Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Lambda Expressions

Definition

- Lambda expression (or lambda) - anonymous block of code
- Notation: We will use ***λ ex*** for *lambda expression*
- A ***λ ex*** describes an anonymous function
- General syntax
  `(<LambdaParametersList>) -> { <LambdaBody> }`
- Lambda Body may
  – Declare local variables;
  – Use statements including break, continue, and return;
  – Throw exceptions, etc.

A ***λ ex*** has no:

- Name
- Return type
  – It is inferred by compiler from the context of its use and from its body
- Throws clause
  – It is inferred from the context of its use and its body
- No generics with ***λ ex***

Definition

Examples

(int x) -> x + 1
(int x, int y) -> x + y
(int x, int y) -> { int max = x > y ? x : y; return max; }
() -> { }
() -> "OK"
(String msg) -> { System.out.println(msg); }
msg -> System.out.println(msg)
(String str) -> str.length()

Note1. Sometime type parameters could be omitted.
The compiler will infer them from context
(x, y) -> x + y

- Explicit-typed ***λ ex*** - declares the types of its parameters
- Implicit-typed ***λ ex*** - …

Note 2. The parentheses can be omitted only if the single parameter also omits its type
Note 3. A block statement is enclosed in braces; single expression – no braces

Definition

***λ ex* type**

- In Java every expression must have a type
- There are two types of expressions
  - Standalone expressions – type can be determined without knowing the context of use;
    - Examples: new Integer(3) => the type is Integer; new ArrayList<Double>
  - Poly Expressions – different types in different contexts => ***λ ex*** are poly expressions
- ***λ ex*** type is **Functional Interface**
  - The exact type depends on the **context** in which it is used
- **Note**. Poly expressions existed prior Java 8 (example ArrayList<>() is a poly expression)

  Examples of context use:
  ArrayList<Integer> idList = new ArrayList<>();
  ArrayList<String> nameList = new ArrayList<>();

Definition

Inferring target type (the type of a *λ ex*)

- The compiler infers the type of a ***λ ex***
- The context in which a ***λ ex*** is used expects a type – this type is called the *target type*
- Consider the example (below

  T t = <LambdaExpression>;
  - The target type of the ***λ ex*** is T

Inferring rules used by compiler

- T must be a **Functional Interface** type
- ***λ ex*** has the same number and type of parameters as the abstract method of T
- For an implicit ***λ ex***, parameters types are inferred from the abstract method of T
- The type of the returned value from the body of the ***λ ex*** should be assignment compatible to the return type of the abstract method of T
- If the body of the ***λ ex*** throws any checked exceptions, they must be compatible with the declared throws clause of the abstract method of T
- It is a compile-time error to throw checked exceptions from the body of a ***λ ex***, if its target type's method does not contain a throws clause.

Definition

Examples of target typing

Examples from [Kishori]

```
@FunctionalInterface
public interface Adder {
  double add(double n1, double n2);
}

@FunctionalInterface
public interface Joiner {
  String join(String s1, String s2);
}
```

Consider the assignment statements

```
  Adder adder = (x, y) -> x + y; // the type of λ ex is Adder
  Joiner joiner = (x, y) -> x + y; // // the type of λ ex is Joiner
```

See how the type of the ***λ ex*** is Adder in one context and Joiner in another context

Definition

Using *λ ex* in programs – Example 1

```
// not much differences as before Java 8
public class Test1 {
  public static void main(String[] args) {
    Adder adder = (x, y) -> x + y; // // Creates an Adder using a lambda expression
    Joiner joiner = (x, y) -> x + y; // Creates a Joiner using a lambda expression
    double sum1 = adder.add(10.34, 89.11); // Adds two doubles
    double sum2 = adder.add(10, 89); // Adds two ints
    String str = joiner.join("Hello", " lambda"); // Joins two strings
  }
}
```

The results
sum1 = 99.45
sum2 = 99.0
str = Hello lambda

Definition

Using *λ ex* in programs – Example 2

// passing Functional Interfaces *as*
// arguments to methods

```
public class Lambda1 {
 public void testAdder(Adder adder) {
  double x = 1.1;
  double y = 2.2;
  double sum = adder.add(x, y);
  System.out.print("Using an Adder:");
  System.out.println(x + " + " + y + " = " + sum);
 }

 public void testJoiner(Joiner joiner) {
  String s1 = "Hello";
  String s2 = "World";
  String s3 = joiner.join(s1,s2);
  System.out.print("Using a Joiner:");
  System.out.println("\"" + s1 + "\" + \"" + s2 +
            "\" = \"" +s3 + "\"");;
 }
}
```

Consider the code

1: Lambda1 lbd1 = new Lambda1();
2: **lbd1.testAdder((x, y) -> x + y);**

How is going?

