Unidad

4

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 7

Genéricos y Colecciones

Genéricos y Colecciones

En Este Capítulo

- Genéricos
- El operador diamante
- Limitación en los parámetros de tipo
- Invariancia
- Covarianza
- Subtipos
- Parámetros desconocidos declarados con caracteres comodines
- La API de Colecciones
- Set
- List
- Iteradores
- Mapas
- Ordenamiento de colecciones
- Genéricos y colecciones
- La API de colecciones genéricas
- Análisis de los parámetros de tipo
- Parámetros comodines en las colecciones
- Genéricos: refactorización de código no genérico existente

Genéricos

La producción de software siempre busca la disminución de errores provocados por descuidos al programar. Uno de los problemas más recurrente es el de la conversión de tipos de objetos en tiempo de ejecución.

Por lo visto hasta el momento, convertir tipos es el arma más importante al utilizar polimorfismo (es más, se basa en ello). Sin embargo esto deja abierta una puerta peligrosa, sobre todo cuando se utiliza Object, porque cualquier tipo puede ser convertido a esta clase.

Uno de los temas más importantes al aprender polimorfismo es no olvidar que cada objeto conserva su identidad a pesar que la referencia que lo señala haya sido convertida al tipo de una superclase y, por lo tanto, esa conversión puede ser revertida. Esta herramienta se convierte en un arma de doble filo cuando tratando de recuperar el tipo original que fue convertido y se equivoca la conversión a la inversa, originando errores en tiempo de ejecución.

Esto impone como meta tratar de poseer una herramienta capaz de detectar este tipo de errores al menos en tiempo de compilación, para que estos sean más fáciles de corregir. Esta es la finalidad de los genéricos.

Los genéricos restringen al código para que los tipos se conviertan en tiempo de compilación en lugar de tiempo de ejecución. Dado este hecho, si un tipo no es “convertible” se detecta en tiempo de compilación y no de ejecución.

A primera vista se puede pensar que esto atenta contra el polimorfismo, sin embargo es todo lo contrario, asegura un mejor diseño de software. La razón es que aquello que es diseñado para ser polimórfico se utilice como tal y aquello que no, se especifique con que tipo se quiere utilizar.

Por ejemplo, si se quiere realizar una clase de pila que maneja enteros un primer diseño puede ser intentar manejar esto con un vector.

Ejemplo

```
package pila.inicio;
import pila.ExcepcionPila;

public class Pila {
	private int vec[] = new int[10];
	private int indice = 0;

	public void agregar(int valor) throws ExcepcionPila {
		if (indice == 9)
			throw new ExcepcionPila("Pila llena");
		vec[indice] = valor;
		indice++;
	}

	public int sacar() throws ExcepcionPila {
		if (indice == 0)
```

```
                throw new ExcepcionPila("Pila vacía");
            indice--;
            return vec[indice];
        }
}
```

El primer problema que se puede apreciar realmente es que el código no refleja una **pila** sino una **pila de enteros**, lo cual es bastante limitado.

En un primer intento de generalizar este código, se puede intentar un nuevo diseño de la clase basada en objetos para generalizar la clase Pila. Esta puede además manejar los tipos primitivos por medio del autoboxing, aunque esto signifique una pérdida de rendimiento considerable en cada transformación.

Ejemplo

```
package pila.objetos;

import pila.ExcepcionPila;

public class Pila {
        private Object vec[] = new Object[10];
        private int indice = 0;

        public void agregar(Object valor) throws ExcepcionPila {
            if (indice == 9)
                throw new ExcepcionPila("Pila llena");
            vec[indice] = valor;
            indice++;
        }

        public Object sacar() throws ExcepcionPila {
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            indice--;
            return vec[indice];
        }
}
```

Una posible clase para utilizar este código se presenta a continuación. Notar que no da ningún tipo de error de compilación.

Ejemplo

```
package pila.objetos.version1;
import pila.ExcepcionPila;
import pila.objetos.Pila;

public class UsaPila {
        public static void main(String[] args) {
            Pila p = new Pila();

            try {
```

```
                p.agregar("Hola");
                p.agregar(new Integer(8));
                p.agregar(new Float(8.8));
                p.agregar(new Integer(9));

                double total = (double) p.sacar() +
                            (double)  p.sacar() +
                            (double)  p.sacar() +
                            (double)  p.sacar();
                System.out.println(total);

            } catch (ExcepcionPila e) {
                e.printStackTrace();
            }
        }
}
```

Cuando se intenta ejecutar el programa se recibe el siguiente mensaje

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.Double
        at pila.objetos.version1.UsaPila.main(UsaPila.java:16)
```

Se asume que error cometido es intentar convertir un objeto a un tipo base, entonces se utiliza una clase de envoltorio como se muestra a continuación.

Ejemplo

```
package pila.objetos.version2;

import pila.ExcepcionPila;
import pila.objetos.Pila;

public class UsaPila {
        public static void main(String[] args) {
            Pila p = new Pila();

            try {
                p.agregar("Hola");
                p.agregar(new Integer(8));
                p.agregar(new Float(8.8));
                p.agregar(new Integer(9));

                double total = ((Double) p.sacar()) +
                            ((Double) p.sacar()) +
                            ((Double) p.sacar()) +
                            ((Double) p.sacar());
                System.out.println(total);

            } catch (ExcepcionPila e) {
                e.printStackTrace();
            }
        }
}
```

Sin embargo al ejecutar el programa se obtiene el mismo resultado.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.Double
	at pila.objetos.version2.UsaPila.main(UsaPila.java:16)
```

En este caso la respuesta parece obvia. Si bien tanto Integer como Double heredan de Number están en cadenas de herencia diferentes. Por lo tanto se opta por la siguiente solución.

Ejemplo

```
package pila.objetos.version3;

import pila.ExcepcionPila;
import pila.objetos.Pila;

public class UsaPila {
	public static void main(String[] args) {
		Pila p = new Pila();

		try {
			p.agregar("Hola");
			p.agregar(new Integer(8));
			p.agregar(new Float(8.8));
			p.agregar(new Integer(9));

			double total = ((Number) p.sacar()).doubleValue() +
					((Number) p.sacar()).doubleValue() +
					((Number) p.sacar()).doubleValue() +
					((Number) p.sacar()).doubleValue();
			System.out.println(total);

		} catch (ExcepcionPila e) {
			e.printStackTrace();
		}
	}
}
```

Con lo cual se debería de solucionar el problema. Sin embargo al ejecutar el programa, si bien cambia el error, se obtiene nuevamente un error en tiempo de ejecución.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
be cast to java.lang.Number
	at pila.objetos.version3.UsaPila.main(UsaPila.java:19)
```

Este último mensaje se hace evidente al examinar el código de cerca en este programa simple, pero puede ser muy difícil de encontrar si el código fuera más extenso. Hasta ahora la pila había recibido valores numéricos y si bien las transformaciones de datos se forzaban respecto de sus valores originales en la suma aritmética, se podía realizar. Sin embargo el último elemento agregado a la pila ***es una cadena a la cual se trata de convertir en un número***. Con el propósito de lograr el resultado se elimina el último valor de la cuenta aritmética.

Ejemplo

```
package pila.objetos.version4;

import pila.ExcepcionPila;
import pila.objetos.Pila;

public class UsaPila {
	public static void main(String[] args) {
		Pila p = new Pila();

		try {
			p.agregar("Hola");
			p.agregar(new Integer(8));
			p.agregar(new Float(8.8));
			p.agregar(new Integer(9));

			double total = ((Number) p.sacar()).doubleValue() +
					((Number) p.sacar()).doubleValue() +
					((Number) p.sacar()).doubleValue();
			System.out.println(total);

		} catch (ExcepcionPila e) {
			e.printStackTrace();
		}
	}
}
```

Y por fin se obtiene el siguiente resultado *que al menos es un valor numérico*.

