Table of Contents

Clases Anidadas (Nested Class)

Una clase Anidada es una clase definida adentro de otra clase. Una clase anidada que no es estática se llama una clase interna. Hay 4 tipos de clases interna.

- **Clase Interna Miembro**: No es estática y es definida como un miembro de la clase.

- **Clase Interna Local**: Se define dentro de un método.

- **Clase Interna Anónima**: Es un caso especial por que no tiene nombre.

- **Clase Anidada Estática**: es una clase definida al mismo nivel de atributo estático variable.

Clases Internas Miembros (Member Inner Classes)

Una clase interna miembro es definida al mismo nivel que un miembro de una clase (como un método atributo o constructor), y tiene la siguientes características:

- Pueden ser declaradas public, private, protected o default access.

- Pueden heredar de alguna clase o implementar interfaces.

- Pueden ser abstractas o final.

- Puede acceder a miembros de la clase externa incluyendo miembros privados.

```
public class Outer {
	private String greeting = "Hi";
	protected class Inner {
		public int repeat = 3;
		public void go() {
			for (int i = 0; i < repeat; i++)
			System.out.println(greeting);
		}
	}
	public void callInner() {
		Inner inner = new Inner();
		inner.go();
	}
	public static void main(String[] args) {
		Outer outer = new Outer();
		outer.callInner();
	}
}
```

Clases Internas Locales (Local Inner Classes)

Una clase interna local es una clase anidada definida dentro de un método. Al igual que las variables locales, una declaración de clase interna local no existe hasta que se invoca el método, y sale del alcance cuando el método pierde scope. Esto significa que puede crear instancias solo desde dentro del método. Esas instancias todavía se pueden devolver desde el método.

Las clases internas locales tienen las siguientes propiedades:

- No tienen un modificador de acceso especifico.

- No pueden ser declaradas static y no pueden tener miembros staticos.

- La clase externa accede a todos los métodos de la clase interna sin importar su modificador de visibilidad.

- La clase interna no tiene acceso a las variables locales de un método a menos que estas sean declaradas final.

```
public class Outer {
	private int length = 5;
	public void calculate() {
		final int width = 20;
		class Inner {
			public void multiply() {
			System. out .println(length * width);
			}
		}
		Inner inner = new Inner();
		inner.multiply();
	}
	public static void main(String[] args) {
		Outer outer = new Outer();
		outer.calculate();
	}
}
```

Clases Internas Anónimas (Anonymous Inner Classes)

Una clase interna anónima es una clase interna local que no tiene nombre. Se declara y crea una instancia de todo en una declaración utilizando la palabra clave **new** . Se requieren clases internas anónimas para extender una clase existente o implementar una interfaz existente. Son útiles cuando tiene una implementación corta que no se utilizará en ningún otro lugar. Aquí hay un ejemplo:

```
public class AnonInner {
	abstract class SaleTodayOnly {
		abstract int dollarsOff();
	}
	public int admission(int basePrice) {
		SaleTodayOnly sale = new SaleTodayOnly() {
			int dollarsOff() { return 3; }
		};
	return basePrice - sale.dollarsOff();
	}
}
```

Clases Anidada Estatica (Static Nested Classes)

El tipo final de clase anidada no es una clase interna. Una clase anidada estática es una clase estática definida en el nivel miembro. Se puede crear una instancia sin un objeto de la clase envolvente, por lo que no puede acceder a las variables de la instancia sin un objeto explícito de la clase envolvente. Por ejemplo, ***new OuterClass().var*** permite el acceso a la variable de instancia var.

En otras palabras, es una clase regular salvo por lo siguiente:

- El anidamiento crea un espacio de nombres porque el nombre de la clase adjunta se debe usar para referirse a él.
- Puede ser private o usar algún modificador de visibilidad o encapsulada.
- La clase que la contiene puede referirse a los campos y métodos de la clase anidada estática.

```
public class Enclosing {
	static class Nested {
		private int price = 6;
	}
	public static void main(String[] args) {
		Nested nested = new Nested();
		System.out.println(nested.price);
	}
}
```

Se puede importar una clase anidada estatica:

```
package bird;
public class Toucan {
	public static class Beak {}
}

package watcher;
import bird.Toucan.Beak;
// regular import ok

public class BirdWatcher {
	Beak beak;
}
```

