Documentación

Los tutoriales de Java ™

Trail: Aprender la lengua de Java
Lección:
Sección de clases y objetos **:** Clases anidadas

Los Tutoriales de Java se han escrito para JDK 8. Los ejemplos y las prácticas que se describen en esta página no aprovechan las mejoras introducidas en versiones posteriores.

Expresiones Lambda

Un problema con las clases anónimas es que si la implementación de su clase anónima es muy simple, como una interfaz que contiene solo un método, entonces la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. En estos casos, por lo general, intenta pasar la funcionalidad como argumento a otro método, como qué acción debe realizarse cuando alguien hace clic en un botón. Las expresiones de Lambda le permiten hacer esto, para tratar la funcionalidad como argumento de método o código como datos.

La sección anterior, Clases anónimas , le muestra cómo implementar una clase base sin darle un nombre. Aunque esto a menudo es más conciso que una clase con nombre, para las clases con un solo método, incluso una clase anónima parece un poco excesiva y engorrosa. Las expresiones Lambda le permiten expresar instancias de clases de método único de forma más compacta.

Esta sección cubre los siguientes temas:

Caso de uso ideal para Lambda Expressions

Supongamos que está creando una aplicación de red social. Desea crear una característica que permita a un administrador realizar cualquier tipo de acción, como enviar un mensaje, a los miembros de la aplicación de redes sociales que satisfagan ciertos criterios. La siguiente tabla describe este caso de uso en detalle:

Campo	Descripción
Nombre	Realizar acción en los miembros seleccionados
Actor principal	Administrador
Condiciones previas	El administrador ha iniciado sesión en el sistema.
Postcondiciones	La acción se realiza solo en miembros que se ajustan a los criterios especificados.
Escenario de éxito principal	1. El administrador especifica los criterios de los miembros para realizar una determinada acción. 2. El administrador especifica una acción para realizar en los miembros seleccionados. 3. El administrador selecciona el botón **Enviar** . 4. El sistema encuentra todos los miembros que coinciden con los criterios especificados. 5. El sistema realiza la acción especificada en todos los miembros coincidentes.
Extensiones	1a. El administrador tiene una opción para obtener una vista previa de los miembros que coinciden con los criterios especificados antes de que él o ella especifique la acción que se realizará o antes de seleccionar el botón **Enviar** .

Frecuencia de ocurrencia	Muchas veces durante el día.

Supongamos que los miembros de esta aplicación de redes sociales están representados por la siguiente `Person`clase:

```
clase pública Persona {

    public enum Sex {
        MACHO FEMENINO
    }

    Nombre de cadena;
    Cumpleaños de LocalDate;
    Sexo sexual;
    String emailAddress;

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

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

Supongamos que los miembros de su aplicación de red social se almacenan en una `List<Person>`instancia.

Esta sección comienza con un enfoque ingenuo para este caso de uso. Mejora este enfoque con clases locales y anónimas, y luego finaliza con un enfoque eficiente y conciso utilizando expresiones lambda. Encuentre los extractos del código descritos en esta sección en el ejemplo `RosterTest`.

Enfoque 1: crear métodos que busquen miembros que coincidan con una característica

Un enfoque simplista es crear varios métodos; cada método busca miembros que coincidan con una característica, como el sexo o la edad. El siguiente método imprime miembros que son anteriores a una edad especificada:

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

Nota : A `List`es un pedido `Collection`. Una *colección* es un objeto que agrupa múltiples elementos en una sola unidad. Las colecciones se utilizan para almacenar, recuperar, manipular y comunicar datos agregados. Para obtener más información acerca de las colecciones, consulte el rastro de Colecciones .

Este enfoque puede hacer que su aplicación se *vuelva frágil* , que es la probabilidad de que una aplicación no funcione debido a la introducción de actualizaciones (como los tipos de datos más nuevos). Supongamos que actualiza su aplicación y cambia la estructura de la `Person`clase de manera que contenga diferentes variables miembro; quizás la clase registra y mide las edades con un tipo de datos o algoritmo diferente. Tendría que volver a escribir una gran cantidad de su API para adaptarse a este cambio. Además, este enfoque es innecesariamente restrictivo; ¿y si quisieras imprimir miembros menores de cierta edad, por ejemplo?

Enfoque 2: crear métodos de búsqueda más generalizados

El siguiente método es más genérico que `printPersonsOlderThan`; imprime miembros dentro de un rango específico de edades:

```
public static void printPersonsWithinAgeRange (
    Lista <Person> roster, int low, int high) {
    para (Persona p: roster) {
        if (bajo <= p.getAge () && p.getAge () <alto) {
            p.printPerson ();
        }
    }
}
```

¿Qué sucede si desea imprimir miembros de un sexo específico o una combinación de un género y un rango de edad específicos? ¿Qué sucede si decide cambiar la `Person`clase y agregar otros atributos como el estado de la relación o la ubicación geográfica? Aunque este método es más genérico que `printPersonsOlderThan`, tratar de crear un método separado para cada consulta de búsqueda posible aún puede llevar a la fragilidad del código. En su lugar, puede separar el código que especifica los criterios para los que desea buscar en una clase diferente.

Método 3: especifique el código de criterios de búsqueda en una clase local

El siguiente método imprime miembros que coinciden con los criterios de búsqueda que especifique:

```
public static void printPersons (
    Listar lista <Person>, tester CheckPerson) {
    para (Persona p: roster) {
        if (tester.test (p)) {
            p.printPerson ();
        }
    }
}
```

Este método comprueba cada `Person`instancia contenida en el `List`parámetro `roster`si cumple los criterios de búsqueda especificados en el `CheckPerson`parámetro `tester`invocando el método `tester.test`. Si el método `tester.test`devuelve un `true`valor, el método `printPersons`se invoca en la `Person`instancia.

Para especificar los criterios de búsqueda, implementa la `CheckPerson`interfaz:

```
interfaz CheckPerson {
    prueba booleana (Persona p);
}
```

La siguiente clase implementa la `CheckPerson`interfaz especificando una implementación para el método `test`. Este método filtra los miembros que son elegibles para el servicio selectivo en los Estados Unidos: devuelve un `true`valor si su `Person`parámetro es masculino y entre las edades de 18 y 25:

```
la clase CheckPersonEligibleForSelectiveService implementa CheckPerson {
    prueba booleana pública (Persona p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge ()> = 18 &&
            p.getAge () <= 25;
    }
}
```

Para usar esta clase, crea una nueva instancia e invoca el `printPersons`método:

```
printPersons (
    lista, nuevo CheckPersonEligibleForSelectiveService ());
```

Aunque este enfoque es menos frágil, no es necesario reescribir los métodos si cambia la estructura del mismo; `Person`aún tiene código adicional: una nueva interfaz y una clase local para cada búsqueda que planea realizar en su aplicación. Debido a que `CheckPersonEligibleForSelectiveService` implementa una interfaz, puede usar una clase anónima en lugar de una clase local y omitir la necesidad de declarar una nueva clase para cada búsqueda.

Método 4: especifique el código de criterios de búsqueda en una clase anónima

Uno de los argumentos de la siguiente invocación del método `printPersons`es una clase anónima que filtra a los miembros que son elegibles para el Servicio selectivo en los Estados Unidos: los que son hombres y tienen entre 18 y 25 años:

```
printPersons (
    lista,
    nuevo CheckPerson () {
        prueba booleana pública (Persona p) {
            return p.getGender () == Person.Sex.MALE
                && p.getAge ()> = 18
                && p.getAge () <= 25;
        }
    }
);
```

Este enfoque reduce la cantidad de código requerido porque no tiene que crear una nueva clase para cada búsqueda que desea realizar. Sin embargo, la sintaxis de las clases anónimas es voluminosa teniendo en cuenta que la `CheckPerson`interfaz contiene solo un método. En este caso, puede usar una expresión lambda en lugar de una clase anónima, como se describe en la siguiente sección.

Método 5: especifique el código de criterios de búsqueda con una expresión lambda

La `CheckPerson` interfaz es una *interfaz funcional* . Una interfaz funcional es cualquier interfaz que contiene solo un método abstracto . (Una interfaz funcional puede contener uno o más métodos predeterminados o métodos estáticos .) Debido a que una interfaz funcional contiene solo un método abstracto, puede omitir el nombre de ese método cuando lo implemente. Para hacer esto, en lugar de utilizar una expresión de clase anónima, utiliza una *expresión lambda* , que se resalta en la siguiente invocación de método:

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

Consulte Sintaxis de expresiones Lambda para obtener información sobre cómo definir expresiones lambda.

Puede utilizar una interfaz funcional estándar en lugar de la interfaz `CheckPerson`, lo que reduce aún más la cantidad de código requerido.

Método 6: utilice interfaces funcionales estándar con expresiones lambda

Reconsiderar la `CheckPerson`interfaz:

```
interfaz CheckPerson {
    prueba booleana (Persona p);
}
```

Esta es una interfaz muy simple. Es una interfaz funcional porque contiene solo un método abstracto. Este método toma un parámetro y devuelve un `boolean`valor. El método es tan simple que puede no valer la pena definir uno en su aplicación. En consecuencia, el JDK define varias interfaces funcionales estándar, que puede encontrar en el paquete `java.util.function`.

Por ejemplo, puede usar la `Predicate<T>` interfaz en lugar de `CheckPerson`. Esta interfaz contiene el método `boolean test(T t)`:

```
interfaz Predicado <T> {
    prueba booleana (T t);
}
```

La interfaz `Predicate<T>`es un ejemplo de una interfaz genérica. (Para obtener más información sobre los genéricos, consulte la lección sobre genéricos (actualización) .) Los tipos genéricos (como las interfaces genéricas) especifican uno o más parámetros de tipo entre corchetes angulares (`<>`). Esta interfaz contiene sólo un parámetro de tipo, `T`. Cuando declara o crea una instancia de un tipo genérico con argumentos de tipo reales, tiene un tipo parametrizado. Por ejemplo, el tipo parametrizado `Predicate<Person>`es el siguiente:

```
interfaz Predicado < Person> {
    prueba booleana ( Persont);
}
```

Este tipo parametrizado contiene un método que tiene el mismo tipo de retorno y parámetros que `CheckPerson.boolean test(Person p)`. En consecuencia, puede usar `Predicate<T>`en lugar de `CheckPerson`como lo demuestra el siguiente método:

```
public static void printPersonsWithPredicate (
    Lista <persona> lista, Predicado <Persona> tester) {
    para (Persona p: roster) {
        if (tester.test (p)) {
            p.printPerson ();
        }
    }
}
```

Como resultado, la siguiente invocación de método es la misma que cuando invocó `printPersons`en el Método 3: Especificar el Código de Criterio de Búsqueda en una Clase Local para obtener miembros que son elegibles para el Servicio Selectivo:

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

Este no es el único lugar posible en este método para usar una expresión lambda. El siguiente enfoque sugiere otras formas de usar expresiones lambda.

Método 7: use expresiones Lambda en toda su aplicación

Reconsidere el método `printPersonsWithPredicate` para ver dónde más podría usar las expresiones lambda:

```
public static void printPersonsWithPredicate (
    Lista <persona> lista, Predicado <Persona> tester) {
    para (Persona p: roster) {
        if (tester.test (p)) {
            p.printPerson ();
        }
    }
}
```

Este método comprueba cada `Person`instancia contenida en el `List`parámetro `roster`si cumple los criterios especificados en el `Predicate`parámetro `tester`. Si la `Person`instancia satisface los criterios especificados por `tester`, el método `printPersron`se invoca en la `Person`instancia.

En lugar de invocar el método `printPerson`, puede especificar una acción diferente para realizar en aquellas `Person`instancias que satisfagan los criterios especificados por `tester`. Puede especificar esta acción con una expresión lambda. Supongamos que desea una expresión lambda similar a `printPerson`, una que toma un argumento (un objeto de tipo `Person`) y devuelve vacía. Recuerde, para usar una expresión lambda, debe

implementar una interfaz funcional. En este caso, necesita una interfaz funcional que contenga un método abstracto que pueda tomar un argumento de tipo `Person`y devuelva el vacío. La `Consumer<T>` interfaz contiene el método `void accept(T t)`, que tiene estas características. El siguiente método reemplaza la invocación `p.printPerson()` con una instancia de `Consumer<Person>`que invoca el método `accept`:

```
public static void processPersons (
    Lista <Person> lista,
    Predicate <Person> tester,
    Bloque <persona> del consumidor ) {
        para (Persona p: roster) {
            if (tester.test (p)) {
                block.accept (p);
            }
        }
}
```

Como resultado, la siguiente invocación de método es la misma que cuando invocó `printPersons`en el Método 3: especificar el código de criterio de búsqueda en una clase local para obtener miembros que son elegibles para el servicio selectivo. Se resalta la expresión lambda utilizada para imprimir miembros:

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

¿Qué sucede si desea hacer más con los perfiles de sus miembros que imprimirlos? Supongamos que quiere validar los perfiles de los miembros o recuperar su información de contacto. En este caso, necesita una interfaz funcional que contenga un método abstracto que devuelva un valor. La `Function<T,R>` interfaz contiene el método `R apply(T t)`. El siguiente método recupera los datos especificados por el parámetro `mapper`y luego realiza una acción en él especificada por el parámetro `block`:

```
public static void processPersonsWithFunction (
    Lista <Person> lista,
    Predicate <Person> tester,
    Función <Person, String> mapper,
    Bloque <String> del consumidor) {
    para (Persona p: roster) {
        if (tester.test (p)) {
            String data = mapper.apply (p);
            block.accept (datos);
        }
    }
}
```

El siguiente método recupera la dirección de correo electrónico de cada miembro incluido en `roster`quién es elegible para el Servicio selectivo y luego lo imprime:

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

Método 8: use genéricos más extensamente

Reconsidere el método `processPersonsWithFunction`. La siguiente es una versión genérica que acepta, como parámetro, una colección que contiene elementos de cualquier tipo de datos:

```
público static <X, Y> void processElements (
    Fuente Iterable <X>,
    Predicado <X> probador,
    Mapeador <X, Y> de funciones,
    Bloque <Y> del consumidor) {
    para (X p: fuente) {
        if (tester.test (p)) {
            Y data = mapper.apply (p);
            block.accept (datos);
        }
    }
}
```

Para imprimir la dirección de correo electrónico de los miembros que son elegibles para el Servicio Selectivo, invoque el `processElements`método de la siguiente manera:

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

Esta invocación de método realiza las siguientes acciones:

1. Obtiene una fuente de objetos de la colección `source`. En este ejemplo, obtiene una fuente de `Person`objetos de la colección `roster`. Observe que la colección `roster`, que es una colección de tipo `List`, también es un objeto de tipo `Iterable`.
2. Filtra los objetos que coinciden con el `Predicate`objeto `tester`. En este ejemplo, el `Predicate`objeto es una expresión lambda que especifica qué miembros serían elegibles para el servicio selectivo.
3. Mapas de cada objeto filtrado a un valor especificado por el `Function`objeto `mapper`. En este ejemplo, el `Function`objeto es una expresión lambda que devuelve la dirección de correo electrónico de un miembro.
4. Realiza una acción en cada objeto mapeado según lo especificado por el `Consumer`objeto `block`. En este ejemplo, el `Consumer`objeto es una expresión lambda que imprime una cadena, que es la dirección de correo electrónico devuelta por el `Function`objeto.

Puede reemplazar cada una de estas acciones con una operación agregada.

Método 9: utilice operaciones agregadas que acepten expresiones Lambda como parámetros

El siguiente ejemplo usa operaciones agregadas para imprimir las direcciones de correo electrónico de aquellos miembros contenidos en la colección `roster`que son elegibles para el Servicio Selectivo:

```
lista
    .corriente()
    .filtrar(
        p -> p.getGender () == Person.Sex.MALE
            && p.getAge ()> = 18
            && p.getAge () <= 25)
    .map (p -> p.getEmailAddress ())
    .forEach (correo electrónico -> System.out.println (correo electrónico));
```

La siguiente tabla mapea cada una de las operaciones que `processElements`realiza el método con la operación agregada correspondiente:

`processElements` Acción	Operación Agregada
Obtener una fuente de objetos	`Stream<E>` **`stream`**`()`
Filtrar objetos que coinciden con un `Predicate`objeto	`Stream<T>` **`filter`**`(Predicate<? super T> predicate)`
Asignar objetos a otro valor según lo especificado por un `Function`objeto	`<R> Stream<R>` **`map`**`(Function<? super T,? extends R> mapper)`
Realice una acción según lo especificado por un `Consumer`objeto	`void` **`forEach`**`(Consumer<? super T> action)`

Las operaciones `filter`, `map`y `forEach`son *operaciones agregadas* . Las operaciones agregadas procesan elementos de una secuencia, no directamente de una colección (que es la razón por la cual el primer método invocado en este ejemplo es `stream`). Una *secuencia* es una secuencia de elementos. A diferencia de una colección, no es una estructura de datos que almacena elementos. En cambio, una secuencia transporta valores desde una fuente, como la recopilación, a través de una canalización. Una *tubería* es una secuencia de operaciones de flujo, que en este ejemplo es `filter`- `map`- `forEach`. Además, las operaciones agregadas generalmente aceptan expresiones lambda como parámetros, lo que le permite personalizar su comportamiento.

Para una discusión más completa de las operaciones agregadas, vea la lección Operaciones agregadas .

Expresiones Lambda en aplicaciones GUI

Para procesar eventos en una aplicación de interfaz gráfica de usuario (GUI), como acciones del teclado, acciones del mouse y acciones de desplazamiento, normalmente se crean manejadores de eventos, lo que generalmente implica implementar una interfaz particular. A menudo, las interfaces del controlador de eventos son interfaces funcionales; ellos tienden a tener solo un método.

En el ejemplo de JavaFX (analizado en la sección previa Clases anónimas), puede reemplazar la clase anónima resaltada con una expresión lambda en esta declaración: `HelloWorld.java`

