Introduction to Programming
Part I

Lecture 13
Type Variance
Java Lambdas & Functional Programming

Eugene Zouev
Fall Semester 2021
Innopolis University

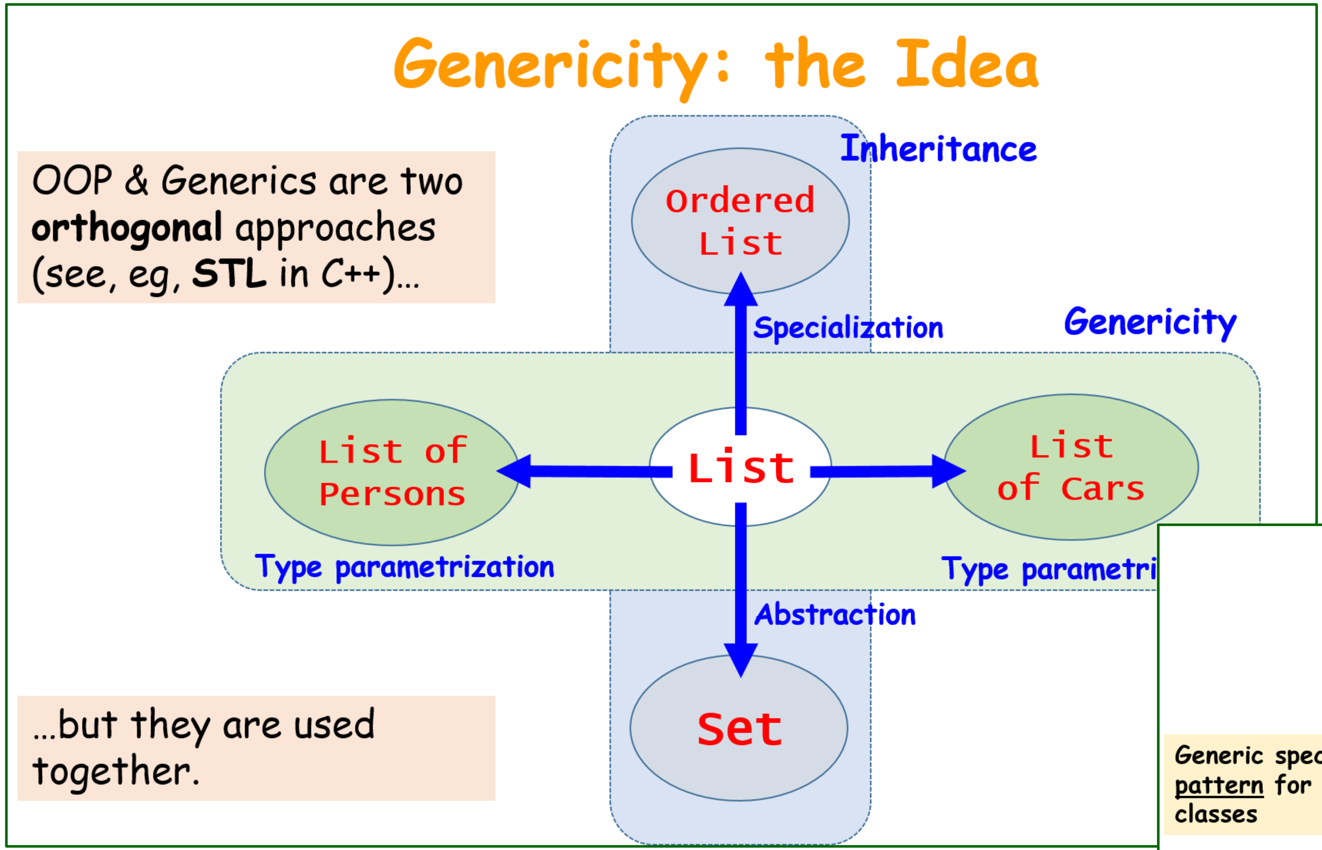
The Previous Lecture: Java Generics

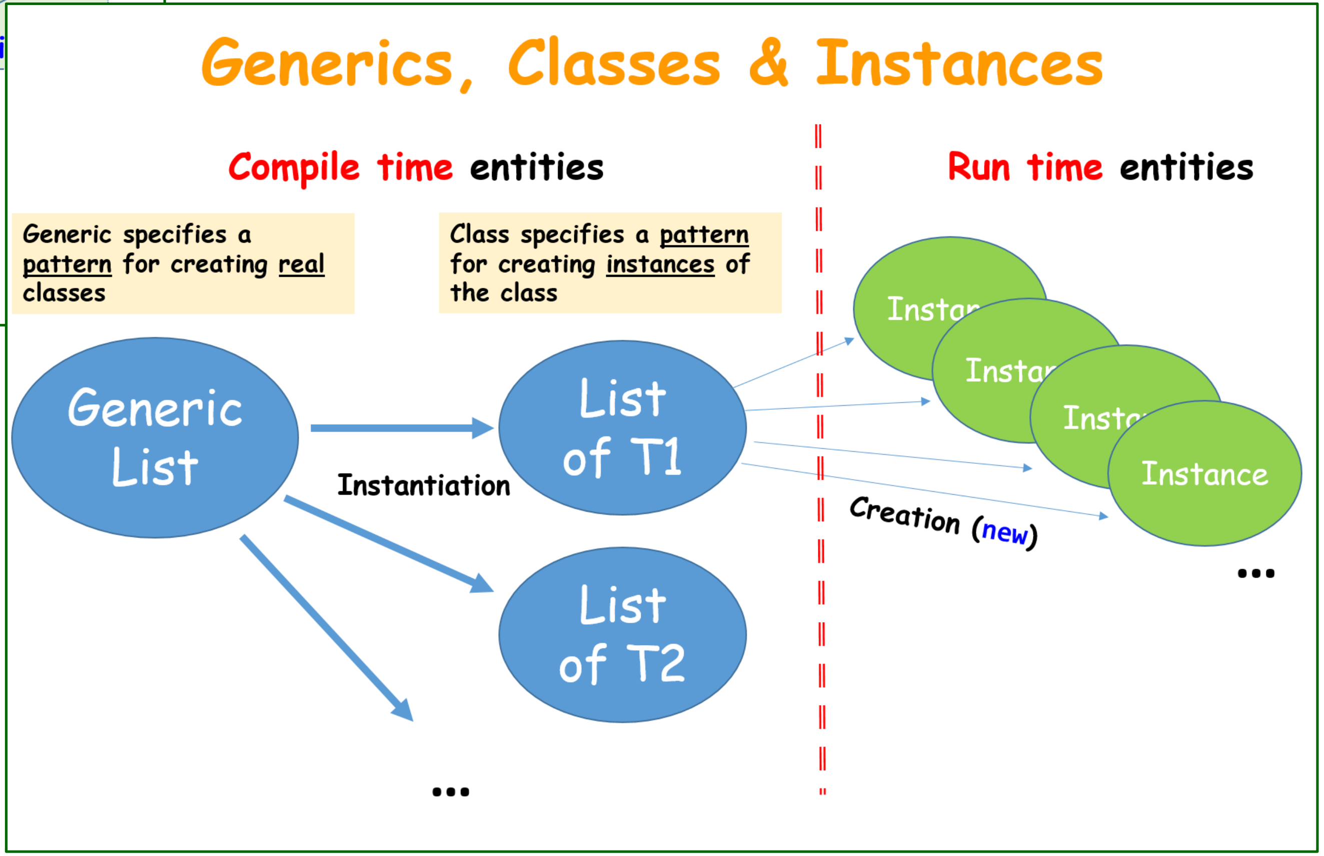
- The idea of genericity
- The life without genericity
- Boxing & unboxing
- Generics in Java: type parametrization
- Requirements on actual types
- The notion of **variance**
- Variance & wildcards

Plan for the rest of the course

- 12 Java generics
- 13 Java lambdas
- 14 Java miscellaneous

From the prev lecture





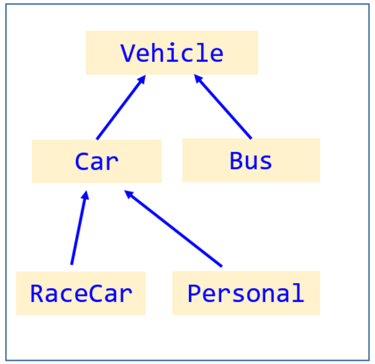
From the prev lecture

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.