TABLE 1.2 Types of nested classes

	Member inner class	**Local inner class**	**Anonymous inner class**	`static` **nested class**
Access modifiers allowed	`public`, `protected`, `private`, or default access	None. Already local to method.	None. Already local to statement.	`public`, `protected`, `private`, or default access
Can extend any class and any number of interfaces	Yes	Yes	No—must have exactly one superclass or one interface	Yes
Can be `abstract`	Yes	Yes	N/A—because no class definition	Yes
Can be `final`	Yes	Yes	N/A—because no class definition	Yes
Can access instance members of enclosing class	Yes	Yes	Yes	No (not directly; requires an instance of the enclosing class)
Can access local variables of enclosing class	No	Yes—if `final` or effectively final	Yes—if `final` or effectively final	No
Can declare `static` methods	No	No	No	Yes

Múltiples declaraciones de clase en un archivo.

javac no prohíbe activamente esto, pero tiene una limitación que significa que no querrás referirte a una clase de nivel superior desde otro archivo a menos que tenga el mismo nombre que el archivo en el que está.

Supongamos que tiene dos archivos, Foo.java y Bar.java.

Foo.java contiene:

- clase publica foo

Bar.java contiene:

- clase pública Bar

- clase Baz

Digamos también que todas las clases están en el mismo paquete (y los archivos están en el mismo directorio).

¿Qué sucede si Foo.java se refiere a Baz pero no a Bar y tratamos de compilar Foo.java? La compilación falla con un error como este:

```
Foo.java:2: cannot find symbol
symbol  : class Baz
location: class Foo
  private Baz baz;
          ^
1 error
```

Esto tiene sentido si lo piensas. Si Foo.java se refiere a Baz, pero no hay Baz.java (o Baz.class), ¿cómo puede javac saber en qué archivo fuente buscar?

Si, en cambio, le dice a javac que compile Foo.java y Bar.java al mismo tiempo, o incluso si ya había compilado Bar.java (dejando el Baz.class donde javac puede encontrarlo), este error desaparece. Sin embargo, esto hace que tu proceso de construcción se sienta muy poco confiable y escamoso.

Debido a la limitación real, que es más bien "no se refiera a una clase de nivel superior de otro archivo a menos que tenga el mismo nombre que el archivo en el que está o también se está refiriendo a una clase que está en ese mismo archivo que se llama la misma cosa que el archivo "es un poco difícil de seguir, la gente suele ir con la convención mucho más directa (aunque más estricta) de simplemente poner una clase de nivel superior en cada archivo. Esto también es mejor si alguna vez cambia de opinión sobre si una clase debe ser pública o no.

A veces realmente hay una buena razón por la que todos hacen algo de una manera particular.

Puedes tener tantas clases como quieras así

```
public class Fun {
    Fun() {
        System.out.println("Fun constructor");
    }
    void fun() {
        System.out.println("Fun mathod");
    }
    public static void main(String[] args) {
        Fun fu = new Fun();
        fu.fun();
        Fen fe = new Fen();
        fe.fen();
        Fin fi = new Fin();
        fi.fin();
        Fon fo = new Fon();
        fo.fon();
        Fan fa = new Fan();
```

```
        fa.fan();
        fa.run();
    }
}

class Fen {
    Fen() {
        System.out.println("fen construuctor");

    }
    void fen() {
        System.out.println("Fen method");
    }
}

class Fin {
    void fin() {
        System.out.println("Fin method");
    }
}

class Fon {
    void fon() {
        System.out.println("Fon method");
    }
}

class Fan {
    void fan() {
        System.out.println("Fan method");
    }
    public void run() {
        System.out.println("run");
    }
}
```

Nota: *En un archivo solo puede existir una clase publica el resto de las clases no podrán usar el modificador public y la clase publica debe tener el mismo nombre que el archivo .java que la contiene*.

Interfaces asuntos avanzados

Propósito de una interfaz

Una interfaz proporciona una manera para que una persona desarrolle un código que utiliza el código de otra persona, sin tener acceso a la implementación subyacente de la otra persona. Las interfaces pueden facilitar el desarrollo rápido de aplicaciones al permitir que los equipos de desarrollo creen aplicaciones en paralelo, en lugar de depender directamente entre sí.