```
25.800000190734863
```

Esto tiene varios inconvenientes. Primero, los errores de cálculo provienen de los algoritmos de punto flotante que no son exactos, y para poder realizar la cuenta completa se fuerza toda la operación al valor más grande posible. Segundo se hicieron cálculos con los últimos tres elementos de la pila, *pero aún queda un elemento en la pila que es de otro tipo, es más, ni siquiera es un número*.

El verdadero problema es en realidad que la pila no puede diferenciar los tipos de datos utilizados, con lo cual no tiene los tipos asegurados (type safe) por estar basada en Object que es superclase de todas las clases de Java y realizar una serie de conversiones en tiempo de ejecución. Por lo tanto, cuando se produzcan errores se detectarán indefectiblemente en tiempo de ejecución también.

A partir de Java 1.5 se agregaron una serie de extensiones al lenguaje Java y una de ellas fueron los genéricos. Estos son similares a los *templates* de C++. Los genéricos permiten abstraerse de los tipos de datos. Los ejemplos más comunes de esto son las clases contenedoras, como por ejemplo la cadena de herencia Collection (que se explicará posteriormente).

Si se quisiera diseñar una clase que pudiera manejar coordenadas sin importar como fueran los valores que almacena, enteros o puntos flotantes, se puede utilizar a los genéricos para que el

compilador identifique el tipo de datos a utilizar en tiempo de compilación y cualquier intento de operación que sea inadecuada para el tipo lo reporte como un error en tiempo de compilación.

Los genéricos exigen que se indique el parámetro que define el tipo a utilizar en la declaración de la clase entre los símbolos de mayor y menor que, y ese parámetro define en si mismo un ***tipo genérico***, que como cualquier tipo se puede utilizar para declarar variables, parámetros o valores de retorno de los métodos. La clase coordenada con genéricos quedaría como la siguiente.

Ejemplo

```
package pila.genericos;

public class Coordenada<T> {
	private T x;
	private T y;
	private T z;

	public Coordenada(T x, T y, T z) {
		super();
		this.x = x;
		this.y = y;
		this.z = z;
	}

	public T getX() {
		return x;
	}

	public void setX(T x) {
		this.x = x;
	}

	public T getY() {
		return y;
	}

	public void setY(T y) {
		this.y = y;
	}

	public T getZ() {
		return z;
	}

	public void setZ(T z) {
		this.z = z;
	}

}
```

Para utilizar la clase, por ejemplo con objetos del tipo Integer (aunque puede ser con cualquier tipo), sólo se debe indicar en la creación del objeto cual es el tipo de parámetro con el cual el compilador debe utilizar las declaraciones dentro de la clase.

Ejemplo

```
package pila.genericos;

public class UsaCoordenada {
	public static void main(String[] args) {
		Coordenada<Integer> c = new Coordenada<Integer>(2,3,4);
		System.out.println("X: " + c.getX() + ", y: "
				+ c.getY() + ", Z: " + c.getZ());
		int suma = c.getX() + c.getY() + c.getZ();
		System.out.println("La suma de las coordenadas es: " + suma);
	}
}
```

El resultado obtenido es

```
X: 2, y: 3, Z: 4
La suma de las coordenadas es: 9
```

Como se puede apreciar, se solucionan varios problemas, como todos aquellos que devienen de convertir tipos de datos. Por eso se puede afirmar que el uso de genéricos **asegura el tipo, pero no las operaciones que se realizan con estos**. Por ejemplo si se define un tipo String, la clase Coordenada lo acepta como parámetro válido, por más que no tenga sentido un objeto de ese tipo. Posteriormente se verá como tratar a parámetros para que sean de una determinada cadena de herencia.

Genéricos y tipos base

Una pregunta válida es ¿se pueden utilizar genéricos con los tipo base como `int` o `long`? La respuesta es un categórico **NO**.

Sin embargo, la duda persiste porque el lenguaje posee la capacidad de autoboxing, ¿no debería manejar automáticamente su capacidad de convertir los tipos base? Nuevamente la respuesta es **NO**.

Las razones de estas respuestas no son simples y residen mucho en la filosofía de diseño del lenguaje. Primero, los tipos base son valores numéricos, con lo cual no hay forma de verificar sus tipos *salvo por su tamaño*, lo cual implicaría un crecimiento interno notable de reglas de validación para el compilador y sus respectivas consecuencias en el rendimiento. Además, incrementaría el número de reglas a aplicar en todo lo que intervenga un tipo base, como promociones automáticas o conversiones de tipo. Por sus consecuencias es evidente que no se incluyó por estos u otros motivos. Se puede definir entonces que

Los genéricos en Java sólo funcionan con objetos como parámetros

Por otra parte, esto debe quedar en claro porque los parámetros genéricos no se comportan como tipos base en las declaraciones. Por ejemplo, si se intenta declarar un vector de un tipo genérico el compilador dará un mensaje de error cuando se intente crear las referencias mediante `new`. La razón es simple, se puede crear una referencia a un tipo de objeto sin necesidad de conocer su estructura, lo cual no es válido en un tipo base.

Retomando el ejemplo de la pila, rápida y descuidadamente se pretende declarar un vector para manejarla como se muestra a continuación.

Ejemplo

```
import pila.ExcepcionPila;

public class Pila <T>{
	private T vec[] = new T[10]; // ERROR
	private int indice = 0;

	public void agregar(T valor) throws ExcepcionPila {
		if (indice == 9)
			throw new ExcepcionPila("Pila llena");
		vec[indice] = valor;
		indice++;
	}

	public T sacar() throws ExcepcionPila {
		if (indice == 0)
			throw new ExcepcionPila("Pila vacía");
		indice--;
		return vec[indice];
	}
}
```

El error en tiempo de compilación es similar al siguiente:

> Cannot create a generic array of T

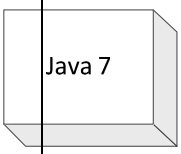
Una solución de compromiso, que tiene aún muchas ventajas respecto al diseño general es la siguiente.

Ejemplo

```
package pila.genericos.version2;

import pila.ExcepcionPila;

public class Pila <T>{
	private Object vec[] = new Object[10];
	private int indice = 0;

	public void agregar(T valor) throws ExcepcionPila {
		if (indice == 9)
			throw new ExcepcionPila("Pila llena");
		vec[indice] = valor;
		indice++;
	}

	public T sacar() throws ExcepcionPila {
		if (indice == 0)
			throw new ExcepcionPila("Pila vacía");
		indice--;
		return (T) vec[indice];
```

```
	}
}
```

Esta solución mantiene controlada las conversiones de tipo para que no se propague fuera de la clase. Sin embargo es una solución incompleta porque se siguen realizando conversiones. El problema radica en el vector y para evitarlo sería necesaria una clase que pudiera manejar en su interior tipos genéricos y funcionase como un vector, una lista, pila, cola etc… Estas clases existen como se verá posteriormente en la denominada API de colecciones.



El operador diamante

A partir de la versión 1.7, se puede declarar un parámetro genérico en un tipo y no repetir la declaración “nuevamente” a la derecha del operador **new**. Esto, aunque parece una tontería, reduce notablemente la cantidad de código escrito.

Ejemplo