26/38

Approach with Type Parametriz

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

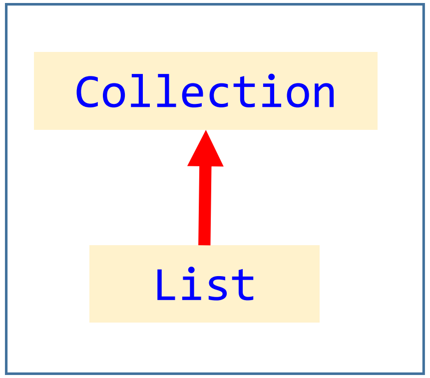
- T denotes something like “**any type**”. It’s called **universal parameter**.
- The whole `List<T>` declaration specifies a list whose all elements are of *some type* T.
- The `List<T>` declaration is (still) an abstraction (“generic”, or “template”): in order to use it we have to **instantiate** it 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.

Part 1
Type Variance

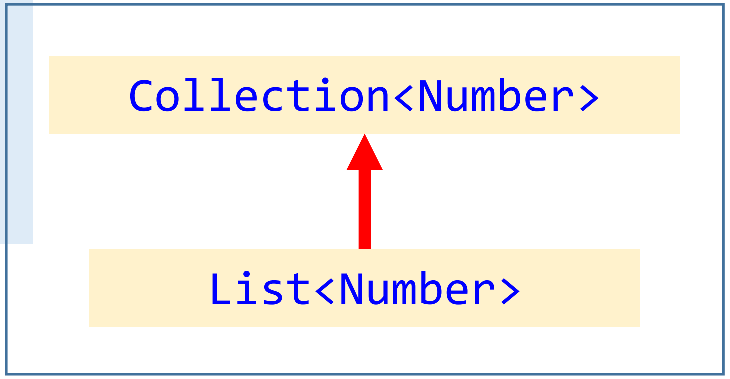
Variance: Preliminary Example

```
// 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>
{
    ...
}
```

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

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

The question:
What is relationship between these two collections?

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.

Invariance

Collection<Base>

Collection<Derived>

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

Variance: Example 1

Number

Integer

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

List<Number>

?

List<Integer>

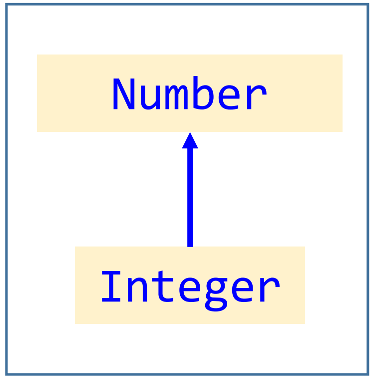
Conclusion:
List<Integer>
is <u>not a subtype</u> of
List<Number>
OR: LSP doesn't apply

```
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);
```

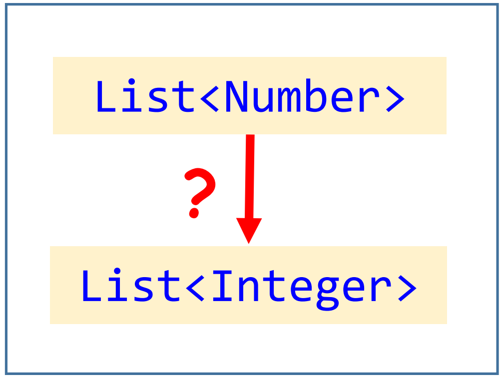
If **covariance** then it's legal: List<Integer> is a subtype of List<Number>

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



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

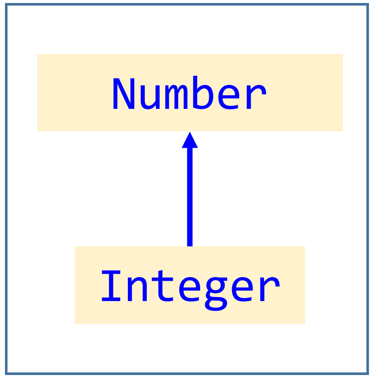


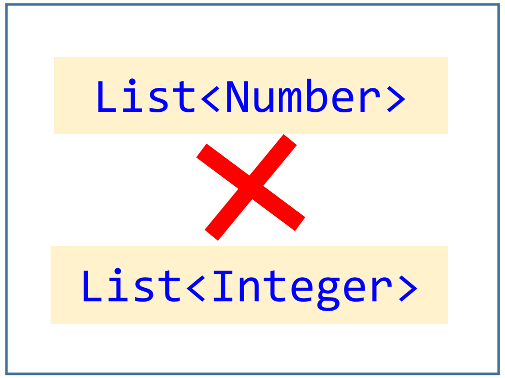

```
class List<T> {
  void extend(T v) { ... }
  ...
}
...
List<Number> nums = new List<Number>();
nums.extend(2.78);
nums.extend(3.14);
List<Integer> ints = nums;
```

