

lambda expression

sf segmentfault.com/a/1190000017152245

博弈



One problem with anonymous classes is that if the implementation of the anonymous class is very simple, such as an interface containing only one method, then the syntax of the anonymous class may seem unwieldy and unclear. In these cases, you will usually try to pass the function as a parameter. Passed to another method, such as an action that should be taken when someone clicks a button, Lambda expressions allow you to perform this action, treating functions as method parameters, or treating code as data.

The previous section on [anonymous classes](#) showed you how to implement a base class without giving it a name. While this is generally more concise than naming a class, even anonymous classes can seem a bit excessive and cumbersome for a class with only one method. , Lambda expressions allow you to express single-method class instances more compactly.

Ideal use cases for lambda expressions

Let's say you are creating a social networking application and you want to create a feature that enables administrators to perform any kind of action on members of the social networking application who meet certain criteria, such as sending messages. This use case is detailed in the following table:

Field	describe
name	Perform actions on selected members
main character	administrator
Prerequisites	Administrator has logged into the system

Field	describe
Postcondition	Only perform operations on members who meet specified criteria
Main success stories	<ol style="list-style-type: none"> 1. The administrator specifies the conditions for members to perform specific operations 2. The administrator specifies the operations to be performed on these selected members 3. The administrator selects the Submit button 4. The system searches for all members who meet the specified conditions 5. The system matches all Members perform specified actions
Expand	Admins can choose to preview members who meet specified criteria before specifying the action to be performed or before selecting the Submit button
occurrence frequency	many times a day

Assume that the members of this social networking application are represented by the following Person class:

```
public class Person {

    public enum Sex {
        MALE , FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge () {
        // ...
    }

    public void printPerson () {
        // ...
    }
}
```

Assume that your social networking application's members are stored in `List<Person>` instances.

This section first introduces a simple approach to this use case, which improves this approach using partial and anonymous classes, and then it is done in an efficient and concise way using lambda expressions. The code excerpt described in this section is found in the example `RosterTest`.

Method 1: Create a method that searches for members matching a characteristic

A simple approach is to create several methods, each of which searches for members matching one characteristic, such as gender or age. The following method prints members over a specified age:

```
public static void printPersonsOlderThan(List<Person> roster , int age ) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Note: List is an ordered collection. A collection is an object that combines multiple elements into a unit. Collections are used to store, retrieve, operate, and deliver aggregated data. For more information about collections, see [Collection Path](#).

This approach can make your application brittle due to the possibility that updates introduced (e.g. newer data types) cause the application to stop working. Suppose you upgrade your application and change `Person` the structure of the class so that it contains different member variables, maybe the class records and measures age using a different data type or algorithm, you have to rewrite a lot of the API to accommodate this change, furthermore, this approach is Unnecessary restrictions; for example, what if you want to print members younger than a certain age?

Method 2: Create more generalized search methods

The following method is `printPersonsOlderThan` more general; it prints members within a specified age range:

```
public static void printPersonsWithinAgeRange (
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

What if you want to print members of a specific gender, or a combination of gender and age range? `Person` What if you decide to change the class and add other properties like relationship status or location? While this approach is `printPersonsOlderThan` more general, trying to create a separate method for each possible search query still results in brittle code. You can instead separate the code that specifies the criteria to be searched in other classes.

Method 3: Specify the search condition code in the local class

The following method prints members that match the search criteria you specify:

```
public static void printPersons (
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p .printPerson ();
        }
    }
}
```

This method `tester.test` checks whether `List` each instance contained in the argument list satisfies the search criteria specified in the argument by calling a method and calling the method on the instance if the method returns a value

`.PersonCheckPerson tester tester .test true Person printPersons`

To specify search criteria, implement `CheckPerson` the interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

The following class `test` implements `CheckPerson` the interface by specifying an implementation of a method that filters members eligible for US Selective Service: If `Person` the argument is male and the age is between 18 and 25, the return `true` value is:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25 ;
    }
}
```

To use this class, you need to create a new instance of it and call `printPersons` methods:

```
printPersons(
    roster, new CheckPersonEligibleForSelectiveService()) ;
```

While this approach is less brittle— `Person` you don't have to override the method if the structure changes—you still need extra code: a new interface and a partial class for every search you plan to perform in your application, because `CheckPersonEligibleForSelectiveService` an interface is implemented, So you can use anonymous classes instead of local classes and don't need to declare a new class for each search.

Method 4: Specify search condition code in an anonymous class

One argument to the method called below `printPersons` is an anonymous class that filters members eligible for U.S. Selective Service: those who are male and between the ages of 18 and 25:

```
printPersons(  
    roster ,  
    new CheckPerson() {  
        public boolean test( Person p) {  
            return p.getGender() == Person.Sex.MALE  
                && p.getAge() >= 18  
                && p.getAge() <= 25 ;  
        }  
    }  
);
```

This approach reduces the amount of code required because you don't have to create a new class for each search you want to perform, however, considering that the `CheckPerson` interface only contains a single method, the syntax of an anonymous class is clunky, in which case you can use lambda expressions instead of anonymous classes, as explained in the next section.

Method 5: Use Lambda expressions to specify search condition codes

The `CheckPerson` interface is a functional interface. A functional interface is any interface that contains only one abstract method (a functional interface may contain one or more default methods or static methods). Since a functional interface contains only one abstract method, the implementation The method name can be omitted. To do this, instead of using an anonymous class expression, a lambda expression is used, which is shown in the following method call:

```
printPersons(  
    roster ,  
    ( Person p) -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18  
        && p.getAge() <= 25  
);
```

For information about how to define lambda expressions, see [Syntax for Lambda Expressions](#).

You can use standard functional interfaces instead of `CheckPerson` interfaces, further reducing the amount of code required.

Method 6: Use standard functional interfaces with Lambda expressions

Rethink `CheckPerson` the interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

This is a very simple interface, it is a functional interface because it contains only one abstract method, this method accepts one parameter and returns a boolean value, the method is very simple, it may not be necessary to define a method in your application. Worthy, therefore, the JDK defines several standard functional interfaces, `java.util.function` which you can find in packages.

For example, you could use `Predicate<T>` an interface instead `CheckPerson`, which contains methods `boolean test(T t)`:

```
interface Predicate < T > {
    boolean test(T t);
}
```

An interface `Predicate<T>` is an example of a generic interface (for more information about generics, see the Generics (Updated) course), a generic type (such as a generic interface) `<T>` specifies one or more type parameters within angle brackets (), which The interface contains only one type parameter `T`. When you declare or instantiate a generic type with actual type parameters, you have a parameterized type. For example, a parameterized type is `Predicate<Person>` as follows:

```
interface Predicate < Person > {
    boolean test(Person t);
}
```

This parameterized type contains a method that has `CheckPerson.boolean test(Person p)` the same return type and parameters, so you can use `Predicate<T>` instead `CheckPerson`, as shown in the following method:

```
public static void printPersonsWithPredicate (
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson ();
        }
    }
}
```

Therefore, the following method call is the same as when called in Method 3: Specify search criteria code in a local class to obtain members eligible for selective service `printperson`:

```

printPersonsWithPredicate(
    roster ,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
) ;

```

This is not the only possible place to use lambda expressions in this method, the following methods suggest other ways to use lambda expressions.

Method 7: Use Lambda expressions throughout your application

Reconsider `printPersonsWithPredicate` the method to see other places where lambda expressions can be used:

```

public static void printPersonsWithPredicate (
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p .printPerson ();
        }
    }
}

```

This method checks whether each instance contained in `List` the parameter satisfies the condition specified in the parameter , and if the instance satisfies the specified condition, calls the method on the instance

`.rosterPersonPredicatetesterPersontesterPersonprintPersron`

Instead of calling a method, you can specify a different action to be performed on those instances that meet `tester` specified conditions . You can specify this action using a lambda expression. Suppose you want a lambda expression similar to , which takes an argument (an object of type) and returns it . Remember that to use lambda expressions, you need to implement a functional interface. In this case, you need a functional interface containing abstract methods that can accept a parameter of type and return it . An interface contains methods, which have these properties. The following method replaces calls with the instance of the calling method

`:PersonprintPersonprintPersonPersonvoidPersonvoidConsumer<T>void accept(T t)p.printPerson()acceptConsumer<Person>`

```

public static void processPersons (
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block .accept (p);
        }
    }
}

```

printPersons Therefore, the following method call is the same as when called in Method 3: Specify search criteria code in a local class to obtain members eligible for selective service. The lambda expression used to print the members is as follows:

```

processPersons(
    roster ,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25 ,
    p -> p.printPerson()
) ;

```

What if you want to do more with a member's profile instead of printing it? Suppose you want to verify a member's profile or retrieve their contact information? In this case, you need a functional interface that contains abstract methods that return values. The **Function<T,R>** interface contains methods **R apply(T t)** that retrieve **mapper** the data specified by the parameters and then **block** perform the operations specified by the parameters:

```

public static void processPersonsWithFunction (
    List<Person> roster,
    Predicate<Person> tester,
    Function <Person, String > mapper,
    Consumer< String > block ) {
    for ( Person p : roster) {
        if (tester. test (p)) {
            String data = mapper. apply (p);
            block.accept (data) ;
        }
    }
}

```

roster The following method retrieves the email address from each member included in the list that is eligible for selective service and then prints it out:

```

processPersonsWithFunction(
    roster ,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25 ,
    p -> p.getEmailAddress(),
    email -> System.out.println( email )
) ;

```

Method 8: Use generics more extensively

Reconsidering the method `processPersonsWithFunction`, here is the generic version of it, which accepts a collection containing elements of any data type as a parameter:

```

public static < X , Y > void processElements(
    Iterable< X > source,
    Predicate< X > tester,
    Function < X , Y > mapper,
    Consumer< Y > block) {
    for ( X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}

```

To print the email addresses of members eligible for selective service, call the method as follows `processElements`:

```

processElements(
    roster ,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25 ,
    p -> p.getEmailAddress(),
    email -> System.out.println( email )
) ;

```

This method call performs the following operations:

1. Gets the source of the object from the collection `source`, in this example it `roster` gets `Person` the source of the object from the collection, note that a collection `roster` is `List` a collection of types and also `Iterable` an object of type.
2. Filters objects that match `Predicate` an object `tester`, which in this example `Predicate` is a lambda expression that specifies which members are eligible for selective service.

3. Maps each filter object to the value specified by **Function**the object **mapper**, which in this example **Function**is a lambda expression that returns the member's email address.
4. Performs an operation on each map object specified by **Consumer**the object **block**, which in this example **Consumer**is a lambda expression that prints the string that is the **Function**email address returned by the object.

You can replace each operation with an aggregate operation.

Method 9: Use an aggregate operation that accepts a Lambda expression as a parameter

roster The following example uses an aggregation operation to print the email addresses of members included in a collection that is eligible for selective service :

```
roster
    .stream ()
    .filter(
        p -> p.get Gender() == Person.Sex.MALE
            && p.get Age() >= 18
            && p.get Age() <= 25 )
    .map(p -> p.get EmailAddress() )
    . for Each( email -> System. out . println ( email ) );
```

The following table **processElements**maps each operation performed by a method to the corresponding aggregation operation:

processElements action	Aggregation operation
Get the source of the object	Stream<E> stream()
Filter Predicate objects matching objects	Stream<T> filter(Predicate<? super T> predicate)
Maps an object to another value specified by a Function object	<R> Stream<R> map(Function<? super T,? extends R> mapper)
Perform Consumer the operation specified by the object	void forEach(Consumer<? super T> action)

The operations **filter**, **map**and are aggregation operations, which process elements from a stream rather than directly from a collection (which is why **forEach**the first method called in this example is). **stream**A stream is a sequence of elements, unlike a collection, it is not a data structure that stores elements, instead a stream carries values from a source (such as a collection) through a pipe, which is a sequence of streaming operations, in this case, and in addition, an aggregation **filter-map-foreach**operation Usually accept lambda expressions as parameters, allowing you to customize how they behave.

For a more complete discussion of aggregation operations, see the Aggregation Operations course.

Lambda expressions in GUI applications

To handle events in a graphical user interface (GUI) application, such as keyboard operations, mouse operations, and scrolling operations, event handlers are typically created, which usually involves implementing a specific interface. Typically, event handler interfaces are functional interfaces; They often only have one approach.

In the JavaFX example `HelloWorld.java` (discussed in the previous section on anonymous classes), you can replace the anonymous class with a lambda expression in this statement:

```
btn.setOnAction( new EventHandler<ActionEvent>() {  
  
    @Override  
    public void handle ( ActionEvent event ) {  
        System.out.println( "Hello World!" );  
    }  
});
```

The method call specifies what happens when the button represented by the object `btn` is selected, this method requires an object of type, the interface contains only one method, this interface is a functional interface, so you can replace it with the lambda expression shown below

```
:btn.setOnAction( new EventHandler<ActionEvent>() {  
    public void handle(T event)
```