1: creates an object of the Lambda1 class

2: calls the testAdder() method on the object, passing the ***λ ex*** of (x, y) -> x + y

- The compiler must infer the type of the ***λ ex***
- The target type of the ***λ ex*** is the type Adder because the argument type of the testAdder(Adder adder) is Adder.
- The rest of the target typing process is the same as in the assignment statement before
- Finally, compiler infers that type of the ***λ ex*** is Adder

Definition

Using *λ ex* in programs – Example 3

// passing *λ ex as* arguments to methods

```
public class TestLambda1 {
 public static void main(String[] args) {
  Lambda1 lbd1 = new Lambda1();

  lbd1.testAdder((x, y) -> x + y);    // Call the testAdder() method
  lbd1.testJoiner((x, y) -> x + y);   // Call the testJoiner() method

  // The Joiner will add a space between the two strings
  lbd1.testJoiner((x, y) -> x + " " + y);

 // The Joiner will reverse the strings and join resulting strings
 // in reverse order adding a comma in between
  lbd1.testJoiner((x, y) -> {
   StringBuilder sbx = new StringBuilder(x);
   StringBuilder sby = new StringBuilder(y);
   sby.reverse().append(",").append(sbx.reverse());
   return sby.toString();
  });
 }
}
```

Output:

```
Using an Adder:1.1 + 2.2 = 3.3
Using a Joiner:"Hello" + "World" = "HelloWorld"
Using a Joiner:"Hello" + "World" = "Hello World"
Using a Joiner:"Hello" + "World" = "dlroW,olleH"
```

Comments

- The testJoiner() method was called three times
- Each time it displayed different results because different lambda expressions were passed to this method
- testJoin method was parametrized
- ***Behavior parametrization*** **(or** ***passing code as data*****)**
  - Changing the behavior of a method through its parameters
  - The code is passed encapsulated in lambda expressions to methods as if it is data

Definition

The case of overloaded methods

- In some contexts (ex. passing ***λ ex*** to overloaded methods) the compiler cannot infer the type of a ***λ ex*** or may generate ambiguity
  - Those contexts do not allow the use of ***λ ex***
  - Some contexts may allow using ***λ ex***, but the use itself may be ambiguous to the compiler

Three methods to help the compiler resolve the ambiguity

- If the ***λ ex*** is implicit, make it explicit by specifying the type of the parameters.
- Use a cast
- Do not use the ***λ ex*** directly as the method argument. First, assign it to a variable of the desired type, and then, pass the variable to the method

Functional Interfaces (FI)

- Functional Interface - an interface that has **exactly one abstract method**
- The following method types in an interface do not count for defining a FI
  - Default methods
  - Static methods
  - Public methods inherited from the Object class
- Annotation: @FunctionalInterface

Example

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);       // abstract method
   boolean equals(Object obj); // re-declaration of equals
   // … Many static and default methods not shown here
}
```

- A FI represents one type of functionality/operation in terms of its single abstract method
- This is the reason why the target type of a lambda expression is always a FI

- A FI can be generic
- Example

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```


Functional Interfaces (FI)

Generic FI

Mapper an example of Generic Functional Interface

- Mapper example (generic FI with a type parameter T)

```
@FunctionalInterface
public interface Mapper<T> {
 int map(T source);    // the abstract method

 // A generic static method
 public static <U> int[] mapToInt(U[] list,  Mapper<? super U> mapper)  {
     int[] mappedValues = new int[list.length];
     for (int i = 0; i < list.length; i++) {
       // maps the object to an int
       mappedValues[i] = mapper.map(list[i]);
     }
     // returns the int array of mapped values
     return mappedValues;
  }
}
```

Functional Interfaces (FI)

Generic FI

Mapper an example of Generic Functional Interface (cont.)

How to use λ *ex* to instantiate the Mapper<T> interface for mapping a String array and an Integer array to int arrays

```
public class MapperTest {
  public static void main(String[] args) {

    // Map names using their length
    System.out.println("Mapping names to their lengths:");
    String[] names = {"Popescu", "Pop", "'Popica"};
    int[] lengthMapping = Mapper.mapToInt(names, (String name) -> name.length());
    printMapping(names, lengthMapping);