Por ejemplo, dos equipos pueden trabajar juntos para desarrollar una interfaz estándar de una página al inicio de un proyecto. Un equipo desarrolla un código que utiliza las interfaces mientras que

otro equipo desarrolla código que implementa la interfaz. Los equipos de desarrollo pueden combinar sus implementaciones hacia el final del proyecto, y siempre que ambos equipos desarrollen con la misma interfaz, serán compatibles. Por supuesto, aún será necesario realizar pruebas para asegurarse de que la clase que implementa la interfaz se comporte como se espera.

Herencia

```
public interface Fly {
	public int getWingSpan() throws Exception;
	public static final int MAX_SPEED = 100;
	public default void land() {
		System.out.println("Animal is landing");
	}
	public static double calculateSpeed(float distance, double time) {
		return distance/time;
	}
}

public class Eagle implements Fly {
	public int getWingSpan() { return 15; }
	public void land() {
		System.out.println("Eagle is diving fast");
	}
}
```

En este ejemplo, el primer método de la interfaz, **getWingSpan()**, declara una excepción en la interfaz. Esto no requiere que la excepción se declare en el método anulado en la clase Eagle.

MAX_SPEED, es una variable estática constante disponible en cualquier lugar dentro de nuestra aplicación.

El siguiente método, **land()**, es un método **default** que se ha invalidado opcionalmente en la clase **Eagle**. Finalmente, el método **calculaSpeed()** es un miembro estático y, como **MAX_SPEED**, está disponible sin una instancia de la interfaz.

Una interfaz puede extender otra, y al hacerlo, hereda todos los métodos abstractos.

```
public interface Walk {
	boolean isQuadruped();
	abstract double getMaxSpeed();
}

public interface Run extends Walk {
	public abstract boolean canHuntWhileRunning();
	abstract double getMaxSpeed();
}

public class Lion implements Run {
	public boolean isQuadruped()		{ return true; }
	public boolean canHuntWhileRunning()	{ return true; }
```

```
      public double getMaxSpeed()              { return 100; }
}
```

En este ejemplo, la Run extiende de Walk y hereda todos los métodos abstractos de la interfaz principal. Tenga en cuenta que los modificadores utilizados en los métodos i**sQuadruped()**, **getMaxSpeed()** y **canHuntWhileRunning()** son diferentes entre las definiciones de clase y de interfaz, como public y abstract. El compilador agrega automáticamente **public** a todos los métodos de interfaz y se abstrae a todos los métodos no estáticos y no predeterminados, si el desarrollador no los proporciona. Por el contrario, la clase que implementa la interfaz debe proporcionar los modificadores adecuados. Por ejemplo, el código no compilaría si **getMaxSpeed()** no se marcó como público en la clase Lion.

Dado que la clase **Lion** implementa **Run**, y **Run** extiende **Walk**, la clase **Lion** debe proporcionar implementaciones concretas de todos los métodos abstractos heredados. Como se muestra en este ejemplo con **getMaxSpeed()**, las definiciones de los métodos de interfaz pueden duplicarse en una interfaz secundaria sin problemas.

Recuerde que una interfaz no puede extender una clase, ni una clase puede extender una interfaz. Por estas razones, ninguna de las siguientes definiciones que usen nuestra interfaz Walk y clase Lion anteriores se compilarán:

```
public interface Sleep extends Lion      {} // DOES NOT COMPILE
public class Tiger extends Walk          {} // DOES NOT COMPILE (No compila
esta probado)
public class Leopard extends Sleep       {} // COMPILE
```

En la primera definición, la interfaz Sleep no puede extender Lion, ya que Lion es una clase. Del mismo modo, la clase Tiger no puede extender la interfaz Walk.