```
A<Double> a2 = new A<>();
```

Limitación en los parámetros de tipo

Existen situaciones en las cuales se desea limitar los parámetros que se puedan asignar a un tipo cuando se declara un objeto. Para esto, Java provee un mecanismo a través de las cadenas de herencia que permite establecer un límite de manera que un parámetro se pueda asignar a un tipo sólo si pertenece a la cadena de herencia declarada.

Para declarar el limite para un parámetro de tipo , se debe escribir el nombre de éste, seguido de la palabra clave **extends**, seguida de su límite superior (superclase de la cadena o sub cadena de herencia), que en el siguiente ejemplo es Number. Nótese que, en este contexto, **extends** se utiliza en un sentido general para significar "extender" (como en las clases) o "implementar" (como en las interfaces).

Ejemplo

```
package limites.metodos;

public class Metodo<T> {

	private T t;

	public void agregar(T t) {
		this.t = t;
	}

	public T obtener() {
		return t;
	}

	public <U extends Number> void inspeccionar(U u) {
		System.out.println("T: " + t.getClass().getName());
```

```
            System.out.println("U: " + u.getClass().getName());
      }

      public static void main(String[] args) {
            Metodo<Integer> enteroA = new Metodo<Integer>();
            enteroA.agregar(new Integer(10));
            // Usar autoboxing
            enteroA.inspeccionar(3);
            // La próxima línea da error porque un
            // String no es del tipo Number
//          enteroA.inspeccionar("una cadena");
      }
}
```

La salida obtenida del programa es

```
T: java.lang.Integer
U: java.lang.Integer
```

Si se quitase el comentario de la línea que la documentación indica que va a dar un error, el programa no compilaría. Esto se debe a que String no pertenece a la cadena o sub cadena de herencia de Number, con lo cual no es un parámetro posible a asignar para los tipos que admite como argumento el método. El error provocado es similar al siguiente.

Bound mismatch: The generic method inspeccionar(U) of type A<T> is not applicable for the arguments (String). The inferred type String is not a valid substitute for the bounded parameter <U extends Number>

Invariancia

El uso de genéricos asegura el tipo. Es más, su principal objetivo es asegurar el tipo utilizado por un determinado objeto. Pero, como se puede asignar cualquier tipo al parámetro genérico, ¿qué pasa si se asigna a un tipo a un parámetro cuando se crea un objeto y se trata de asignar a una referencia de ese tipo, que se creó con un parámetro diferente?

Es claro que nunca estos objetos pueden ser “exactamente del mismo tipo” porque el compilador asignó diferentes tipos al parámetro genérico y es como si con declaraciones estándar se hubiesen hecho dos clases diferentes. Al mecanismo que asegura el tipo de esta forma se lo llama invariancia y su principal función es asegurar el tipo.

Ejemplo

```
package limites.invariancia;
public class VerificaLimitesParametros {

      public static void main(String[] args) {
            Z<A> z1 = new Z<>();
            Z<B> z2 = new Z<>();
            Z<C> z3 = new Z<>();

            z1 = z2; // ERROR
            z2 = z3; // ERROR
      }
```

```
}
```

Las asignaciones subrayadas comenten exactamente ese error. En el primer caso z1 se creó con un parámetro genérico del tipo A, por lo tanto intentar asignar un objeto que se creó con un parámetro del tipo B es incompatible. El error que se registra es el siguiente:

Type mismatch: cannot convert from Z<B> to Z<A>

El segundo error es similar, lo que cambia son las clases con lo que se intenta hacer el mismo tipo de conversión no admitida entre referencias. El error en este caso es el siguiente:

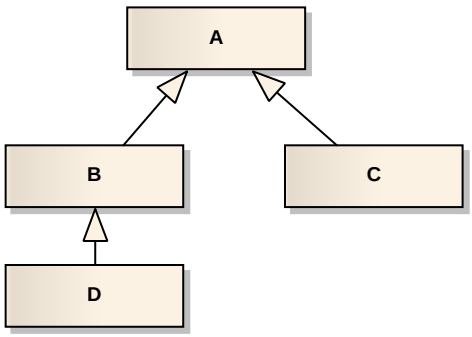
Type mismatch: cannot convert from Z<C> to Z<B>

Covarianza

Se define a la covarianza de la siguiente manera

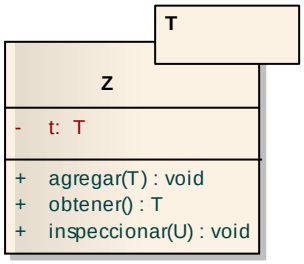
> *Cuando en las asignaciones se preserva el orden natural de los subtipos respecto a sus súper tipos en una asignación (de lo más específico a lo más genérico) la asignación es covariante*

Para ampliar un poco el concepto, se puede suponer la siguiente cadena de herencia.



Si se tratan de establecer los límites en un método de una clase Z para que utilice estas dos cadenas de herencia se puede optar por poner a la clase A como límite superior y ver cómo reaccionaría un programa al intentar utilizar los diferentes tipos de objetos.

La clase Z puede tener definidos a la vez parámetros de tipo, como la clase Método para generalizar el ejemplo expuesto anteriormente



Ejemplo

```
package limites.metodos;

public class Z <T>{
	private T t;

	public void agregar(T t) {
		this.t = t;
	}

	public T obtener() {
		return t;
	}
```

```
	public <U extends A> void inspeccionar(U u) {
		System.out.println("T: " + t.getClass().getName());
		System.out.println("U: " + u.getClass().getName());
	}
}
```

La clase verifica el correcto funcionamiento de la clase Z para la cadena de herencia del gráfico UML.

Ejemplo

```
package limites.metodos;
public class VerificaLimitesParametros {
	public static void main(String[] args) {
		Z<A> z1 = new Z<>();
		Z<B> z2 = new Z<>();
		Z<C> z3 = new Z<>();
		Z<D> z4 = new Z<>();

		z1.agregar(new A());
		z1.inspeccionar(new A());
		z1.inspeccionar(new B());
		z1.inspeccionar(new C());
		z1.inspeccionar(new D());

		// La siguiente línea da Error porque el parámetro
		// del tipo para z2 es B
		//z2.agregar(new A());
		z2.agregar(new B());
		z2.inspeccionar(new A());
		z2.inspeccionar(new B());
		z2.inspeccionar(new C());
		z2.inspeccionar(new D());

		z3.agregar(new C());
		z3.inspeccionar(new A());
		z3.inspeccionar(new B());
		z3.inspeccionar(new C());
		z3.inspeccionar(new D());

		z4.agregar(new D());
		z4.inspeccionar(new A());
		z4.inspeccionar(new B());
		z4.inspeccionar(new C());
		z4.inspeccionar(new D());
	}
}
```

Si se ejecuta el programa se obtiene la siguiente salida.