    // Map Integers to their squares
    System.out.println("\nMapping integers to their squares:");
    Integer[] numbers = {7, 3, 67};
    int[] countMapping = Mapper.mapToInt(numbers, (Integer n) -> n * n);
    printMapping(numbers, countMapping);
  }

  public static void printMapping(Object[] from, int[] to) {
    for(int i = 0; i < from.length; i++) {
      System.out.println(from[i] + " mapped to " + to[i]);
    }
  }
}
```

Mapping names to their lengths:
Popescu mapped to 7
Pop mapped to 3
Popica mapped to 5
Mapping integers to their squares:
7 mapped to 49
3 mapped to 9
67 mapped to 4489

Functional Interfaces (FI)

Common Functional Interfaces defined in java.util.function [Kishori]

Interface Name	Method	Description
Function<T,R>	R apply(T t)	Represents a function that takes an argument of type T and returns a result of type R.
BiFunction<T,U,R>	R apply(T t, U u)	Represents a function that takes two arguments of types T and U, and returns a result of type R.
Predicate<T>	boolean test(T t)	In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument.
BiPredicate<T,U>	boolean test(T t, U u)	Represents a predicate with two arguments.
Consumer<T>	void accept(T t)	Represents an operation that takes an argument, operates on it to produce some side effects, and returns no result.
BiConsumer<T,U>	void accept(T t, U u)	Represents an operation that takes two arguments, operates on them to produce some side effects, and returns no result.
Supplier<T>	T get()	Represents a supplier that returns a value.
UnaryOperator<T>	T apply(T t)	Inherits from Function<T,T>. Represents a function that takes an argument and returns a result of the same type.
BinaryOperator<T>	T apply(T t1, T t2)	Inherits from BiFunction<T,T,T>. Represents a function that takes two arguments of the same type and returns a result of the same.

Note. Some of the listed FI also contain default or static methods

Functional Interfaces (FI)

Function<T, R> Interface

Interface **Function**

Abstract method

R **apply**(T t) – the abstract method which applies this function to the argument

Default and static methods

default <V> Function<T, V> **andThen**(Function<? super R, ? extends V> after)

default <V> Function<V, R> **compose**(Function<? super V,? extends T> before)

static <T> Function<T, T> **identity**()

- andThen
  - Returns a composed Function that first applies this Function to the argument and then applies the specified **after** function to the result
- compose
  - Returns a composed function that first applies the before function to its input and then applies this function to the result
- identity
  - Returns a function that always returns its input argument

Functional Interfaces (FI)

Function<T, R> Interface

Examples

```
// Create two functions
Function<Long, Long> square = x -> x * x;
Function<Long, Long> addOne = x -> x + 1;

// Compose functions from the two functions
Function<Long, Long> squareAddOne = square.andThen(addOne);
Function<Long, Long> addOneSquare = square.compose(addOne);

// Get an identity function
Function<Long, Long> identity = Function.<Long>identity();

// Test the functions
long num = 5L;
System.out.println("Number : " + num);
System.out.println("Square and then add one: " +
                        squareAddOne.apply(num));
System.out.println("Add one and then square: " +
                        addOneSquare.apply(num));
System.out.println("Identity: " + identity.apply(num));
```

Output

Number: 5
Square and then add one: 26
Add one and then square: 36
Identity: 5

Functional Interfaces (FI)

Function<T, R> Interface

Chaining lambda functions

- A function may be composed of by many functions (not only 2)
- Hints to the compiler may be necessary to disambiguate

- Example of a chain of 3 functions

```
// Square the input, add one to the result, and square the result
Function<Long, Long> chainedFunction = ((Function<Long, Long>)(x -> x * x))
                                        .andThen(x -> x + 1)
                                        .andThen(x -> x * x);
System.out.println(chainedFunction.apply(3L));
```

Output
100

Functional Interfaces (FI)

Predicate<T> Interface

Interface **Predicate**

Abstract method

boolean test(T t) – the abstract method

Default and static methods (they allow compose a predicate based on other predicates and logical operators NOT, AND, OR (all these methods can be chained to create complex predicates)

```
// negates the original predicate
default Predicate<T> negate()

// returns the short circuited logical AND (OR) predicate of this predicate and argument predicate
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)

// returns a predicate that tests if the specified targetRef is equal to the
// specified argument for the predicate according to
// Objects.equals(Object o1, Object o2)
// Note. If two inputs are null this predicate evaluates to true
static <T> Predicate<T> isEqual(Object targetRef)
```

Functional Interfaces (FI)

Predicate<T> Interface

Examples [Kishori]

```
// Create some predicates
Predicate<Integer> greaterThanTen = x -> x > 10;
Predicate<Integer> divisibleByThree = x -> x % 3 == 0;
Predicate<Integer> divisibleByFive = x -> x % 5 == 0;
Predicate<Integer> equalToTen = Predicate.isEqual(10);