Las interfaces también sirven para proporcionar soporte limitado para herencia múltiple dentro del lenguaje Java, ya que una clase puede implementar múltiples interfaces, como en el siguiente ejemplo:

```
public interface Swim                    { }
public interface Hop                     { }
public class Frog implements Swim, Hop   { }
```

Herencia con método default.

```
public interface Animal{
    default void caminar(){
        System.out.println("caminado");
    }
}

public interface Tortuga{
    default void caminar(){
```

```
        System.out.println("caminando lento");
    }
}

public class TortugaNinja implements Animal,Tortuga{ }

//Esta clase no compila por que ambas interfaces tienen métodos default con
//el mismo nombre y misma firma de parámetros. Si no fueran default los métodos
//heredados si compilarian.


//Ejemplo 2
public interface Animal{
    default void caminar(){
        System.out.println("caminado");
    }
}
public interface Tortuga{
    void caminar();
}

public class TortugaNinja implements Animal,Tortuga{
    @Override
    public void caminar() {
        System.out.println("Quiero pizza!!");
    }
}
//En este ejemplo en una clase no es default y compila


//Ejemplo 3
public interface Animal{
    default void caminar(){
        System.out.println("caminado");
    }
}
public interface Tortuga{
    void caminar();
}

public class TortugaNinja implements Animal,Tortuga{
    @Override
    public void caminar() {
        Animal.super.caminar();
    }
}
//En este ejemplo en una clase no es default y compila, se sobreEscribio el
//método y se llama con super al método default.


//Ejemplo 4
public interface Animal{
    void caminar();
}
public interface Tortuga{
    void caminar();
}

public class TorugaNinja implements Animal,Tortuga{
    @Override
    public void caminar() {
```

```
        System.out.println("Quiero pizza!!");
    }
}
//Ningún método es default, no hay conflicto con repetir el nombre y la firma de
//parámetros.
```

Introducción a la Programación Funcional

Java define una interfaz funcional como una interfaz que contiene un único método abstracto. Las interfaces funcionales se utilizan como base para las expresiones lambda en la programación funcional.

Una expresión lambda es un bloque de código que se pasa, como un método anónimo. Dado que las expresiones lambda y la programación funcional son una piedra angular de Java 8.

Declaración de una interface funcional

```
@FunctionalInterface
public interface Sprint {
      public void sprint(Animal animal);
}

public class Tiger implements Sprint {
      public void sprint(Animal animal) {
            System.out.println("Animal is sprinting fast! "+animal.toString());
      }
}
```

Ejemplo de Interfaces Funcionales.

```
public interface Run extends Sprint {}

public interface SprintFaster extends Sprint {
      public void sprint(Animal animal);
}

public interface Skip extends Sprint {
      public default int getHopCount(Kangaroo kangaroo) { return 10; }
      public static void skip(int speed) {}
}
```

Las tres son interfaces funcionales válidas! La primera interfaz, **Run**, no define nuevos métodos, pero como se extiende **Sprint**, que define un solo método abstracto, también es una interfaz funcional. La segunda interfaz, **SprintFaster**, extiende **Sprint** y define un método abstracto, pero esto es una anulación del método **sprint()** principal; por lo tanto, la interfaz resultante tiene un solo método abstracto y se considera una interfaz funcional.

La tercera interfaz, **Skip** , extiende **Sprint** y define un método estático y un método predeterminado, cada uno con una implementación. Dado que ninguno de estos métodos es abstracto, la interfaz resultante tiene solo un método abstracto y es una interfaz funcional.

Ahora que ha visto algunas variaciones de interfaces funcionales válidas, veamos algunas que no son válidas utilizando nuestra definición de interfaz funcional **Sprint** anterior:

Ejemplo de interfaces no funcionales.

```
public interface Walk {}

public interface Dance extends Sprint {
      public void dance(Animal animal);
}

public interface Crawl {
      public void crawl();
      public int getCount();
}
```

Aunque las tres interfaces se compilarán, ninguna de ellas se considera una interfaz funcional. La interfaz **Walk** no extiende ninguna clase de interfaz funcional ni define ningún método, por lo que no es una interfaz funcional. **Dance** amplía a **Sprint**, que ya incluye un solo método abstracto, lo que hace que el total sea dos métodos abstractos; Por lo tanto, **Dance** no es una interfaz funcional. Finalmente, el **Crawl** define dos métodos abstractos; por lo tanto no puede ser una interfaz funcional.