```
});  
btn.setOnAction ( event -> System.out.println( "Hello World!" )  
);
```

Lambda expression syntax

The lambda expression contains the following:

- A comma-separated list of formal parameters in parentheses. `CheckPerson.test` The method contains one parameter `p`, which represents `Person` an instance of the class. Note: You can omit the data type of the parameters in the lambda expression. In addition, if there is only one parameter, you can omit the parentheses. For example, the following lambda expression is also valid:

```
p -> p.getGender () == Person.Sex.MALE  
    && p.getAge () >= 18  
    && p.getAge () <= 25
```

- arrow mark, `->`

- The body of code, consisting of a single expression or block of statements, this example uses the following expression:

```
p .getGender () == Person .Sex .MALE
  && p .getAge () >= 18
  && p .getAge () <= 25
```

If you specify a single expression, the Java runtime will evaluate the expression and return its value. Alternatively, you can use `return` the statement:

```
p -> {
    return p.get Gender() == Person.Sex.MALE
        && p.get Age() >= 18
        && p.get Age() <= 25 ;
}
```

`return` Statements are not expressions. In a lambda expression, you must enclose the statement in curly braces (`{ }`). However, you do not have to enclose the method call in curly braces `void`. For example, the following is a valid lambda expression:

```
email -> System.out.println( email )
```

Note that lambda expressions look a lot like method declarations, and you can think of lambda expressions as anonymous methods—methods without names.

The following example Calculator is an example of a lambda expression that takes multiple formal parameters:

```
public class Calculator {

    interface IntegerMath {
        int operation ( int a, int b ) ;
    }

    public int operateBinary ( int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main (String... args) {

        Calculator myApp = new Calculator ();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println( "40 + 2 = " +
            myApp.operateBinary( 40 , 2 , addition));
        System.out.println( "20 - 10 = " +
            myApp.operateBinary( 20 , 10 , subtraction));
    }
}
```

The method `operateBinary` performs a mathematical operation on two integer operands. The operation itself `IntegerMath` is specified by the instance. This example uses lambda expressions, `addition` and `subtraction` defines two operations. The example prints the following:

```
40 + 2 = 42
20 - 10 = 10
```

Access local variables in an enclosing scope

Like local and anonymous classes, lambda expressions can capture variables , which have the same access to local variables of the enclosing scope, however, unlike local and anonymous classes, lambda expressions do not have any shadowing issues (for more information, please See Masking), Lambda expressions have lexical scope. This means that they do not inherit any names from the supertype or introduce new levels of scope. Declarations within lambda expressions are interpreted the same as declarations in the enclosing environment. The following example `LambdaScopeTest` demonstrates this :

```

import java.util.function.Consumer;

public class LambdaScopeTest {

    public int x = 0 ;

    class FirstLevel {

        public int x = 1 ;

        void methodInFirstLevel ( int x ) {

            // The following statement causes the compiler to generate
            // the error "local variables referenced from a lambda expression
            // must be final or effectively final" in statement A:
            //
            // x = 99;

            Consumer<Integer> myConsumer = (y) ->
            {
                System.out.println( "x = " + x); // Statement A
                System.out.println( "y = " + y);
                System.out.println( "this.x = " + this.x);
                System.out.println( "LambdaScopeTest.this.x = " +
                    LambdaScopeTest.this.x);
            };

            myConsumer.accept(x);

        }

    }

    public static void main ( String... args ) {
        LambdaScopeTest st = new LambdaScopeTest();
        LambdaScopeTest.FirstLevel fl = st.new FirstLevel () ;
        fl.methodInFirstLevel( 23 );
    }
}

```

This example produces the following output:

```

x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0

```

If you substitute **myConsumer** parameters in the declaration of a lambda expression , the compiler generates an error:**xy**

```

Consumer<Integer> myConsumer = (x) -> {
    // ...
}

```

The compiler generates the error "variable x is already defined in method methodInFirstLevel(int)" because the lambda expression does not introduce a new scope level, therefore, you can directly access the fields, methods and local variables of the enclosing scope. For example, a lambda expression directly accesses `methodInFirstLevel` the parameters of a method `x`. To access a variable in an enclosing class, use the keyword `this`, in this example, `this.x` to reference a member variable `FirstLevel.x`.

However, like local and anonymous classes, lambda expressions can only access `final` or valid `final` the local variables and parameters of the enclosing block. For example, suppose you `methodInFirstLevel` add the following assignment statement immediately after the definition statement:

```
void method InFirstLevel( int x ) {  
    x = 99 ;  
    // ...  
}
```

Due to this assignment statement, the variable `FirstLevel.x` is no longer `final`, therefore, the Java compiler generates an error message similar to "local variables referenced from a lambda expression must be final or effectively final" where the lambda expression `myConsumer` tries to access `FirstLevel.x` the variable:

```
System.out.println ("x = " + x);
```

target type

How do you determine the type of a lambda expression? Recall the lambda expression that selects men and members between the ages of 18 and 25:

```
p -> p .getGender () == Person .Sex .MALE  
    && p .getAge () >= 18  
    && p .getAge () <= 25
```

This lambda expression is used in the following two methods:

- Method 3: Specify the search conditions in the local class in the code `public static void printPersons(List<Person> roster, CheckPerson tester)`
- Method 6: Using standard functional interfaces with Lambda expressions `public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)`

When Java runtime calls a method `printPersons`, it expects `CheckPerson` a data type of , so the lambda expression belongs to this type. However, when Java runtime calls a method `printPersonsWithPredicate`, it expects a data type of `Predicate<Person>`, so the lambda expression belongs to this type. The data type expected by these methods is called the target type. To determine the type of a lambda expression, the Java compiler uses the target

type of the context or situation in which the lambda expression was discovered. Therefore, you can determine the target type only if the Java compiler can determine the target type.

Case using lambda expression:

- variable declaration
- Assignment
- **Return** statement
- Array initialization
- method or constructor parameters
- Lambda expression body
- conditional expression, **?:**
- conversion expression

Target type and method parameters

For method parameters, the Java compiler uses two additional language features to determine the target type: overload resolution and type parameter inference.

Consider the following two functional interfaces ([java.lang.Runnable](#) and [java.util.concurrent.Callable<V>](#)):

```
public interface Runnable {
    void run () ;
}

public interface Callable < V > {
    V call () ;
}
```

Methods **Runnable.run** do not return values **Callable<V>.call**;

Assume that you have overloaded the method as follows **invoke**(see [Defining Methods](#) for details on overloading methods):

```
void invoke ( Runnable r ) {
    r . run () ;
}

< T > T invoke ( Callable < T > c ) {
    return c . call () ;
}
```

Which method will be called in the following statement?

```
String s = invoke( () -> "done" );
```

The method will be called `invoke(Callable<T>)` because the method returns a value and the method `invoke(Runnable)` does not, in this case `() -> "done"` the type of the lambda expression is `Callable<T>`.

Serialization

A lambda expression can be serialized if its target type and its captured parameters are serializable, but, like inner classes, serialization of lambda expressions is strongly discouraged.

method reference

You use lambda expressions to create anonymous methods. However, sometimes, the lambda expression will just call an existing method. In these cases, it is often clearer to refer to the existing method by name. Method references allow you to do this; for cases where there is already Named methods, they are compact, easy-to-read lambda expressions.

Consider the Person class again :

```
public class Person {

    public enum Sex {
        MALE , FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge () {
        // ...
    }

    public Calendar getBirthday () {
        return birthday;
    }

    public static int compareByAge ( Person a, Person b ) {
        return a. birthday . compareTo (b. birthday );
    }
}
```

Assume that the members of your social networking application are contained in an array, and you want to sort the array by age, you can use the following code (find the code excerpt described in this section in the example [MethodReferencesTest](#)):

```
Person [] rosterAsArray = roster. to Array( new Person[ roster . size () ] );
```

```
class PersonAgeComparator implements Comparator<Person> {  
    public int compare(Person a, Person b) {  
        return a.get Birthday() .compare To( b . getBirthday () );  
    }  
}
```

```
Arrays . sort(rosterAsArray,new PersonAgeComparator());
```

The method signature for this call sorting is as follows:

```
static <T> void sort (T[] a, Comparator<? super T> c)
```

Note that **Comparator** the interface is a functional interface, therefore, you can use lambda expressions instead of defining and then creating **Comparator** new instances of the implementing class:

```
Arrays.sort ( rosterAsArray ,  
    (Person a, Person b) -> {  
        return a.get Birthday() .compare To( b . getBirthday () );  
    }  
);
```

However, this **Person** method of comparing the date of birth of two instances already exists **Person.compareByAge**, and you can call this method in the body of a lambda expression:

```
Arrays.sort ( rosterAsArray ,  
    (a, b) -> Person . compare ByAge( a , b )  
);
```

Because this lambda expression calls an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort (rosterAsArray, Person ::compareByAge) ;
```

Method references **Person::compareByAge** are semantically **(a, b) -> Person.compareByAge(a, b)** identical to lambda expressions, each with the following characteristics:

- Its formal parameter list is **Comparator<Person>.compare** copied from , which is **(Person, Person)**.
- Its body calls the method **Person.compareByAge**.