// Create complex predicates using NOT, AND, and OR on other predcates
Predicate<Integer> lessThanOrEqualToTen = greaterThanTen.negate();
Predicate<Integer> divisibleByThreeAndFive = divisibleByThree.and(divisibleByFive);
Predicate<Integer> divisibleByThreeOrFive = divisibleByThree.or(divisibleByFive);

// Test the predicates
int num = 10;
System.out.println("Number: " + num);
System.out.println("greaterThanTen: " + greaterThanTen.test(num));
System.out.println("divisibleByThree: " + divisibleByThree.test(num));
System.out.println("divisibleByFive: " + divisibleByFive.test(num));
System.out.println("lessThanOrEqualToTen: " + lessThanOrEqualToTen.test(num));
System.out.println("divisibleByThreeAndFive: " + divisibleByThreeAndFive.test(num));
System.out.println("divisibleByThreeOrFive: " + divisibleByThreeOrFive.test(num));
System.out.println("equalsToTen: " + equalToTen.test(num));
```

Output

```
Number: 10
greaterThanTen: false
divisibleByThree: false
divisibleByFive: true
lessThanOrEqualToTen: true
divisibleByThreeAndFive: false
divisibleByThreeOrFive: true
equalsToTen: false
```


Functional Interfaces (FI)

Function<T, R> Interface

Specialized versions (non-generics)

- **Function<T, R>** specializations:

IntFunction<R>

LongFunction<R>

DoubleFunction<R>

  – Take an argument of int, long or double and return a value of type R

ToIntFunction<T>

ToLongFunction<T>

ToDoubleFunction<T>

  – Take argument of type T and return an int, long or double

- Similar specializations exist for other types of generic functions in the table

Functional Interfaces (FI)

Libraries and APIs

- FI are extensively used in Java Library for Collection and Stream API
- Benefit: the code is more concise and more readable
- When a method in the API takes a FI as an argument, the user of the API should use a λ *ex* to pass the argument
- Example of API design and use

Functional Interfaces (FI)

Libraries and APIs - Example

```
// Gender.java [source Kishori]
public enum Gender { MALE, FEMALE }

// Person.java
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class Person {
  private String firstName;
  private String lastName;
  private LocalDate dob;
  private Gender gender;

  public Person(String firstName,
         String lastName, LocalDate dob,
         Gender gender) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.dob = dob;
    this.gender = gender;
  }
```

```
public String getFirstName() { return firstName; }
public void setFirstName(String firstName) {
  this.firstName = firstName;
}
public String getLastName() { return lastName; }
public void setLastName(String lastName) {
  this.lastName = lastName;
}
public LocalDate getDob() { return dob; }
public void setDob(LocalDate dob) { this.dob = dob; }
public Gender getGender() { return gender; }
public void setGender(Gender gender) {
  this.gender = gender;
}
// cont. on next slide
```

Functional Interfaces (FI)

Libraries and APIs - Example

```
@Override
public String toString() {
  return firstName + " " + lastName + ", " + gender + ", " + dob;
}
// A utility method
public static List<Person> getPersons() {
   ArrayList<Person> list = new ArrayList<>();
   list.add(new Person("Ion", "Ionescu", LocalDate.of(1975, 1, 20), MALE));
   list.add(new Person("Vasile", "Vasilescu", LocalDate.of(1965, 9, 12), MALE));
   list.add(new Person("Ana", "Ionescu", LocalDate.of(1970, 9, 12), FEMALE));
   return list;
}
} // end of class Person
```


Functional Interfaces (FI)

Libraries and APIs - Example

```
// FunctionUtil.java
// Defines methods that apply a function on java.util.List
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
public class FunctionUtil {
  // Applies an action on each item in a list
  public static <T> void forEach (List<T> list,
                          Consumer<? super T> action) {
    for(T item : list) { action.accept(item); }
  }
  // Applies a filter to a list;
  // returns the filtered list items
  public static <T> List<T> filter(List<T> list,
                  Predicate<? super T> predicate) {
    List<T> filteredList = new ArrayList<>();
    for(T item : list) {
      if (predicate.test(item)) { filteredList.add(item); }
    }
    return filteredList;
  }
  // Maps each item in a list to a value
  public static <T, R> List<R> map(List<T> list,
                      Function<? super T, R> mapper) {
    List<R> mappedList = new ArrayList<>();
    for(T item : list) {
         mappedList.add(mapper.apply(item));
    }
    return mappedList;
  }
} // end of class FunctionUtil
```

Functional Interfaces (FI)

Libraries and APIs

- Suppose FunctionUtil class is part of a library and its FI will be used as target types of *λ ex-s*
- The following program performs the following actions using lambda expressions
  - gets a list of persons,
  - applies a filter to the list to get a list of only males,
  - maps persons to the year of their birth, and
  - adds one year to each male's date of birth.
- Note the conciseness of the code - one line of code to perform each action

Functional Interfaces (FI)

Example