En estos ejemplos, la aplicación de la anotación @FunctionalInterface en cualquiera de estas interfaces resultaría en un error del compilador, al igual que intentar usarlas implícitamente como interfaces funcionales en una expresión lambda.

```
@FunctionalInterface
public interface Walk {}      // No Compila
```

Implementando Interfaces Funcionales con Lambdas

Ahora que hemos definido una interfaz funcional, le mostraremos cómo implementarlas utilizando expresiones lambdas. Como dijimos anteriormente, una expresión lambda es un bloque de código que se pasa, como un método anónimo. Comencemos con una interfaz funcional simple de CheckTrait, que tiene un solo método test (), que toma como entrada una instancia de una clase de animales. Las definiciones de la clase y la interfaz funcional son las siguientes:

```
public class Animal {
      private String species;
      private boolean canHop;
      private boolean canSwim;
      public Animal(String speciesName, boolean hopper, boolean swimmer) {
            species = speciesName;
            canHop = hopper;
            canSwim = swimmer;
      }
      public boolean canHop()  { return canHop;  }   //saltar
      public boolean canSwim() { return canSwim; }
      public String toString() { return species; }
}

@FunctionalInterface
```

```
public interface CheckTrait {  //CheckearRasgo
	public boolean test(Animal a);
}
```

Ahora que hemos definido una estructura, hagamos algo con ella. El siguiente programa simple utiliza una expresión lambda para determinar si algunos animales de muestra coinciden con los criterios especificados:

```
public class FindMatchingAnimals {
	private static void print(Animal animal, CheckTrait trait) {
		if(trait.test(animal))
			System.out.println(animal);
	}

	public static void main(String[] args) {
		print(new Animal("fish", false, true), a -> a.canHop());
		print(new Animal("kangaroo", true, false), a -> a.canHop());
	}
}
```

Para propósitos ilustrativos, la expresión lambda elegida para este programa es bastante simple:

```
a -> a.canHop();
```

Esta expresión significa que Java debe llamar a un método con un parámetro Animal que devuelve un valor booleano que es el resultado de **a.canHop()**. Sabemos todo esto porque escribimos el código. Pero, ¿cómo sabe Java? Java se basa en el contexto cuando determina lo que significan las expresiones lambda. Estamos pasando esta lambda como el segundo parámetro del método **print()**. Ese método espera un **CheckTrait** como el segundo parámetro. Ya que estamos pasando un lambda en su lugar, Java trata a **CheckTrait** como una interfaz funcional y trata de asignarlo al método abstracto único:

```
boolean test(Animal a);
```

Como el método de esta interfaz toma un Animal como parámetro, significa que el parámetro lambda tiene que ser un Animal. Y como el método de esa interfaz devuelve un valor booleano, sabemos que la lambda retorna un valor booleano.

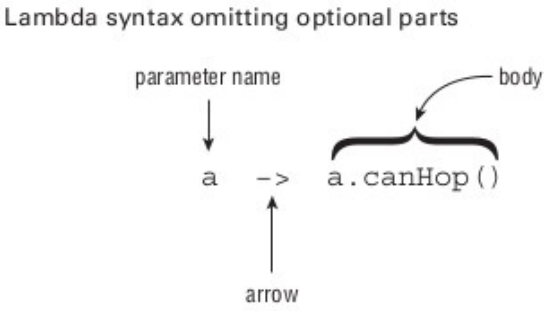
Detalles de la sintaxis Lambda

La sintaxis de las expresiones lambda es complicada porque muchas partes son opcionales. Estas dos líneas son equivalentes y hacen exactamente lo mismo:

```
a -> a.canHop()
(Animal a) -> { return a.canHop(); }
```

Veamos lo que está pasando aquí. El lado izquierdo del operador de flecha -> indica los parámetros de entrada para la expresión lambda. Puede ser consumido por una interfaz funcional cuyo método abstracto tiene el mismo número de parámetros y tipos de datos compatibles. El lado derecho se conoce como el cuerpo de la expresión lambda. Puede ser consumido por una interfaz funcional cuyo método abstracto devuelve un tipo de datos compatible.

Como la sintaxis de estas dos expresiones es un poco diferente, veamoslas más de cerca.

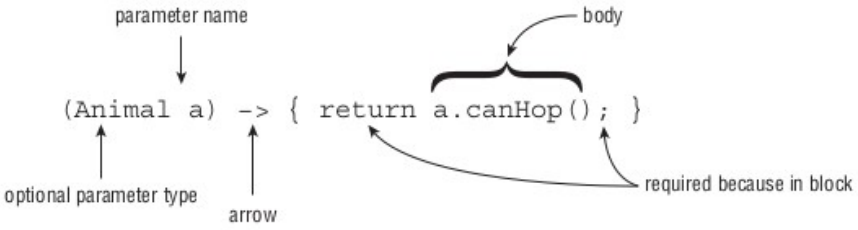


■ Especificamos un solo parámetro con el nombre a.