```
T: limites.metodos.A
U: limites.metodos.A
T: limites.metodos.A
U: limites.metodos.B
T: limites.metodos.A
```

```
U: limites.metodos.C
T: limites.metodos.A
U: limites.metodos.D
T: limites.metodos.B
U: limites.metodos.A
T: limites.metodos.B
U: limites.metodos.B
T: limites.metodos.B
U: limites.metodos.C
T: limites.metodos.B
U: limites.metodos.D
T: limites.metodos.C
U: limites.metodos.A
T: limites.metodos.C
U: limites.metodos.B
T: limites.metodos.C
U: limites.metodos.C
T: limites.metodos.C
U: limites.metodos.D
T: limites.metodos.D
U: limites.metodos.A
T: limites.metodos.D
U: limites.metodos.B
T: limites.metodos.D
U: limites.metodos.C
T: limites.metodos.D
U: limites.metodos.D
```

Notar como el argumento declarado en el método admitió los objetos de las dos cadenas de herencia como parámetros.

Se puede extender el concepto a la declaración de clases para limitar los parámetros que estas reciben. Si se modifica levemente la clase Z, se puede declarar para que sólo admita subtipos de A, por ejemplo:

```
public class Z <T extends A>
```

Con una declaración de este tipo, si se intenta usar la clase Z con un parámetro que no sea una subclase de A, se obtiene un error. Así una declaración como la siguiente:

```
Z<Double> z5 = new Z<>(); // Error
```

Genera un error como el siguiente:

Multiple markers at this line
- Cannot infer type arguments for Z<>
- Bound mismatch: The type Double is not a valid substitute for the bounded parameter <T extends A> of the type Z<T>

Para especificar interfaces adicionales que deben ser implementadas, se debe utilizar el carácter **&**, como en el siguiente ejemplo:

```
<U extends Number & MyInterface>
```

De esta manera, la limitación en el parámetro debe cumplir ambas condiciones.

Ejemplo

```
package limites.metodos.interfaces;

public interface I {
	public void met();
}
```

```
package limites.metodos.interfaces;
import limites.metodos.A;

public class F<H> extends A implements I {
	private H h;


	public F(H h) {
		this.h = h;
	}

	public void met() {
		System.out.println("H: " + h.getClass().getName() +
				". Invocado polimórficamente");
	}
}
```

```
package limites.metodos.interfaces;
import limites.metodos.B;

public class D<H> extends B implements I{
	private H h;

	public void met() {
		System.out.println("H: " + h.getClass().getName() +
				". Invocado polimórficamente");
	}
}
```

```
package limites.metodos.interfaces;
import limites.metodos.A;

public class Z <T>{
	private T t;

	public void agregar(T t) {
		this.t = t;
	}

	public T obtener() {
```

```
            return t;
      }

      public <U extends A & I> void inspeccionar(U u) {
            System.out.println("T: " + t.getClass().getName());
            System.out.println("U: " + u.getClass().getName());
            u.met();
      }
}


package limites.metodos.interfaces;
import limites.metodos.A;
import limites.metodos.B;

public class VerificaLimitesParametros {
      public static void main(String[] args) {
            Z<F<B>> z1 = new Z<>();

            z1.agregar(new F<B>(new B()));
            z1.inspeccionar(new F<D<A>>(new D<A>()));
      }
}
```

Este último programa produce la siguiente salida.

```
T: limites.metodos.interfaces.F
U: limites.metodos.interfaces.F
H: limites.metodos.interfaces.D. Invocado polimórficamente
```

Subtipos

Como se mencionara anteriormente, es posible asignar un objeto de un tipo a uno de otro tipo, siempre que estos tengan bien definida una conversión de tipo en su cadena de herencia. Por ejemplo, puede asignar un número entero a un objeto, ya que Object es superclase de Integer:

```
            Object objeto = new Object();
            Integer entero = new Integer(10);
            objeto = entero; // Bien
```

En la terminología orientada a objetos, esto se llama una relación "es un”. Dado que un número entero es un tipo de objeto, la asignación está permitida. Pero Integer es también un tipo Number, por lo que el siguiente código es válido también:

```
      public void unMetodo(Number n) {
            // se omite el cuerpo
      }

      public void otroMetodo(Number n) {
            unMetodo(new Integer(10)); // Bien
            unMetodo(new Double(10.1)); // Bien
      }
```

Lo mismo ocurre con los genéricos. Se puede realizar una invocación de tipo genérica, pasando como argumento Number de tipo, y cualquier llamada posterior de complemento se permitirá si el argumento es compatible con Number:

```
A<Number> a = new A<Number>();
a.agregar(new Integer(10)); // Bien
a.agregar(new Double(10.1)); // Bien
```

Ahora considérese el siguiente método en la clase A:

```
public void unMetodo(A<Number> a){
      // Hace algo
}
```

La clase B que se define como

```
public class B<T> extends A<T> {}
```

Y la siguiente invocación

```
A<Double> a2 = new A<>();
a.unMetodo(a2); //Error
B<Double> b1 = new B<>();
a2.unMetodo(b1); //Error
B<Number> b2 = new B<>();
a2.unMetodo(b2); //Correcto
```

¿Por qué dan error las invocaciones de métodos que se encuentran subrayadas? En el primer error se intenta pasar como parámetro al método un tipo A<Double> cuando admite como argumento A<Number>. Por más que Number es superclase de Double, ***las conversiones de tipo están definidas para las clases, no para los parámetros genéricos***. Esto se realizó de esta manera para mantener la consistencia del lenguaje. Se debe recordar que la finalidad de los parámetros genéricos es asegurar el tipo, pero si los parámetros admitieran subtipos de ellos, con sólo declarar a Object como el tipo del parámetro se podría realizar cualquier asignación y se perdería la seguridad de tipo buscada.

Por otra parte, en el segundo error subrayado, si bien B es un subtipo de A, el parámetro asignado a la creación de B no es exactamente Number, y el problema se repite. Sin embargo, en la última línea, como B es subtipo de A y el parámetro utilizado es Number, la conversión de tipo entre A y B está bien definida.

Parámetros desconocidos declarados con caracteres comodines

Al codificar se presentan situaciones como las siguientes cuando se quiere crear un parámetro genérico limitado al declarar una referencia (este tipo de declaraciones se utilizan para indicar que tipos admiten que se le asignen una determinada referencia):

- Se desconoce la superclase del parámetro
- Se desconoce la subclase del parámetro

Java admite estos dos estilos de declaraciones usando un carácter de comodín, el símbolo **?**.
Cuando se desconoce la subclase, pero se quiere declarar el parámetro la declaración es:

```
<? extends A>
```

Esta declaración debe leerse de la siguiente manera:

El parámetro es un tipo desconocido que es A o una subclase de A

Cuando lo que se desconoce es la superclase, la declaración se transforma en:

```
<? super D>
```

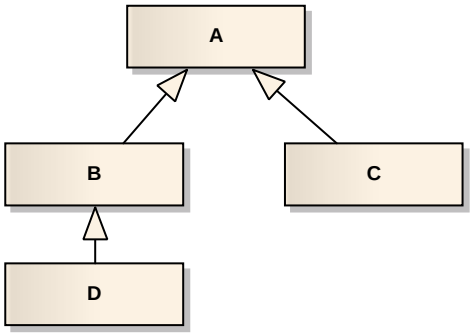
Que deberá leerse como

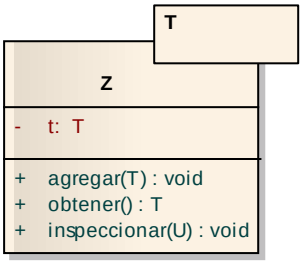
El parámetro es un tipo desconocido que es D o una superclase de D

Si se pretendiera declarar referencias a la clase Z de esta manera, las mismas deberán respetar el límite impuesto.

Ejemplo