```
// FunctionUtilTest.java
import java.util.List;
public class FunctionUtilTest {
  public static void main(String[] args) {
    List<Person> list = Person.getPersons();
    // Use the forEach() method to print each person in the list
    System.out.println("Original list of persons:");
    FunctionUtil.forEach(list, p -> System.out.println(p));
    // Filter only males
    List<Person> maleList = FunctionUtil.filter(list, p -> p.getGender() == MALE);
    System.out.println("\nMales only:");
    FunctionUtil.forEach(maleList, p -> System.out.println(p));
    // Map each person to his/her year of birth
    List<Integer> dobYearList = FunctionUtil.map(list, p -> p.getDob().getYear());
    System.out.println("\nPersons mapped to year of their birth:");
    FunctionUtil.forEach(dobYearList, year -> System.out.println(year));
    // Apply an action to each person in the list  - add one year to each male's dob
    FunctionUtil.forEach(maleList, p -> p.setDob(p.getDob().plusYears(1)));
    System.out.println("\nMales only after ading 1 year to DOB:");
    FunctionUtil.forEach(maleList, p -> System.out.println(p));
  }
}
```

Output

Original list of persons:
Ion Ionescu, MALE, 1975-01-20
Vasile Vasilescu, MALE, 1965-09-12
Ana Ionescu, FEMALE, 1970-09-12
Males only:
Ion Ionescu, MALE, 1975-01-20
Vasile Vasilescu, MALE, 1965-09-12
Persons mapped to year of their birth:
1975 1965 1970
Males only after ading 1 year to DOB:
Ion Ionescu, MALE, 1976-01-20
Vasile Vasilescu, MALE, 1966-09-12

forEach

-takes a Consumer function
-Each item is passed to this Consumer function.
-The function can take any action on the item (in this case a Consumer that prints the item on the standard output as shown:
FunctionUtil.forEach(list,
p -> System.out.println(p));
Typically, a Consumer applies an action on the item it receives to produce side effects (print the item).

Method References (MR)

- MR - shorthand to create ***λ exs*** using existing methods
  - can only be used where a ***λ ex*** can be used
- MR is not
  - a new type in Java;
  - a pointer to functions (as in other languages);
- Syntax

<Qualifier>::<MethodName>
  - <Qualifier> depends on the type of the method reference
  - <MethodName> is the name of the method
- MR does not call the method when it is declared (the method is called later, when the method of its target type is called)

- **Example - *λ ex*** used to define a anonymous function (takes String argument and returns its length)

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction =
                              str -> str.length();
String name = "Popescu";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ",
                              length = " + len);
```

- Example – same functionality using MR to the method **length** of class **String**

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction =
                                   String::length;
String name = "Popescu";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ",
                                 length = " + len);
```


Method References (MR)

- **In a MR one cannot specify parameter type and return type (this may generate confusion)**
- Since MR is a shorthand of a ***λ ex*** the target type (i.e. a FI) determines the details
- If the method is overloaded, compiler will choose the most specific method based on the context

Types of Method References

Syntax	Description
`TypeName::staticMethod`	A method reference to a static method of a class, an interface, or an enum
`objectRef::instanceMethod`	A method reference to an instance method of the specified object
`ClassName::instanceMethod`	A method reference to an instance method of an arbitrary object of the specified class
`TypeName.super::instanceMethod`	A method reference to an instance method of the supertype of a particular object
`ClassName::new`	A constructor reference to the constructor of the specified class
`ArrayTypeName::new`	An array constructor reference to the constructor of the specified array type

Method References (MR)

Static MR

- Class **Integer,** method
  static String toBinaryString(int i)
  - Takes an int argument (unsigned int in base 2) and returns its String representation

- **Using a lambda expression**
  Function<Integer, String> func1 = x -> Integer.toBinaryString(x);
  System.out.println(func1.apply(17));
- For ***λ ex*** compiler infers the type of x as Integer and the return type as String, by using the target type

- **Using a static MR**
  Function<Integer, String> func2 = Integer::toBinaryString;
  System.out.println(func2.apply(17));
- The compiler finds on the right-hand side of = operator a static method reference to **the toBinaryString()** method of the **Integer** class which takes an int as an argument and returns a String
- The target type of the method reference is a function: takes an Integer argument and returns a String
- The compiler verifies that the method reference and target type are assignment compatible

Method References (MR)

Static MR

- Class **Integer,** method

static int sum(int a, int b)
  - Takes two int arguments and returns their sum as int

Function<Integer, Integer> func2 = Integer::sum;
// compile-time error - due to mismatch in number of arguments (target type Function takes only one argument)

- Fix the error

// 1. Use a lambda expression
BiFunction<Integer, Integer, Integer> func1 =
(x, y) -> Integer.sum(x, y);
System.out.println(func1.apply(17, 15));

// 2. Use a method reference
BiFunction<Integer, Integer, Integer> func2 =
Integer::sum;
System.out.println(func2.apply(17, 15));

- Class **Person,** method **getPersons()**

static List<Person> getPersons()

- Example

Supplier<List<Person>> supplier =
Person::getPersons;
List<Person> personList = supplier.get();
FunctionUtil.forEach(personList,
p -> System.out.println(p));

Method References (MR)

Static MR – Overloaded static methods

- Class **Integer,** method **valueOf**

