ORACLE Java Documentation

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Classes and Objects
Section: Nested Classes

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

This section covers the following topics:

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters
- Lambda Expressions in GUI Applications
- Syntax of Lambda Expressions
- Accessing Local Variables of the Enclosing Scope
- Target Typing
  - Target Types and Method Arguments
- Serialization

Ideal Use Case for Lambda Expressions

Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria. The following table describes this use case in detail:

Field	Description
Name	Perform action on selected members
Primary Actor	Administrator
Preconditions	Administrator is logged in to the system.
Postconditions	Action is performed only on members that fit the specified criteria.
Main Success Scenario	1. Administrator specifies criteria of members on which to perform a certain action. 2. Administrator specifies an action to perform on those selected members. 3. Administrator selects the **Submit** button. 4. The system finds all members that match the specified criteria. 5. The system performs the specified action on all matching members.
Extensions	1a. Administrator has an option to preview those members who match the specified criteria before he or she specifies the action to be performed or before selecting the **Submit** button.
Frequency of Occurrence	Many times during the day.

Suppose that 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() {
            // ...
        }
    }
```

Suppose that the members of your social networking application are stored in a `List<Person>` instance.

This section begins with a naive approach to this use case. It improves upon this approach with local and anonymous classes, and then finishes with an efficient and concise approach using lambda expressions. Find the code excerpts described in this section in the example `RosterTest`.

Approach 1: Create Methods That Search for Members That Match One Characteristic

One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. The following method prints members that are older than a specified age:

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

Note: A `List` is an ordered `Collection`. A *collection* is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. For more information about collections, see the Collections trail.

This approach can potentially make your application *brittle*, which is the likelihood of an application not working because of the introduction of updates (such as newer data types). Suppose that you upgrade your application and change the structure of the `Person` class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm. You would have to rewrite a lot of your API to accommodate this change. In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

Approach 2: Create More Generalized Search Methods

The following method is more generic than `printPersonsOlderThan`; it prints members within a specified range of ages:

```
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 specified sex, or a combination of a specified gender and age range? What if you decide to change the `Person` class and add other attributes such as relationship status or geographical location? Although this method is more generic than `printPersonsOlderThan`, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

Approach 3: Specify Search Criteria Code in a Local Class

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

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

This method checks each `Person` instance contained in the `List` parameter `roster` whether it satisfies the search criteria specified in the `CheckPerson` parameter `tester` by invoking the method `tester.test`. If the method `tester.test` returns a `true` value, then the method `printPersons` is invoked on the `Person` instance.

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

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

The following class implements the `CheckPerson` interface by specifying an implementation for the method `test`. This method filters members that are eligible for Selective Service in the United States: it returns a `true` value if its `Person` parameter is male and between the ages of 18 and 25:

```
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 create a new instance of it and invoke the `printPersons` method:

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

Although this approach is less brittle—you don't have to rewrite methods if you change the structure of the `Person`—you still have additional code: a new interface and a local class for each search you plan to perform in your application. Because `CheckPersonEligibleForSelectiveService` implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

Approach 4: Specify Search Criteria Code in an Anonymous Class

One of the arguments of the following invocation of the method `printPersons` is an anonymous class that filters members that are eligible for Selective Service in the United States: 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 that you want to perform. However, the syntax of anonymous classes is bulky considering that the `CheckPerson` interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

Approach 5: Specify Search Criteria Code with a Lambda Expression

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.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it. To do this, instead of using an anonymous class expression, you use a *lambda expression*, which is highlighted in the following method invocation:

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

See Syntax of Lambda Expressions for information about how to define lambda expressions.

You can use a standard functional interface in place of the interface `CheckPerson`, which reduces even further the amount of code required.

Approach 6: Use Standard Functional Interfaces with Lambda Expressions

Reconsider the `CheckPerson` interface:

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

This is a very simple interface. It's a functional interface because it contains only one abstract method. This method takes one parameter and returns a `boolean` value. The method is so simple that it might not be worth it to define one in your application. Consequently, the JDK defines several standard functional interfaces, which you can find in the package `java.util.function`.

For example, you can use the `Predicate<T>` interface in place of `CheckPerson`. This interface contains the method `boolean test(T t)`:

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

The interface `Predicate<T>` is an example of a generic interface. (For more information about generics, see the Generics (Updated) lesson.) Generic types (such as generic interfaces) specify one or more type parameters within angle brackets (<>). This interface contains only one type parameter, `T`. When you declare or instantiate a generic type with actual type arguments, you have a parameterized type. For example, the parameterized type `Predicate<Person>` is the following:

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

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

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