```
Z<? extends A> z1 = new Z<B>();
Z<? extends A> z2 = new Z<C>();
Z<? extends A> z3 = new Z<D>();
Z<? super C> z4 = new Z<A>();
Z<? super C> z5 = new Z<B>(); // Error
Z<? super C> z6 = new Z<A>();

z1.agregar(new A()); // Error
z1.agregar(new A()); // Error
z1.agregar(new B()); // Error
z1.agregar(new C()); // Error
z1.agregar(new D()); // Error

z2.agregar(new A()); // Error
z2.agregar(new B()); // Error
z2.agregar(new C()); // Error
z2.agregar(new D()); // Error
```





Los errores se encuentran subrayados. El primer error es el siguiente:

Type mismatch: cannot convert from Z<B> to Z<? super C>

Es el más simple de detectar, porque B no está en cadena de herencia de C.

Los otros no son tan sencillos y requieren un poco de análisis. Todos los errores son similares con un mensaje parecido.

The method agregar(capture#1-of ? extends A) in the type Z<capture#1-of ? extends A> is not applicable for the arguments (A)

El método agregar tiene un prototipo como el siguiente

```
public void agregar(T t)
```

Si fuera posible agregar de la misma manera en Z un tipo A como un subtipo de A, la declaración de la variable genérica de Z, que define su estado, podría ser B, C o D, lo cual implica que el estado de Z puede ser un objeto de cualquiera de estos tipos, lo cual es incongruente ya que un objeto puede variar su estado, pero siempre **sobre el mismo tipo de variable**.

La API de Colecciones

Antecedentes

Las colecciones son un concepto intuitivo implementado desde los comienzos del lenguaje en diversas clases, algunas de uso interno y otras provistas por las API. Algunos ejemplos son los siguientes:

Vectores

Los vectores en Java están definidos para ser una colección de tipos primitivos de tamaño fijo, el cual es definido por una variable entera (Atención: utilizar otro tipo de datos para definir el tamaño de un vector genera un error, inclusive otro tipo entero como long).

El hecho que los vectores son objetos se hace evidente porque se puede acceder a propiedades de los mismos, como length o más aún, a los métodos que se obtienen por heredar de Object.

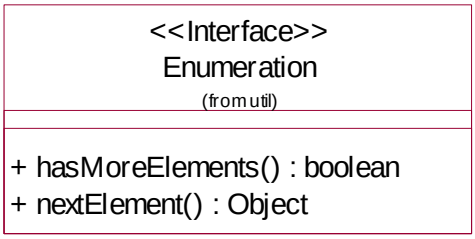
Las clases Vector y Stack

Vector es una clase histórica de Java. Permite la implementación de un vector de dimensión extensible por la incorporación de nuevos elementos. Esta clase fue remplazada en las nuevas versiones por ArrayList, sin embargo no fue depreciada en la actualidad y sigue vigente.

Por otro lado, la clase Stack implementa una pila y tiene como superclase a Vector. Si bien la operatoria de la pila se lleva a cabo con los métodos push y pop, se debe tener precaución en su forma de uso porque los métodos públicos de Vector siguen activos.

La interfaz Enumeration

Esta interfaz permite iterar sobre los elementos de una colección.



La nueva estructura de colecciones la remplazó por la interfaz Iterator la cual es utilizada en la mayoría de las situaciones. Sin embargo, muchas clases de utilizada todavía no soportan el uso de Iterator, por eso aún se sigue utilizando.

Ambas interfaces tienen un uso similar y conceptualmente son lo mismo, simplemente cambian los nombres de los métodos a utilizar.

Ejemplo

