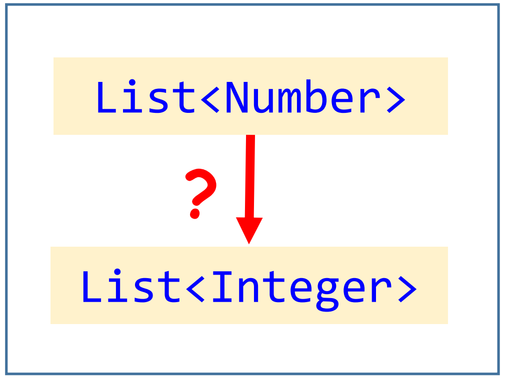
```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Number> nums = new List<Number>();
nums.extend(2.78);
nums.extend(3.14);
List<Integer> ints = nums;
```

If **contrvariance** then it's **a nonsense**: list of `Integer`s refers to the list of `Number`s.

Variance: Example 2



Let's **assume** that `List<Integer>` is a supertype of `List<Number>`: **contravariance**

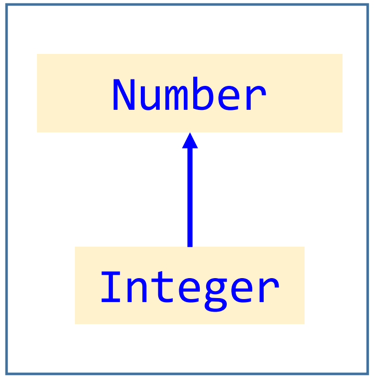


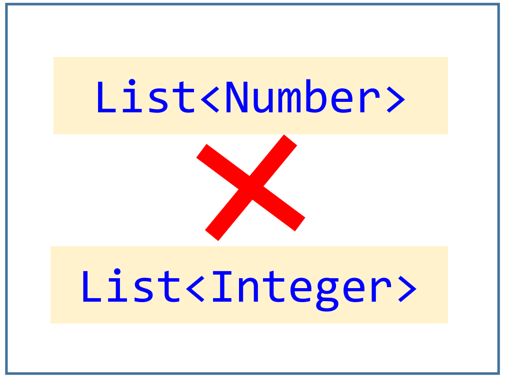

```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Number> nums = new List<Number>();
nums.extend(2.78);
nums.extend(3.14);
List<Integer> ints = nums;
```

If **contrvariance** then it's **a nonsense**: list of `Integers` refers to the list of `Numbers`.

Conclusion: `List<Integer>` **is <u>not a supertype</u> of** `List<Number>`
OR: LSP doesn't apply

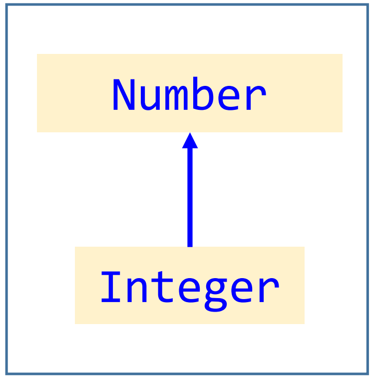
Variance: Conclusion

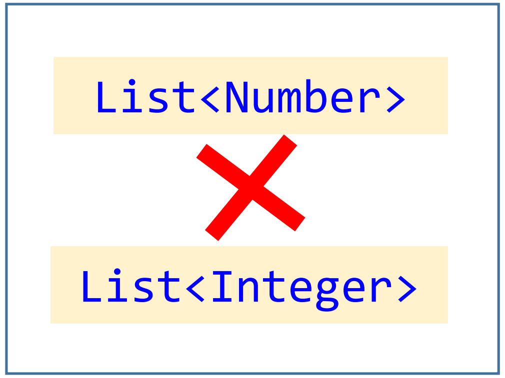




List<Integer> and List<Number>
are **invariant**

Variance: Conclusion





`List<Integer>` and `List<Number>`
are **invariant**

However, **arrays** behave quite <u>differently</u>:
`Integer[]` <u>is a subtype</u> `Number[]`.

Variance & Wildcards

How to overcome invariance for generic classes?

We would like addAnotherList to add list of elements that consists of elements of type T and T's subtypes.

```
class List<T> {
   public void addAnotherList(...) { ... }
   ...
}
```

?

Variance & Wildcards

How to overcome invariance for generic classes?

We would like addAnotherList to add list of elements that consists of elements of type T and T's subtypes.

```
class List<T> {
  public void addAnotherList(...) { ... }
  ...
}
```

?

addAnotherList can be invoked with any List with elements of type T **OR** with elements of **any subtype** of T.

```
class List<T>
{
  public void addAnotherList (List<? extends T> newLst) { ... }

  ...
}
```

Variance & Wildcards

How to overcome invariance for generic classes?

We would like addAnotherList to add list of elements that consists of elements of type T and T's subtypes.

```
class List<T> {
  public void addAnotherList(...) { ... }
  ...
}
```

?

```
class List<T>
{
  public void addAnotherList (List<? extends T> newLst) { ... }
  public void addAnotherList2(List<? super T> newLst) { ... }
  ...
}
```

addAnotherList can be invoked with any List with elements of type T **OR** with elements of **any subtype** of T.

addAnotherList2 can be invoked with any List with elements of type T **OR** with elements of **any supertype** of T.

Variance: The Exrecise

Not a task but **recommended**

1. Implement generic class `List<T>`.
   The `List` interface from `java.util` package can be used as a prototype.

2. Implement `addAnotherList` and `addAnotherList2` methods of `List<T>`.

3. Check how these methods work for `List<Number>` and `List<Integer>`.

* 4. Think where these methods are appropriate and where **they're not**.
   Hint: think about difference between assigning values and reading them. Try to write code that uses methods.