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**

However, **arrays** behave quite differently:
Integer[] is a subtype 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(...) { ... }
  ...
}
```

?

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

`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.

Part 2
Java & Functional Programming

Why Learn & Use Functional?

Functional is the Trend!

Almost every modern programming language has at least some "functional" features.

- **C++**: lambda expressions, function types, functions without side effects, type inference
- **C#**: function types ("delegates"), function literals, type inference
- **Java 8**: lambda expressions
- **Swift**: functions as values; closures; local functions
- **Rust**: functions as variables, as arguments, as return values; anonymous functions

Imperative vs Functional

- Functional programming treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- In functional code, the output value of a function depends only on its arguments, so calling a function with the same value for an argument always produces the same result ("pure functions").
- Imperative programming changes state with **assignment**.
  Imperative programming has subroutines, but these are not mathematical functions. They can have side effects that may change a program's state. Because of this the same language expression can result in different values at different times depending on the state of the executing program.

Imperative vs Functional

C/C++
Java
C#
Kotlin
Swift
Rust
Eiffel

Imperative programming	Functional programming
State(data) and code	No state
Routine: action or query	Functional as transformation of elements of one set into the other
Routine: procedure or function	High order functions
Side effects – change in global data, in class attributes	No side effects
Aligned with HW architecture	Aligned with mathematical approaches
Statements	Expressions and declarations
Mutable variables	Immutable variables
Loops	Recursion

Common Lisp
Scheme
Clojure
Scala
Erlang
Ocaml
Haskell
F#
Wolfram

Imperative vs Functional

From Tutorial 1

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

Euclid algorithm:
Finds the greatest common denominator for two numbers

Наибольший общий делитель

Imperative paradigm

Important points:

- The algorithm is organized as a series of **steps**.
- The variables **change their values** on each step.
- There are **three local variables** used in the algorithm.
- This is the **iterative** algorithm (with loop).

Imperative vs Functional

From Tutorial 1

```
int gcd(int x, int y)
{
  int a = x, b = y;
  while ( a != 0 )
  {
    int temp = a;
    a = b % a
    b = temp
  }
  return b;
}
```

```
int gcd(int x, int y)
{
  return (y == 0) ? x : gcd(y, x%y);
}
```

Often, “conventional” languages can be used to program in the functional style!

Functional paradigm

Important points:

- **No** local variables.
- Variables (parameters) do not **change their values**.
- This is the **recursive** algorithm: recursion is used instead of iteration
- The code is much more concise and readable.

Imperative vs Functional: Java Case

```
public class Example1
{
   public int sum(int x, int y)
   {
      return x + y;
   }
}
```

```
public class Example2
{
   private int value;
   public int add(int next)
   {
      this.value += next;
      return this.value;
   }
}
```

"Pure" function:

- The function doesn't have side effects: no changes outside of the function
- The result of the function depends only on its input parameters

Ordinary function:

- The function produces the side effect: it's changes the state of the object it belongs to.
- The result of the function depends on the current state of the object it belongs to.

"Pure" Functions: Advantages

- If the result of a pure function is not used, it can be completely removed without affecting other expressions.
- If a pure function is called with arguments that cause no side-effects, the result is constant and <u>can be cached and immediately returned on next call</u> with the same arguments (referential transparency)
- If there is no data dependency between two pure expressions, their execution can be done <u>in any order or in parallel</u> (thread-safety).
- If the entire language does not allow side-effects, then <u>any evaluation strategy can be used</u>; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program.

Functions as First-Class Objects

Functions are **values** - just as integers, arrays etc.

- Function is an abstraction of an **operation**.
- Define functions anywhere (just as other variables).
- "Constant" (or unnamed) functions are allowed (just as integer constants):
  functional constants
  literal functions
  anonymous functions

Functions as First-Class Objects

```
final int x = 777;
```

- 777 is unnamed constant; it represents itself
- The declaration binds the constant with a name; such a binding is hard

```
x = 999;
```

```
int sum(int x, int y) { return x+y; }
```

- The same approach applies: function parameters and the function body can be considered as a value (constant), …
- …and function declaration binds such a value with the name `sum`.

Therefore, all we need to know about a function is:

Parameters

```
(int x, int y)
```

Body

```
{ return x+y; }
```

…And we could bind parameters and body with any name!!

Java Basic Notation

Lambda expression

```
(int x, int y) -> { return x+y; }
```

This is a usual value ("functional value") that can be
- Passed as an argument to some function
- Bound with (assigned to) some variable

```
______ fun = (int x, int y) -> { return x+y; }
```

What's the type
of the lambda expression???

Java Functional Conventions

The rule:

The type of a lambda expression is an **interface** with the single abstract method whose signature matches the lambda's signature.