■ El operador de flecha → separa el parámetro del cuerpo.

■ El cuerpo llama a un solo método y devuelve el resultado de ese método.

FIGURE 2.2 Lambda syntax, including optional parts



El segundo ejemplo también tiene tres partes, como se muestra en la Figura; es simplemente más detallado:

- Especificamos un único parámetro con el nombre a y declaramos que el tipo es Animal, envolviendo los parámetros de entrada entre paréntesis ().

- El operador de flecha -> separa el parámetro del cuerpo.

- El cuerpo tiene una o más líneas de código, incluidas las llaves {}, un punto y coma y una declaración de retorno.

```
() -> new Duck()
d -> { return d.quack(); }

(Duck d) -> d.quack()
(Animal a, Duck d) -> d.quack()
```

Ejemplos de expresiones invalidas

```
Duck d -> d.quack()                 // DOES NOT COMPILE
a,d -> d.quack()                    // DOES NOT COMPILE
Animal a, Duck d -> d.quack()       // DOES NOT COMPILE
//En los 3 casos anteriores faltan los paréntesis ().
//los paréntesis se pueden omitir solo si hay exactamente un parámetro y no se
//especifica el tipo de datos.
```

pudimos omitir las llaves {}, punto y coma ;, y declaración de retorno, porque este es un atajo especial que Java permite para cuerpos lambda de una sola línea. Este atajo especial no funciona cuando tiene dos o más declaraciones. Al menos esto es consistente con el uso de {} para crear bloques de código en otro lugar en Java. Cuando se utiliza {} en el cuerpo de la expresión lambda, debe usar la instrucción return si el método de interfaz funcional que implementa lambda devuelve un valor. Alternativamente, una declaración de retorno es opcional cuando el tipo de retorno del método es nulo.

Veamos algunos ejemplos más:

```
//Retorno de parámetros
() -> true                                  // 0 parámetros
a -> {return a.startsWith("test");}         // 1 parámetro
(String a) -> a.startsWith("test")          // 1 parámetro
(int x) -> {}                               // 1 parámetro
(int y) -> {return;}                        // 1 parámetro
```

El primer ejemplo no toma argumentos y siempre devuelve verdadero. El segundo y el tercer ejemplo toman un solo valor de String, usando una sintaxis diferente para lograr lo mismo.

```
(a, b) -> a.startsWith("test")                  // 2 parameters
(String a, String b) -> a.startsWith("test")    // 2 parameters
```

```
a, b -> a.startsWith("test")                    // DOES NOT COMPILE
c -> return 10;                                 // DOES NOT COMPILE
a -> { return a.startsWith("test") }            // DOES NOT COMPILE

(int y, z) -> {int x=1; return y+10; }          // DOES NOT COMPILE
(String s, z) -> { return s.length()+z; }       // DOES NOT COMPILE
(a, Animal b, c) -> a.getName()                 // DOES NOT COMPILE

(a, b) -> { int a = 0; return 5;}               // DOES NOT COMPILE
```

Usando el Predicado en la interface

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

```
public class FindMatchingAnimals {
      private static void print(Animal animal, Predicate<Animal> trait) {
            if(trait.test(animal))
            System.out.println(animal);
      }
      public static void main(String[] args) {
            print(new Animal("fish", false, true), a -> a.canHop());
            print(new Animal("kangaroo", true, false), a -> a.canHop());
      }
}
```

Colecciones concurrentes

Ejemplo

Las colecciones concurrentes son una generalización de las colecciones seguras para subprocesos, que permiten un uso más amplio en un entorno concurrente.

Si bien las colecciones seguras para subprocesos tienen la adición o eliminación segura de elementos de varios subprocesos, no necesariamente tienen una iteración segura en el mismo contexto (es posible que uno no pueda recorrer la colección en un subproceso en una secuencia, mientras que otro lo modifica agregando quitando elementos).

Aquí es donde se utilizan las colecciones concurrentes.