```
btn.setOnAction ( new EventHandler <ActionEvent> () {

    @Override
    public void handle (evento ActionEvent) {
        System.out.println ("¡Hola mundo!");
```

```
            }
        } );
```

La invocación al método `btn.setOnAction`especifica qué sucede cuando se selecciona el botón representado por el `btn`objeto. Este método requiere un objeto de tipo `EventHandler<ActionEvent>`. La `EventHandler<ActionEvent>` interfaz contiene sólo un método, `void handle(T event)`. Esta interfaz es una interfaz funcional, por lo que podría usar la siguiente expresión lambda resaltada para reemplazarla:

```
btn.setOnAction (
   event -> System.out.println ("¡Hola mundo!")
);
```

Sintaxis de Lambda Expressions

Una expresión lambda consiste en lo siguiente:

- Una lista de parámetros formales separados por comas encerrados entre paréntesis. El `CheckPerson.test`método contiene un parámetro, `p`que representa una instancia de la `Person`clase.

  Nota : Puede omitir el tipo de datos de los parámetros en una expresión lambda. Además, puede omitir los paréntesis si solo hay un parámetro. Por ejemplo, la siguiente expresión lambda también es válida:

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

- La ficha de flecha `->`

- Un cuerpo, que consiste en una sola expresión o un bloque de instrucción. Este ejemplo usa la siguiente expresión:

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

  Si especifica una sola expresión, el tiempo de ejecución de Java evalúa la expresión y luego devuelve su valor. Alternativamente, puede usar una declaración de devolución:

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

  Una declaración de devolución no es una expresión; en una expresión lambda, debe adjuntar declaraciones entre llaves (`{}`). Sin embargo, no es necesario que encierre una invocación de método nulo entre llaves. Por ejemplo, la siguiente es una expresión lambda válida:

  ```
  correo electrónico -> System.out.println (correo electrónico)
  ```

Tenga en cuenta que una expresión lambda se parece mucho a una declaración de método; puede considerar las expresiones lambda como métodos anónimos: métodos sin nombre.

El siguiente ejemplo `Calculator`,, es un ejemplo de expresiones lambda que toman más de un parámetro formal:

```
Calculadora de clase pública {

    interfaz IntegerMath {
        operación int (int a, int b);
    }

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

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

        Calculadora myApp = nueva Calculadora ();
        Suma IntegerMath = (a, b) -> a + b;
        IntegerMath resta = (a, b) -> a - b;
        System.out.println ("40 + 2 =" +
            myApp.operateBinary (40, 2, adición));
        System.out.println ("20 - 10 =" +
            myApp.operateBinary (20, 10, resta));
    }
}
```

El método `operateBinary`realiza una operación matemática en dos operandos enteros. La operación en sí está especificada por una instancia de `IntegerMath`. El ejemplo define dos operaciones con expresiones lambda, `addition`y `subtraction`. El ejemplo imprime lo siguiente:

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

Acceder a las variables locales del alcance adjunto

Al igual que las clases locales y anónimas, las expresiones lambda pueden capturar variables ; tienen el mismo acceso a las variables locales del alcance adjunto. Sin embargo, a diferencia de las clases locales y anónimas, las expresiones lambda no tienen ningún problema de sombreado (consulte Sombreado para obtener más información). Las expresiones Lambda tienen un alcance léxico. Esto significa que no heredan ningún nombre de un supertipo ni introducen un nuevo nivel de alcance. Las declaraciones en una expresión lambda se interpretan tal como están en el entorno circundante. El siguiente ejemplo `LambdaScopeTest`, demuestra esto:

```
import java.util.function.Consumer;

clase pública LambdaScopeTest {

    public int x = 0;

    clase FirstLevel {

        public int x = 1;

        void methodInFirstLevel (int x) {

            // La siguiente declaración hace que el compilador genere
            // el error "variables locales a las que se hace referencia desde una expresión lambda
            // debe ser final o efectivamente final "en la declaración A:
            //
            // x = 99;