```
static Integer valueOf(int i)
static Integer valueOf(String s)
static Integer valueOf(String s, int radix)

// Uses Integer.valueOf(int)
Function<Integer, Integer> func1 = Integer::valueOf;

// Uses Integer.valueOf(String)
Function<String, Integer> func2 = Integer::valueOf;

// Uses Integer.valueOf(String, int)
BiFunction<String, Integer, Integer> func3 = Integer::valueOf;

System.out.println(func1.apply(17));
System.out.println(func2.apply("17"));
System.out.println(func3.apply("10001", 2));
```


Method References (MR)

Instance MR

- Invoked on object's references
- The object reference on which the instance method is invoked is known as the **receiver of method invocation**
- Receiver of method invocation can be:
  - Object reference
  - Expression that evaluates to object reference
- Examples of receivers

String name = “Popescu";
int len1 = **name**.length(); // name is the receiver of the length() method
int len2 = **"Hello"**.length(); // "Hello" is the receiver of the length() method
int len3 = **(new String(«Popescu"))**.length(); // (new String(“Popescu")) is the receiver of length() method

- **Bound receiver**
  - In a MR for an instance method, you can specify the receiver of the method invocation **explicitly**

  objectRef::instanceMethod
- **Unbound receiver**
  - In a MR for an instance method, you can specify the receiver of the method invocation **implicitly when the method is invoked**

  ClassName::instanceMethod

Method References (MR)

Instance MR – Bound Receiver

Syntax objectRef::instanceMethod

Example 1

- As Lambda Expression

```
Supplier<Integer> supplier = () -> “Popescu".length();
System.out.println(supplier.get());
```

- As MR - Re-write using Instance MR (The object “Popescu” is the bound receiver, Supplier<Integer> is the target type

```
Supplier<Integer> supplier = “Popescu"::length;
System.out.println(supplier.get());
```

Example 2

- As Lambda Expression

```
Consumer<String> consumer = str -> System.out.println(str);
consumer.accept("Hello");
```

- As MR with System.out as bound receiver

```
Consumer<String> consumer = System.out::println;
consumer.accept("Hello");
```

Example 2 (comment)

- When the MR System.out::println is used, the checks target type, which is Consumer<String> that represents a function type that takes a String as an argument and returns void
- The compiler finds a println(String) method in the PrintStream class of the System.out object and uses that method for the method reference

Example 3

```
List<Person> list = Person.getPersons();
FunctionUtil.forEach(list,
                     System.out::println);
```


Method References (MR)

Instance MR – Unbound Receiver

Syntax ClassName::instanceMethod

Example 1

```
Function<Person, String> fNameFunc = (Person p) -> p.getFirstName();
```

- Use Instance MR

```
Function<Person, String> fNameFunc = Person::getFirstName;
```

Two confusions

- The syntax is the same as the syntax for a method reference to a static method;
  - Clarify: Look at the method name and check whether it is a static or instance
- Which object is the receiver of the instance method invocation?
  - Clarify using the rule: the first argument to the function represented by the target type is the receiver of the method invocation.

Example 2

- Consider an instance method reference called String::length that uses an unbound receiver. The receiver is supplied as the first argument to the apply() method, as shown:

```
Function<String, Integer> strLengthFunc = String::length;
String name =“Popescu"; // name is the receiver of String::length
int len = strLengthFunc.apply(name);
System.out.println("name = " + name + ", length = " + len);
```

Method References (MR)

Instance MR – Unbound Receiver

Example 3

- The instance method concat() of the String class has the following declaration:

String concat(String str)

- MR String::concat is an instance MR for a target type whose function takes two String arguments and returns a String.
- The first argument will be the receiver of the concat() method and the second argument will be passed to the concat() method