```
Enumeration enum = ...;
while (enum.hasNextElement()) {
        Object elemento = enum.nextElement();
        // procesar el elemento
```

Nota: Todas las clases del tipo colección presentadas que no sean genéricas se muestran con un diagrama diferente

Las Clases Dictionary, Hashtable, Properties

La clase Dictionary es una clase abstracta llena de métodos abstractos, lo cual, a nivel de diseño indica claramente, que debería haber sido una interfaz. Esta clase se creó originalmente para mantener conjuntos de pares clave – valor y fue remplazada por Map.

Las clases que heredan de Dictionary son Hashtable y Properties y conforman parte del conjunto original de diseño de colecciones que poseía Java.

Hashtable es una clase que cumple las funciones de un diccionario permitiendo almacenar objetos y sus claves asociadas (excepto `null`). La versión nueva de esta clase es HashMap que implementa la interfaz Map al igual que Hashtable, sin embargo, cuando se utiliza las propiedades de "thread safe", Hashtable es un poco más rápida que su nueva versión.

La clase Properties es una implementación de Hashtable para utilizar tan sólo strings, por lo tanto, no se necesitan conversiones de tipo para extraer los elementos porque están manejados por su tipo y no son un tipo Object. Esta colección, como se mencionó anteriormente, permite el ingreso de valores desde una corriente de ingreso así como también guardar sus elementos en una corriente de salida.

Su uso más frecuente es a través de acceder al Singlenton cuya referencia se obtiene de System.getProperties(), la cual Java utiliza durante el inicio de un programa para guardar inicialmente la propiedades de la máquina virtual.

Se puede resumir el conjunto de clases a utilizar como colecciones, entre históricas (las versiones originales) y nuevas (versiones nuevas del framework de colecciones), en la siguiente tabla:

Interfaz	Implementación		Versión original
Set	HashSet	TreeSet	
List	ArrayList	LinkedList	Vector
			Stack
Map	HaspMap	TreeMap	Hashtable
			Properties

Se debe tener en cuenta que las versiones originales de clases para el manejo de colecciones siguen vigentes por cuestiones de compatibilidad y se pueden utilizar, sin embargo, en caso de ser posible, es conveniente utilizar las nuevas implementaciones.

Como se mencionó anteriormente, las clases históricas para el manejo de colecciones se mantuvieron en las nuevas estructuras diseñadas. Se pueden enumerar las mismas como parte de los ejemplos y examinarlas desde el punto de vista de las nuevas implementaciones. De esta manera:

- Vector implementa la interfaz List
- Stack es una subclase de Vector y soporta los métodos push, pop y peek
- Hashtable implementa la interfaz Map.
- Enumeration es una variación de la interfaz Iterator:
  - Una enumeración es retornada por el método elements() en Vector, Stack y Hashtable

Estas clases son thread-safe y por lo tanto "importantes" conocerlas y apreciar la diferencia respecto de las nuevas implementaciones que no lo son y dejan la responsabilidad de sincronizar memoria compartida al programador

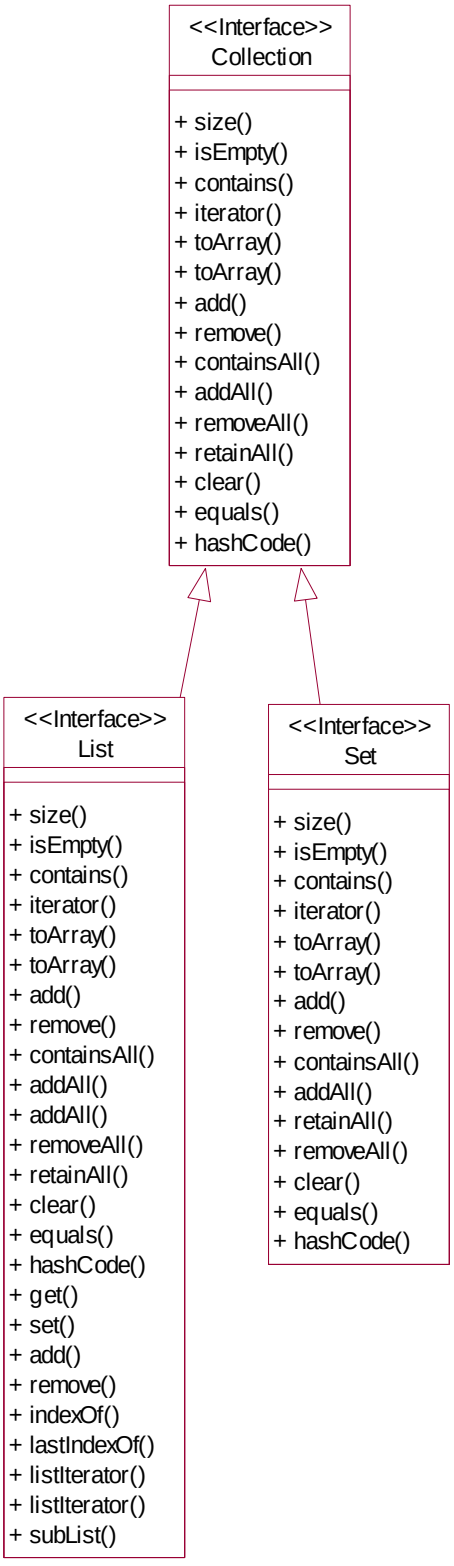
Nota: Existe una diferencia muy importante respecto de las nuevas clases. Cuando se diseñó la estructura de colecciones, se decidió que la sincronización de métodos debe ser responsabilidad del programador, por lo tanto, si se quiere una versión "thread safe", se deben sincronizar los métodos a utilizar, contrariamente a las clases históricas cuyos métodos están sincronizados.

Si un programador sabe utilizar Vector, también sabe como usar ArrayList y viceversa. El cambio fundamental radica en el método que actualiza un valor que en ArrayList se llama set mientras que en Vector se llama setElementAt.

Una visión actualizada de la clasificación de las colecciones más allá de las que fueron remplazadas, es ver las implementaciones de las interfaces según sus funcionalidades

	Tabla Hash	Vector Redimensionable	Árbol Balanceado	Lista Enlazada	Tabla Hash + Lista Enlazada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

La estructura de colecciones esta pensada como un conjunto de interfaces de las cuales especializan clases concretas para su uso. Este formato brinda flexibilidad en el manejo de toda la estructura para la declaración y gestión de referencias en las asociaciones con otras clases. Otro beneficio adicional de esta forma de implementación, es que los cambios que se puedan suscitar con las diferentes versiones no afecten a las implementaciones existentes, ya que ante la necesidad de funcionalidades específicas, se puede implementar las interfaces necesarias y especializar la clase según el requerimiento en particular.



Por lo tanto, para entender el framework de colecciones (estructura de clases e interfaces para el trabajo con colecciones) es fundamental comprender la filosofía de diseño y el funcionamiento esperado a través de las interfaces.

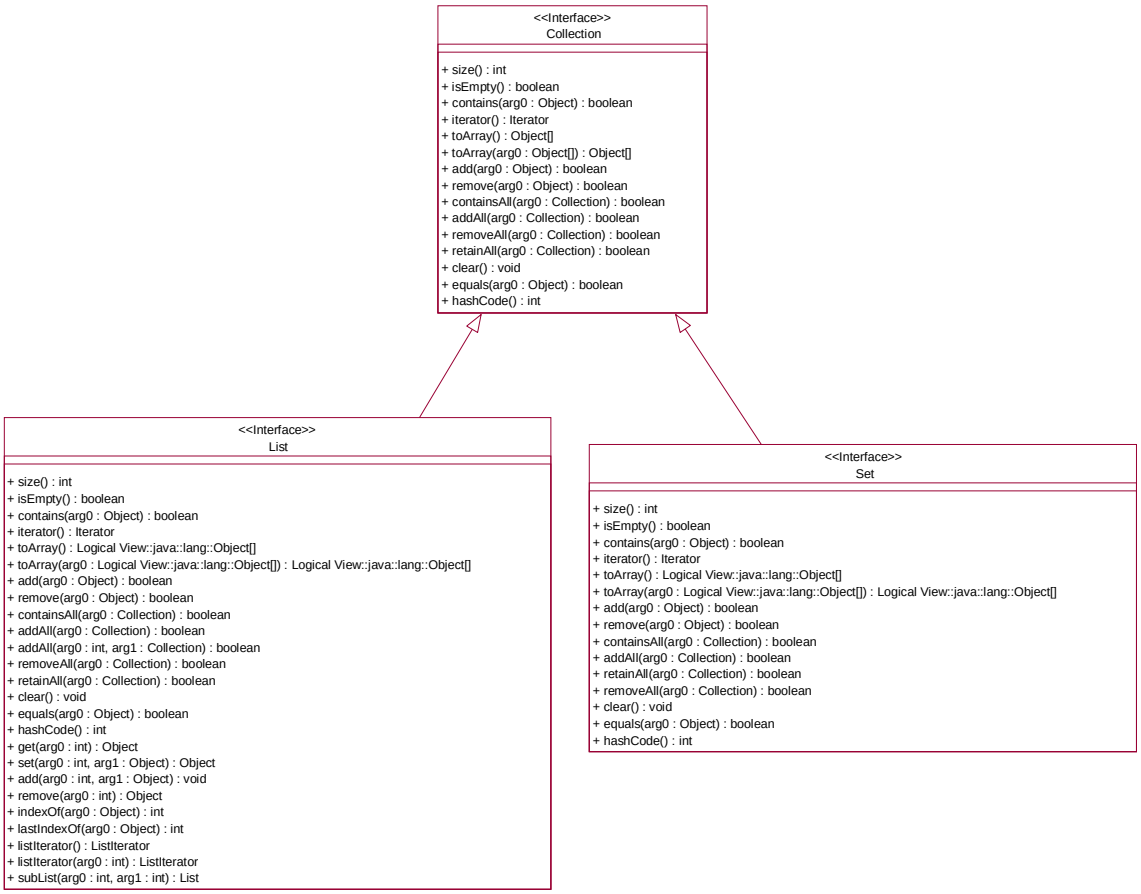
Con el apoyo que brinda UML, se puede analizar las interfaces desde sus diagramas. Por ejemplo, la interfaz Collection es superclase de List y Set, tal como muestra el siguiente diagrama UML.

Las colecciones se pueden interpretar como un conjunto, recordando que este último es una serie de elementos únicos agrupados, por lo tanto, no contiene duplicados o como listas, que si contienen duplicados pero conservan el orden en que se agregaron los elementos. Las colecciones, como son un concepto genérico, debe admitir en su diseño ambas posibilidades. En Java por cuestiones de diseño, se separan los conceptos en dos interfaces independientes que admiten estas funcionalidades distintas: Set y List. La primera no admite duplicados mientras que la segunda si.

La API Collection contiene interfaces que agrupan objetos como:

- **Collection** – Un grupo de objetos llamados elementos. Cualquier orden o la falta de este y la posibilidad de duplicados se especifica en cada implementación
- **Set** – Una colección sin orden. No se permiten duplicados
- **List** – Una colección ordenada. Se permiten duplicados

A continuación se muestra un diagrama completo de las cadenas de herencia entre las interfaces y sus respectivos diagramas individuales



Para trabajar con las colecciones y sus interfaces, conviene recordar la siguiente tabla:

Interfaz	Tipo	Superclase	Admite Duplicados	Orden
Collection	Superclase		Si	Secuencial
Map	Superclase		No	Indexado por clave
Set	Especialización	Collection	No	No
List	Especialización	Collection	Si	Indexado por subíndice

Las operaciones de la interfaz Collection

Esta interfaz soporta operaciones básicas como insertar y remover objetos. Por ejemplo, si se quiere remover objetos, con invocar al método apropiado se saca tan sólo una instancia del mismo si existiese.

Los métodos para agregar y remover son:

- `boolean` add(Object element)

- `boolean` remove(Object element)

Otros métodos de utilidad para consultas sobre el estado de la colección o su gestión son:

- `int` size()
- `boolean` isEmpty()
- `boolean` contains(Object element)
- Iterator iterator()

La interfaz Iterator

Esta interfaz es similar a Enumeration (la cuál se explicará posteriormente, aunque esta en desuso), sólo cambian los métodos utilizados. Las operaciones definidas soportan la navegación de una colección, la recuperación de un elemento y su remoción.

Operaciones grupales

La interfaz Collection permite operaciones grupales que pueden afectar a todos los elementos o a un grupo de ellos. Dichos métodos son los siguientes:

Método	Descripción
`boolean` containsAll(Collection collection)	Si contiene todos los elementos de otra colección
`boolean` addAll(Collection collection)	Si se pueden agregar todos los elementos de otra colección retorna verdadero y los agrega
`void` clear()	Remueve todos los elementos de la colección
`void` removeAll(Collection collection)	Remueve todos los elementos que se igual a los que se encuentran en la colección que se pasa como argumento
`void` retainAll(Collection collection)	Deja en la colección sólo aquellos elementos que se encuentran en la colección que es argumento

Set

Un ejemplo de la vida real de conjuntos puede ser el siguiente:

- Las letras minúsculas de la “a” a la “z”
- Los número naturales
- Los protocolos de una red
- Las medidas de longitud

De los ejemplos citados, se pueden definir una serie de propiedades respecto de los conjuntos, como ser:

- Existe una instancia de cada ítem (todos los ejemplos anteriores)
- Pueden ser finitos (protocolos de una red) o infinitos (números naturales)
- Pueden definir conceptos abstractos (medidas de longitud)

Como se mencionó anteriormente, Set es una interfaz que aquellas clases que la implementen deben ser colecciones que no admitan duplicados.

Ejemplo