```
interface Func {
   int action(int x, int y);
}

class someClass {
   Func lambda = (int x, int y) -> { return x+y; }
}
```

Functional interface

Functional interface implementation

However, if an interface has more than one method without implementation it will no longer be a functional interface, and could not be implemented by a Java lambda expression.

Java Functional Conventions

Zero Parameters

If the lambda expression takes no parameters, it can be like this:

```
() -> { System.out.println("Zero parameter lambda"); }
```

```
() -> System.out.println("Zero parameter lambda")
```

If the body contains only one statement then braces can be omitted

One Parameter

If the lambda takes one parameter, it can be like this:

```
(int p) -> System.out.println("One parameter lambda " + p)
```

```
(p) -> System.out.println("One parameter lambda " + p)
```

The type of parameter can be inferenced from the body

```
p -> System.out.println("One parameter lambda " + p)
```

Parentheses can be omitted

Java Functional Conventions

Multiple Parameters

If the lambda expression takes multiple parameters, the parameters need to be listed inside parentheses

```
(int p1, int p2) -> System.out.println("Multiple pars " + p1+p2)
```

```
(p1, p2) -> System.out.println(" Multiple pars " + p1 + p2)
```

References to Methods

```java
public interface MyPrinter
{
  public void print(String s);
}
```

Functional interface

```java
MyPrinter printer =
   s -> { System.out.println(s); };
```

Lambda instance:
The only action of the instance is passing parameter to `println`…

```java
MyPrinter printer = System.out::println;
```

…therefore, it can be replaced just for the method reference

Method reference:
`println` hast the same signature as `print` from `MyPrinter`

The following methods can be referenced:

- Static methods
- Instance methods
- Constructors

Using Lambdas

```
public interface MyPrinter
{
  public void print(String s);
}
```

Functional interface

```
MyPrinter printer = System.out::println;
```

Here, `printer` is a **functional object**: <u>it refers to some function</u>. Therefore, we can <u>invoke</u> the method it refers to either directly OR indirectly, via the reference:

```
void someOtherMethod()
{
  System.out.println("some string");

  printer.print("some string");
}
```

Using Lambdas

```
public interface Deserializer
{
  public int deserialize(String v1);
}
```

Functional interface

```
public class StringConverter
{
  public int convertToInt(String v1)
  {
    return Integer.valueOf(v1);
  }
}
```

Method convertToInt's signature matches the functional interface

```
StringConverter stringConverter =
                    new StringConverter();
Deserializer des =
            stringConverter::convertToInt;
```

des refers to the convertToInt method of the instance of StringConverter created before

des is the reference to the <u>instance</u> method

Using Lambdas

```
public interface Finder
{
   public int find(String s1, String s2);
}
```

Functional interface

```
public class MyClass
{
  public static int doFind(String s1, String s2)
  {
    return s1.lastIndexOf(s2);
  }
}
```

Method `doFind`'s signature matches the functional interface

```
Finder finder = MyClass::doFind;
```

`finder` variable refers to the `doFind` - <u>static</u> method of the `MyClass` class

`finder` is the reference to the <u>class</u> method

Using Lambdas

```
public interface Factory
{
  public String create(char[] val);
}
```

Functional interface

```
public class String { // from the Standard Library
  ...
  public String(char[] chars)
  {
    return s1.lastIndexOf(s2);
  }
  ...
}
```

There are a few constructors in this library class

```
Factory factory = String::new;

Factory factory2 = chars -> new String(chars);
```

factory variable refers to the String's <u>constructor</u> with the given parameter type

factory is the reference to the <u>constructor</u>

Lambda: Capturing Variables

```
public class SomeClass
{
  public interface Factory {            // Functional interface
     public String create(char[] val);
  }
  String myString = "Test";

  // The variable refers to a lambda (an unnamed function)
  MyFactory myFactory = (chars)-> return myString + ":" + new String(chars);
}
```

Important point:

Lambda refers to a variable declared outside of the lambda body.

It is said that the lambda **captures** the variable, and such a lambda is often called **closure**.

This is possible if, and only if, the variable being captured is "**effectively final**", meaning it does not change its value after being assigned. If the `myString` variable had its value changed later, the compiler issues the error message.
(Static variables do not follow this rule.)