            Consumidor <Entero> myConsumer = (y) ->
            {
                System.out.println ("x =" + x); // Declaración 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);
    }
}
```

Este ejemplo genera el siguiente resultado:

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

Si sustituye el parámetro `x`en lugar de `y`en la declaración de la expresión lambda `myConsumer`, el compilador genera un error:

```
Consumidor <Entero> myConsumer = (x) -> {
    // ...
}
```

El compilador genera el error "la variable x ya está definida en el método methodInFirstLevel (int)" porque la expresión lambda no introduce un nuevo nivel de ámbito. En consecuencia, puede acceder directamente a los campos, métodos y variables locales del ámbito adjunto. Por ejemplo, la expresión lambda accede directamente al parámetro `x`del método `methodInFirstLevel`. Para acceder a las variables en la clase adjunta, use la palabra clave `this`. En este ejemplo, se `this.x`refiere a la variable miembro `FirstLevel.x`.

Sin embargo, al igual que las clases locales y anónimas, una expresión lambda solo puede acceder a las variables locales y los parámetros del bloque adjunto que son finales o efectivamente finales. Por ejemplo, suponga que agrega la siguiente instrucción de asignación inmediatamente

después de la `methodInFirstLevel`declaración de definición:

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

Debido a esta declaración de asignación, la variable `FirstLevel.x`ya no es efectivamente definitiva. Como resultado, el compilador de Java genera un mensaje de error similar a "las variables locales a las que se hace referencia desde una expresión lambda deben ser finales o efectivamente finales", donde la expresión lambda `myConsumer`intenta acceder a la `FirstLevel.x`variable:

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

Mecanografía de destino

¿Cómo se determina el tipo de expresión lambda? Recuerde la expresión lambda que seleccionó miembros masculinos y entre las edades de 18 y 25 años:

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

Esta expresión lambda se utilizó en los dos métodos siguientes:

- `public static void printPersons(List<Person> roster, CheckPerson tester)` en el Enfoque 3: Especificar el Código de Criterios de Búsqueda en una Clase Local
- `public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)` en el Enfoque 6: Usar interfaces funcionales estándar con expresiones lambda

Cuando el tiempo de ejecución de Java invoca el método `printPersons`, espera un tipo de datos `CheckPerson`, por lo que la expresión lambda es de este tipo. Sin embargo, cuando el tiempo de ejecución de Java invoca el método `printPersonsWithPredicate`, espera un tipo de datos `Predicate<Person>`, por lo que la expresión lambda es de este tipo. El tipo de datos que esperan estos métodos se llama *tipo de destino* . Para determinar el tipo de expresión lambda, el compilador de Java usa el tipo de destino del contexto o situación en la que se encontró la expresión lambda. De esto se desprende que solo puede usar expresiones lambda en situaciones en las que el compilador de Java puede determinar un tipo de destino:

- Declaraciones de variables
- Asignaciones
- Declaraciones de devolución
- Inicializadores de matriz
- Método o argumentos del constructor
- Cuerpos de expresión lambda
- Expresiones condicionales `?:`
- Expresiones de molde

Tipos de objetivos y argumentos de métodos

Para los argumentos de método, el compilador de Java determina el tipo de destino con otras dos características de lenguaje: resolución de sobrecarga e inferencia de argumento de tipo.

Considere las siguientes dos interfaces funcionales (`java.lang.Runnable`y `java.util.concurrent.Callable<V>`):

```
interfaz pública Runnable {
    void run ();
}

interfaz pública Llamable <V> {
    V llamada ();
}
```

El método `Runnable.run`no devuelve un valor, mientras que lo `Callable<V>.call`hace.

Supongamos que ha sobrecargado el método de la `invoke`siguiente manera (consulte Definición de métodos para obtener más información sobre métodos de sobrecarga):

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

```
<T> T invoke (que se puede llamar <T> c) {
    devuelve c.call ();
}
```

¿Qué método se invocará en la siguiente declaración?

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

El método `invoke(Callable<T>)` se invocará porque ese método devuelve un valor; el método `invoke(Runnable)` no. En este caso, el tipo de la expresión lambda `() -> "done"` es `Callable<T>`.

Publicación por entregas

Puede serializar una expresión lambda si su tipo de destino y sus argumentos capturados son serializables. Sin embargo, al igual que las clases internas , la serialización de las expresiones lambda se desaconseja enérgicamente.

Página anterior: Clases anónimas
Página siguiente: Método de referencias