```
        }
    }
```

As a result, the following method invocation is the same as when you invoked `printPersons` in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service:

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

This is not the only possible place in this method to use a lambda expression. The following approach suggests other ways to use lambda expressions.

Approach 7: Use Lambda Expressions Throughout Your Application

Reconsider the method `printPersonsWithPredicate` to see where else you could use lambda expressions:

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

This method checks each `Person` instance contained in the `List` parameter `roster` whether it satisfies the criteria specified in the `Predicate` parameter `tester`. If the `Person` instance does satisfy the criteria specified by `tester`, the method `printPerson` is invoked on the `Person` instance.

Instead of invoking the method `printPerson`, you can specify a different action to perform on those `Person` instances that satisfy the criteria specified by `tester`. You can specify this action with a lambda expression. Suppose you want a lambda expression similar to `printPerson`, one that takes one argument (an object of type `Person`) and returns void. Remember, to use a lambda expression, you need to implement a functional interface. In this case, you need a functional interface that contains an abstract method that can take one argument of type `Person` and returns void. The `Consumer<T>` interface contains the method `void accept(T t)`, which has these characteristics. The following method replaces the invocation `p.printPerson()` with an instance of `Consumer<Person>` that invokes the method `accept`:

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

As a result, the following method invocation is the same as when you invoked `printPersons` in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service. The lambda expression used to print members is highlighted:

```
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 your members' profiles than printing them out. Suppose that you want to validate the members' profiles or retrieve their contact information? In this case, you need a functional interface that contains an abstract method that returns a value. The `Function<T,R>` interface contains the method `R apply(T t)`. The following method retrieves the data specified by the parameter `mapper`, and then performs an action on it specified by the parameter `block`:

```
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);
        }
    }
}
```

The following method retrieves the email address from each member contained in `roster` who is eligible for Selective Service and then prints it:

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

Approach 8: Use Generics More Extensively

Reconsider the method `processPersonsWithFunction`. The following is a generic version of it that accepts, as a parameter, a collection that contains elements of any data type:

```
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 e-mail address of members who are eligible for Selective Service, invoke the `processElements` method as follows:

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

This method invocation performs the following actions:

1. Obtains a source of objects from the collection `source`. In this example, it obtains a source of `Person` objects from the collection `roster`. Notice that the collection `roster`, which is a collection of type `List`, is also an object of type `Iterable`.
2. Filters objects that match the `Predicate` object `tester`. In this example, the `Predicate` object is a lambda expression that specifies which members would be eligible for Selective Service.
3. Maps each filtered object to a value as specified by the `Function` object `mapper`. In this example, the `Function` object is a lambda expression that returns the e-mail address of a member.
4. Performs an action on each mapped object as specified by the `Consumer` object `block`. In this example, the `Consumer` object is a lambda expression that prints a string, which is the e-mail address returned by the `Function` object.

You can replace each of these actions with an aggregate operation.

Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

The following example uses aggregate operations to print the e-mail addresses of those members contained in the collection `roster` who are eligible for Selective Service:

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

The following table maps each of the operations the method `processElements` performs with the corresponding aggregate operation:

`processElements` Action	**Aggregate Operation**
Obtain a source of objects	`Stream<E>` **`stream`**`()`
Filter objects that match a `Predicate` object	`Stream<T>` **`filter`**`(Predicate<? super T> predicate)`
Map objects to another value as specified by a `Function` object	`<R> Stream<R>` **`map`**`(Function<? super T,? extends R> mapper)`
Perform an action as specified by a `Consumer` object	`void` **`forEach`**`(Consumer<? super T> action)`

The operations `filter`, `map`, and `forEach` are *aggregate operations*. Aggregate operations process elements from a stream, not directly from a collection (which is the reason why the first method invoked 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 collection, through a pipeline. A *pipeline* is a sequence of stream operations, which in this example is `filter`- `map`- `forEach`. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

For a more thorough discussion of aggregate operations, see the Aggregate Operations lesson.

Lambda Expressions in GUI Applications

To process events in a graphical user interface (GUI) application, such as keyboard actions, mouse actions, and scroll actions, you typically create event handlers, which usually involves implementing a particular interface. Often, event handler interfaces are functional interfaces; they tend to have only one method.

In the JavaFX example `HelloWorld.java` (discussed in the previous section Anonymous Classes), you can replace the highlighted 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 invocation `btn.setOnAction` specifies what happens when you select the button represented by the `btn` object. This method requires an object of type `EventHandler<ActionEvent>`. The `EventHandler<ActionEvent>` interface contains only one method, `void handle(T event)`. This interface is a functional interface, so you could use the following highlighted lambda expression to replace it:

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

Syntax of Lambda Expressions

A lambda expression consists of the following:

- A comma-separated list of formal parameters enclosed in parentheses. The `CheckPerson.test` method contains one parameter, `p`, which represents an instance of the `Person` class.

  Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:

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

- The arrow token, `->`

- A body, which consists of a single expression or a statement block. This example uses the following expression:

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

  If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

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

  A return statement is not an expression; in a lambda expression, you must enclose statements in braces (`{}`). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

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

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name.

The following example, `Calculator`, is an example of lambda expressions that take more than one formal parameter:

```
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 is specified by an instance of `IntegerMath`. The example defines two operations with lambda expressions, `addition` and `subtraction`. The example prints the following:

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

Accessing Local Variables of the Enclosing Scope

Like local and anonymous classes, lambda expressions can capture variables; they 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 (see Shadowing for more information). Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are 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) {

            int z = 2;

            Consumer<Integer> myConsumer = (y) ->
            {
                // The following statement causes the compiler to generate
                // the error "Local variable z defined in an enclosing scope
                // must be final or effectively final"
                //
                // z = 99;

                System.out.println("x = " + x);
                System.out.println("y = " + y);
                System.out.println("z = " + z);
                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 generates the following output:

```
x = 23
y = 23
z = 2
this.x = 1
LambdaScopeTest.this.x = 0
```

If you substitute the parameter `x` in place of `y` in the declaration of the lambda expression `myConsumer`, then the compiler generates an error:

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

The compiler generates the error "Lambda expression's parameter x cannot redeclare another local variable defined in an enclosing scope" because the lambda expression does not introduce a new level of scoping. Consequently, you can directly access fields, methods, and local variables of the enclosing scope. For example, the lambda expression directly accesses the parameter `x` of the method `methodInFirstLevel`. To access variables in the enclosing class, use the keyword `this`. In this example, `this.x` refers to the member variable `FirstLevel.x`.

However, like local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final. In this example, the variable `z` is effectively final; its value is never changed after it's initialized. However, suppose that you add the following assignment statement in the the lambda expression `myConsumer`:

```
Consumer<Integer> myConsumer = (y) -> {
    z = 99;
    // ...
}
```

Because of this assignment statement, the variable `z` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "Local variable z defined in an enclosing scope must be final or effectively final".

Target Typing

How do you determine the type of a lambda expression? Recall the lambda expression that selected members who are male and between the ages 18 and 25 years:

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

This lambda expression was used in the following two methods:

- `public static void printPersons(List<Person> roster, CheckPerson tester)` in Approach 3: Specify Search Criteria Code in a Local Class

- `public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)` in Approach 6: Use Standard Functional Interfaces with Lambda Expressions

When the Java runtime invokes the method `printPersons`, it's expecting a data type of `CheckPerson`, so the lambda expression is of this type. However, when the Java runtime invokes the method `printPersonsWithPredicate`, it's expecting a data type of `Predicate<Person>`, so the lambda expression is of this type. The data type that these methods expect 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 found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, `?:`
- Cast expressions

Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument 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();
}
```

The method `Runnable.run` does not return a value, whereas `Callable<V>.call` does.

Suppose that you have overloaded the method `invoke` as follows (see Defining Methods for more information about overloading methods):

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

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

Which method will be invoked in the following statement?

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

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

Serialization

You can serialize a lambda expression if its target type and its captured arguments are serializable. However, like inner classes, the serialization of lambda expressions is strongly discouraged.

Previous page: Anonymous Classes
Next page: Method References