```
String greeting = "Hello";
String name = "Viorel";
// Uses a lambda expression
BiFunction<String, String, String> func1 =
                                (s1, s2) -> s1.concat(s2);
System.out.println(func1.apply(greeting, name));
// Uses an instance MR on an unbound receiver
BiFunction<String, String, String> func2 =
                                String::concat;
System.out.println(func2.apply(greeting, name));
```

Example 4

- Using the Instance MR Person::getFirstName, an Instance MR to an unbound receiver

```
List<Person> personList = Person.getPersons();
// Maps each Person object to its first name
List<String> firstNameList =
                FunctionUtil.map(personList,
Person::getFirstName);
// Prints the first name list
FunctionUtil.forEach(firstNameList,
                                System.out::println);
```


Method References (MR)

Supertype Instance MR

- The keyword **super** is used as a qualifier to invoke the overridden method in a class or an interface.
- The keyword is available only in an instance context
- Use the following syntax to construct a method reference that refers to the instance method in the supertype and the method that's invoked on the current instance:

TypeName.super::instanceMethod

- Example (next slide)

Example

```
// Priced.java
public interface Priced {
  default double getPrice() { return 1.0; }
}
// Item.java
import java.util.function.Supplier;
public class Item implements Priced {
  private String name = "Unknown";
  private double price = 0.0;
  public Item() {
   System.out.println("Constructor
          Item() called.");
  }
  public Item(String name) {
    this.name = name;
    System.out.println("Constructor
          Item(String) called.");
  }
  public Item(String name, double price) {
  this.name = name;
  this.price = price;
  System.out.println("Constructor
         Item(String, double) called.");
  }
public String getName() { return name; }
public void setName(String name) { this.name = name;}
public void setPrice(double price) { this.price = price;}
@Override
public double getPrice() { return price; }
@Override
public String toString() {
  return "name = " + getName() + ", price = " + getPrice();
}
public void test() {
   // Uses the Item.toString() method
   Supplier<String> s1 = this::toString;
   // Uses Object.toString() method
   Supplier<String> s2 = Item.super::toString;
   // Uses Item.getPrice() method
   Supplier<Double> s3 = this::getPrice;
   // Uses Priced.getPrice() method
   Supplier<Double> s4 = Priced.super::getPrice;
   // Uses all method references and prints the results
   System.out.println("this::toString: " + s1.get());
   System.out.println("Item.super::toString: " + s2.get());
   System.out.println("this::getPrice: " + s3.get());
   System.out.println("Priced.super::getPrice: " + s4.get());
 }
} // End of class Item
```


Method References (MR)

Supertype Instance MR

- The test() method in the Item class uses four method references with a bound receiver
- The receiver is the Item object on which the test() method is called
  - The method reference **this::toString** refers to the toString() method of the Item class.
  - The method reference **Item.super::toString** refers to the toString() method of the Object class, which is the superclass of the Item class.
  - The method reference **this::getPrice** refers to the getPrice() method of the Item class.
  - The method reference **Priced.super::getPrice** refers to the getPrice() method of the Priced interface, which is the superinterface of the Item class.

- Testing the Item

```
// ItemTest.java
public class ItemTest {
  public static void main(String[] args) {
    Item apple = new Item("Apple", 0.75);
    apple.test();
  }
}
```

Output:

```
Constructor Item(String, double) called.
this::toString: name = Apple, price = 0.75
Item.super::toString: com.jdojo.lambda.Item@24d46ca6
this::getPrice: 0.75
Priced.super::getPrice: 1.0
```

Method References (MR)

Constructor References

- Body of a lambda expression may be an object creation expression

```
Supplier<String> func1 = () -> new String();
Function<String,String> func2 = str -> new String(str);
```

- Syntax of constructor references
- ClassName must be instantiable (not abstract class for example)

```
ClassName::new
ArrayTypeName::new
```

- A class may have multiple constructors
- The compiler selects a specific constructor based on the context (target type and the number of arguments in the abstract method of the target type)

Example

```
Supplier<Item> func1 = () -> new Item();
Function<String,Item> func2 = name -> new Item(name);
BiFunction<String,Double, Item> func3 =
            (name, price) -> new Item(name, price);
System.out.println(func1.get());
System.out.println(func2.apply("Apple"));
System.out.println(func3.apply("Apple", 0.75));
// A compile-time error, no Item class constructor to match,
// i.e. to take a Double argument
Function<Double,Item> func4 = Item::new;
```

Output