Various method references

There are four ways to reference:

type

Example

type	Example
Reference static method	<code>ContainingClass::staticMethodName</code>
Instance methods that reference a specific object	<code>containingObject::instanceMethodName</code>
Instance methods that reference any object of a specific type	<code>ContainingType::methodName</code>
reference constructor	<code>ClassName::new</code>

Reference static method

A method reference `Person::compareByAge` is a reference to a static method.

Instance methods that reference a specific object

The following is an example of a reference to an instance method of a specific object:

```
class ComparisonProvider {
    public int compare ByName(Person a , Person b ) {
        return a.get Name() .compare To( b . getName ( ) );
    }

    public int compare ByAge(Person a , Person b ) {
        return a.get Birthday() .compare To( b . getBirthday ( ) );
    }
}
ComparisonProvider myComparisonProvider = new ComparisonProvider() ;
Arrays . sort(rosterAsArray, myComparisonProvider::compareByName);
```

The method reference `myComparisonProvider::compareByName` calls the method `compareByName`, which is `myComparisonProvider` part of the object, and the JRE infers the method type parameters, in this case `(Person, Person)`.

A reference to an instance method of any object of a specific type

The following is an example of a reference to an instance method of any object of a specific type:

```
String [ ] stringArray = { "Barbara" , "James" , "Mary" , "John" ,
    "Patricia" , "Robert" , "Michael" , "Linda" } ;
Arrays . sort ( stringArray , String ::compareToIgnoreCase ) ;
```

The equivalent lambda expression of a method reference `String::compareToIgnoreCase` will have a formal parameter list `(String a, String b)`, where `a` and `b` are arbitrary names to better describe this example, and the method reference will invoke the method `a.compareToIgnoreCase(b)`.

reference constructor

new You can reference a constructor by name in the same way as a static method. The following method copies elements from one collection to another:

```
public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>
    DEST transferElements(
        SOURCE sourceCollection,
        Supplier<DEST> collectionFactory) {

    DEST result = collectionFactory.get();
    for (T t : sourceCollection) {
        result.add(t);
    }
    return result;
}
```

A functional interface **Supplier** contains a method that takes no parameters and returns an object **get**, so you can call the method using a lambda expression **transferElements** like this:

```
Set <Person> rosterSetLambda =
    transferElements (roster, () -> { return new HashSet<>(); }) ;
```

You can use constructor references instead of lambda expressions, like this:

```
Set<Person> rosterSet = transfer Elements( roster , HashSet:: new ) ;
```

The Java compiler infers that you want to create a collection containing **Person** elements of type **HashSet**, or, you can specify it as follows:

```
Set<Person> rosterSet = transfer Elements( roster , HashSet<Person>:: new ) ;
```

When to use nested classes, partial classes, anonymous classes, and lambda expressions

As discussed in the Nested Classes section, nested classes enable you to logically group classes that are used in only one place, increase the use of encapsulation, and create more readable and maintainable code. Partial classes, anonymous classes, and lambda expressions also confer these advantages, however, they are intended for use in more specific situations:

- Partial classes: Use this if you need to create multiple instances of a class, access its constructors or introduce new named types (e.g. you need to call other methods later).
- Anonymous class: Use this if you need to declare fields or other methods.

- lambda expression:
 - Use it if you want to encapsulate a single unit of behavior that you want to pass to other code, for example if you want to perform an operation on each element of the collection, or when the process completes, or when the process encounters an error, you would use lambda expression.
 - Use it if you need a simple instance of a functional interface and the preceding conditions don't apply (for example, you don't need constructors, named types, fields, or other methods).
- Nested classes: You need to use this if your requirements are similar to those of local classes, you want to use the type more extensively and you don't need to access local variables or method parameters.

If you need to access non-public fields and methods of the enclosing instance, use a non-static nested class (or inner class), if you don't need this access, use a static nested class.

[Previous article: Anonymous class](#)

[Next article: Enumeration type](#)