```
package conjuntos;

import java.util.*;
public class EjemploDeSet {
	public static void main(String[] args) {
		Set set = new HashSet();
		set.add("uno");
		set.add("segundo");
		set.add("3ro");
		set.add(new Integer(4));
		set.add(new Float(5.0F));
		set.add("segundo ");
		set.add(new Integer(4)); // duplicado, no se agrega
		System.out.println(set);
	}
}
```

La salida generada por este programa es:

```
[4, 5.0, segundo , 3ro, segundo, uno]
```

List

Para entender el tratamiento particular que se diseño para cada tipo de colección, resta ver un ejemplo de listas. El siguiente código muestra el tratamiento que hace de esto un objeto del tipo ArrayList manteniendo el orden secuencial de ingreso y agregando duplicados.

Ejemplo

```
package conjuntos;

import java.util.*;
public class EjemploDeList {
	public static void main(String[] args) {
	List list = new ArrayList();
	list.add("uno");
	list.add("segundo");
	list.add("3ro");
	list.add(new Integer(4));
	list.add(new Float(5.0F));
	list.add("segundo"); // duplicado, se agrega
	list.add(new Integer(4)); // duplicado, se agrega
	System.out.println(list);
	}
}
```

La salida obtenida es

```
[uno, segundo, 3ro, 4, 5.0, segundo, 4]
```

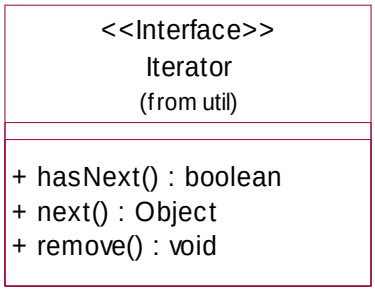
Notar la diferencia en la salida respecto del ejemplo de Set

Iteradores

Cuando un programa tiene un ciclo de ejecución, itera sobre las instrucciones definidas para dicho ciclo. El concepto de iterador en una colección es similar, salvo que se aplica a los elementos que posee, por lo tanto se puede definir para colecciones como:

La iteración es el proceso de recuperar cada elemento en una colección

La interfaz Iterator



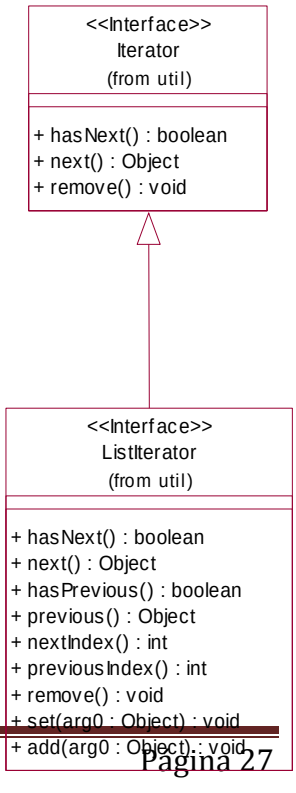
Un ejemplo de la interfaz se muestra en el diagrama UML.

Se utiliza next para obtener el elemento actual y moverse al próximo, siempre y cuando esto sea posible. Cuando se invoca el método remove, se remueve el último elemento accedido por el método next.

Ejemplo

```
Collection coleccion = ...;
Iterator iterador = collection.iterator();
while (iterador.hasNext()) {
  Object elemento = iterador.next();
  if (removerElemento(elemento)) {
       iterador.remove();
  }
}
```

Sin embargo, los iteradores fueron implantados como interfaces para ser especializadas según el funcionamiento en particular que se le quiera dar a la iteración para cada tipo particular de colección. De esta manera, se pueden citar los siguientes ejemplos:



- Un Iterator de un Set no tiene orden
- Un ListIterator de una lista puede inspeccionarse usando el método next() hacia adelante o hacia atrás con previous():

Por ejemplo, se puede ver en el siguiente código como se utiliza un ListIterator.

Ejemplo

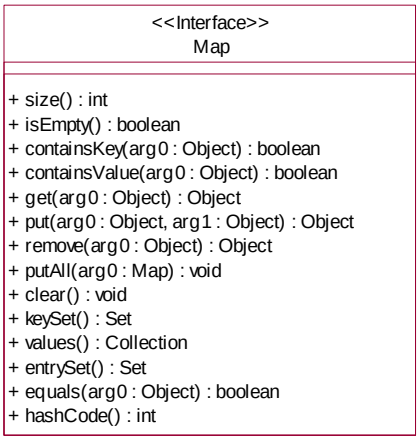
```
List lista = new ArrayList();
// Agregar algunos elementos
Iterator elementos = lista.iterator();
while (elementos.hasNext()) {
```