```
Constructor Item() called.
name = Unknown, price = 0.0
Constructor Item(String) called.
name = Apple, price = 0.0
Constructor Item(String, double) called.
name = Apple, price = 0.75
```


Method References (MR)

Constructor References

- Arrays – no constructors in Java
- Array constructors are treated to have one argument of int type that is the size of the array

```
// Uses a lambda expression
IntFunction<int[]> arrayCreator1 = size -> new int[size];
int[] empIds1 = arrayCreator1.apply(5); // Creates an int array of five elements
// Uses an array constructor reference
IntFunction<int[]> arrayCreator2 = int[]::new;
int[] empIds2 = arrayCreator2.apply(5); // Creates an int array of five elements
```

- Using a Function<Integer,R> type to use an array constructor reference, where R is the array type.

```
// Uses an array constructor reference
Function<Integer,int[]> arrayCreator3 = int[]::new;
int[] empIds3 = arrayCreator3.apply(5); // Creates an int array of five elements
```

- Creating a two-dimensional int array with the first dimension having the length of 5:

```
// Uses an array constructor reference
IntFunction<int[][]> TwoDimArrayCreator = int[][]::new;
int[][] matrix = TwoDimArrayCreator.apply(5); // Creates an int[5][] array
```

Method References (MR)

Generic Method References

- The compiler determines the actual type for generic type parameters when a method reference refers to a generic method
- Consider the following generic method in the Arrays class (java.util package):

static <T> List<T> asList(T... a)

- You can use **Arrays::asList** as the MR for this method
- If you are passing String objects to the **asList()** method, its method reference can be written as **Arrays::<String>asList**

- In the following code, Arrays::asList will work the same as the compiler will infer String as the type parameter for the asList() method by examining the target type

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
...
Function<String[], List<String>>asList = Arrays::<String>asList;
String[] namesArray = {"Ion", "Vasile", "Sandu"};
List<String> namesList = asList.apply(namesArray);
for(String name : namesList) { System.out.println(name); }
```


Method References (MR)

Recursive Lambda Expressions

- A lambda expression does not support recursive invocations.
- If you need a recursive function, you need to use a MR or an anonymous inner class.

Method References (MR)
Comparing Objects

- The Comparator interface is a FI

```
package java.util;
@FunctionalInterface
public interface Comparator<T> {
int compare(T o1, T o2);
// Other methods (not shown)
}
```

- Comparator interface contains many default and static methods that can be used along with lambda expressions to create Comparator instances
- It is recommended to explore API documentation for the interface
- Two important methods of the Comparator interface

static <T,U extends Comparable<? super U>> Comparator<T>
comparing (Function<? super T,? extends U> keyExtractor)
default <U extends Comparable<? Super U>>Comparator<T>
thenComparing (Function<? super T,? extends U> keyExtractor)

Method References (MR)
Comparing Objects

- **comparing**
  - takes a Function and returns a Comparator
  - the Function should return a Comparable that is used to compare two objects
- You can create a Comparator object to compare Person objects based on their first name, as shown

Comparator<Person> firstNameComp = Comparator.comparing(Person::getFirstName);

- **thenComparing**
  - It is used to specify a secondary comparison if two objects are the same in sorting order based on the primary comparison.
- The following statement creates a Comparator<Person> that sorts Person objects based on their last names, first names, and DOBs

```
Comparator<Person> lastFirstDobComp =
    Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName)
        .thenComparing(Person::getDob);
```

Method References (MR)

Comparing Objects

- See (next slides) how to use the method references to create a Comparator objects to sort Person objects
- The program uses the sort() default method of the List interface to sort the list of persons
  - sort() method takes a Comparator as an argument.
  - ort is facilitated by lambda expressions and default methods in interfaces

Method References (MR)

Comparing Objects

```
// ComparingObjects.java
import java.util.Comparator;
import java.util.List;
public class ComparingObjects {
  public static void main(String[] args) {
    List<Person> persons = Person.getPersons();
    // Sort using the first name
    persons.sort(Comparator.comparing(Person::getFirstName));
    // Print the sorted list
    System.out.println("Sorted by the first name:");
    FunctionUtil.forEach(persons, System.out::println);
    // Sort using the last name, first name, and then DOB
    persons.sort(Comparator.comparing(Person::getLastName)
      .thenComparing(Person::getFirstName)
      .thenComparing(Person::getDob));
    // Print the sorted list
    System.out.println("\nSorted by the last name, first name, and dob:");
    FunctionUtil.forEach(persons, System.out::println);
  }
}
```