Como la iteración suele ser la implementación básica de varios métodos masivos en colecciones, como `addAll` , `removeAll` , o también la copia de colecciones (a través de un constructor u otros medios), clasificación, ... el caso de uso de colecciones concurrentes es en realidad bastante grande.

Por ejemplo, Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` es una implementación de `Lis` t segura para subprocesos, su javadoc afirma:

> El método del iterador de estilo "instantánea" utiliza una referencia al estado de la matriz en el punto en que se creó el iterador. Esta matriz nunca cambia durante la vida útil del iterador, por lo que la interferencia es imposible y se garantiza que el iterador no lanzará ConcurrentModificationException.

Por lo tanto, el siguiente código es seguro:

```
public class ThreadSafeAndConcurrent {

public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

public static void main(String[] args) throws InterruptedException {
    Thread modifier = new Thread(new ModifierRunnable());
    Thread iterator = new Thread(new IteratorRunnable());
```

```
    modifier.start();
    iterator.start();
    modifier.join();
    iterator.join();
}

public static final class ModifierRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 50000; i++) {
                LIST.add(i);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static final class IteratorRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                long total = 0;
                for(Integer inList : LIST) {
                    total += inList;
                }
                System.out.println(total);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

Otra colección concurrente con respecto a la iteración es `ConcurrentLinkedQueue` , que establece:

> Los iteradores son débilmente consistentes, devolviendo elementos que reflejan el estado de la cola en algún momento en o desde la creación del iterador. No lanzan java.util.ConcurrentModificationException, y pueden proceder simultáneamente con otras operaciones. Los elementos contenidos en la cola desde la creación del iterador se devolverán exactamente una vez.

Uno debe revisar los javadocs para ver si una colección es concurrente o no. Los atributos del iterador devueltos por el método `iterator()` ("falla rápidamente", "débilmente consistente", ...) es el atributo más importante que debe buscarse.

Hilo seguro pero ejemplos no concurrentes

En el código anterior, cambiando la declaración `LIST` a
`public class ThreadSafeAndConcurrent {`

```
public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

public static void main(String[] args) throws InterruptedException {
    Thread modifier = new Thread(new ModifierRunnable());
    Thread iterator = new Thread(new IteratorRunnable());
    modifier.start();
    iterator.start();
    modifier.join();
    iterator.join();
}

public static final class ModifierRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 50000; i++) {
                LIST.add(i);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static final class IteratorRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                long total = 0;
                for(Integer inList : LIST) {
                    total += inList;
                }
                System.out.println(total);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

Podría (y estadísticamente lo hará en la mayoría de las arquitecturas de CPU / núcleo con múltiples CPU) dar lugar a excepciones.

Las colecciones sincronizadas de los métodos de utilidad de `Collections` son seguras para subprocesos para la adición / eliminación de elementos, pero no la iteración (a menos que la colección subyacente que se le pasa ya lo sea).

TABLE 7.9 Concurrent collection classes

Class Name	Java Collections Framework Interface	Elements Ordered?	Sorted?	Blocking?
`ConcurrentHashMap`	`ConcurrentMap`	No	No	No
`ConcurrentLinkedDeque`	`Deque`	Yes	No	No
`ConcurrentLinkedQueue`	`Queue`	Yes	No	No

`ConcurrentSkipListMap`	`ConcurrentMap`
	`SortedMap`
	`NavigableMap`
		Yes	Yes	No

`ConcurrentSkipListSet`	`SortedSet` `NavigableSet`	Yes	Yes	No
`CopyOnWriteArrayList`	`List`	Yes	No	No
`CopyOnWriteArraySet`	`Set`	No	No	No
`LinkedBlockingDeque`	`BlockingQueue` `BlockingDeque`	Yes	No	Yes
`LinkedBlockingQueue`	`BlockingQueue`	Yes	No	Yes

Ejemplo de uso de colecciones concurrentes

```
Map<String,Integer> map = new ConcurrentHashMap<>();
map.put("zebra", 52);
map.put("elephant", 10);
System.out.println(map.get("elephant"));
Queue<Integer> queue = new ConcurrentLinkedQueue<>();
queue.offer(31);
System.out.println(queue.peek());
System.out.println(queue.poll());
Deque<Integer> deque = new ConcurrentLinkedDeque<>();
deque.offer(10);
deque.push(4);
System.out.println(deque.peek());
System.out.println(deque.pop());
```