```
            System.out.println(elementos.next());
        }
```

Notar que la interfaz define métodos para utilizar en las listas que son necesarios cuando se maneja una colección de este tipo. El siguiente diagrama UML lo ejemplifica.

Mapas

Un mapa es un caso particular del conjunto, donde cada elemento ahora es un par "clave=valor", definiendo así una relación bidireccional entre los componentes de cada elemento, ya que cada componente está "mapeado" al otro. Algunos ejemplos simples de mapas pueden ser:



- Las direcciones IP y los nombres ingresados en un servidor DNS
- Las claves de una tabla y los registros asociados a cada una de ellas
- El nombre de una palabra y la cantidad de veces que aparece en un texto
- Los títulos de un libro y el número de página donde se encuentran
- Las conversiones de base numérica

Algunas propiedades que se pueden derivar de estos ejemplos son:

- Existe un solo par clave – valor en el mapa (todos los ejemplos anteriores)
- Pueden ser finitos (direcciones IP y nombres) o infinitos (conversiones de base numérica)
- Pueden definir conceptos abstractos (conversiones de base numérica)

Conceptualmente se agrega al desarrollador una cuarta interfaz a tener en cuenta para trabajar con la estructura de colecciones. Un ejemplo de implementación de la interfaz en una clase concreta se puede apreciar en el siguiente diagrama

Por definición, los objetos del tipo Map no admiten claves duplicadas y una clave solo puede asignarse a un valor. Si se proporciona otro valor para una clave existente, éste sobrescribe al anterior.

La interfaz Map proporciona tres métodos que permiten ver el contenido de un mapa como colecciones:

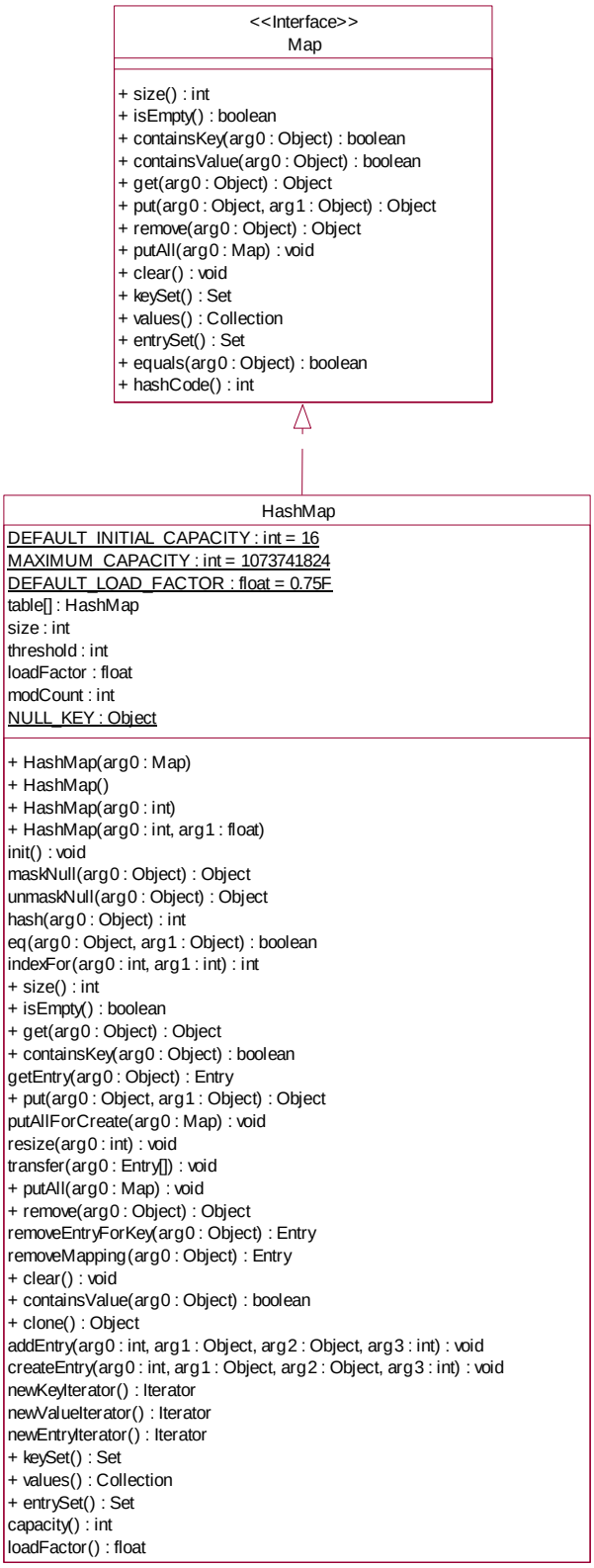
- **entrySet**: devuelve una variable Set que contiene todos los pares formados por una clave y un valor.
- **keySet**: devuelve una variable Set con todas las claves del mapa.
- **values**: devuelve una variable Collection con todos los valores contenidos en el mapa.

La interfaz Map no hereda la interfaz Collection porque representa asignaciones y no una colección de objetos. La interfaz SortedMap hereda de la interfaz Map. Algunas de las clases que implementan la interfaz Map son HashMap, TreeMap, IdentityHashMap y WeakHashMap.

El orden que presentan los iteradores de estas implementaciones de colección de mapas es específico de cada iterador.

En el ejemplo mostrado en el siguiente código el programa declara una variable mapa de tipo Map y la asigna a un objeto HashMap nuevo.

A continuación, agrega unos elementos al mapa mediante la operación put. Para demostrar que los mapas no admiten claves duplicadas, el programa intenta



agregar un valor nuevo con una clave ya existente. El resultado es que el valor nuevo sustituye al valor agregado anteriormente para la clave. A continuación, el programa utiliza las operaciones de visualización de colecciones keySet, values y entrySet para recuperar el contenido del mapa.

Ejemplo

```
package mapas;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class EjemploMap {
	public static void main(String args[]) {
		Map mapa = new HashMap();
		mapa.put("uno", "1ro");
		mapa.put("segundo", new Integer(2));
		mapa.put("tercero", "3º");
		// Sobrescribe la asignación anterior
		mapa.put("tercero", "III");
		// Devuelve el conjunto de las claves
		Set conjunto1 = mapa.keySet();
		// Devuelve la vista Collection de los valores
		Collection coleccion = mapa.values();
		// Devuelve el conjunto de las asignaciones de claves a valores
		Set conjunto2 = mapa.entrySet();
		System.out.println(conjunto1 + "\n" + coleccion + "\n" +
				conjunto2);
	}
}
```

La salida es

```
[tercero, segundo, uno]
[III, 2, 1ro]
[tercero=III, segundo=2, uno=1ro]
```

Ordenamiento de colecciones

Los métodos de Object y sus consecuencias en las colecciones

Mientras que el lenguaje Java no proporciona soporte directo a los vectores asociativos (aquellos que pueden tomar cualquier objeto como un índice), la presencia del método hashCode () en la clase Object anticipa claramente el uso de HashMap (y su predecesor, Hashtable) para resolver problemas que involucran claves y valores asociados. En condiciones ideales, los contenedores basados en hash ofrecen tanto la inserción como búsqueda eficaz. Soportar hash directamente en el modelo de objetos facilita el desarrollo y el uso de contenedores basados en este algoritmo.

Cómo se mostró en el módulo 5, dos objetos Integer sólo son iguales si contienen el mismo valor entero. Esto, junto con el hecho que sea inmutable, hace que sea práctico utilizar un número

entero como clave en un HashMap. Este enfoque basado en el valor almacenado para determinar la igualdad es utilizado por todas las clases de envoltorio primitivas en la biblioteca de clases de Java, tales como Integer, Float, Character y Boolean, así como también lo hace String (dos objetos de tipo cadena son iguales si contienen la misma secuencia de caracteres). Debido a que estas clases son inmutables e implementan hashCode y equals con sensatez, todas ellas son buenas claves hash (esto tiene suma importancia al ordenar objetos en base a otros que actúen de clave).

¿Qué pasaría si Integer no rescribiera equals y hashCode? Nada, si nunca se usa un entero como una clave en un HashMap o de otro tipo de colección basada en hash. Sin embargo, si se tuviera que utilizar un objeto entero como una clave en un HashMap, este no se sería capaz de recuperar con fiabilidad el valor asociado correspondiente, a menos que se utilice exactamente la misma instancia de Integer en la llamada a get como con la que se invocó a put. Para ello sería necesario asegurarse de que sólo se usa una sola instancia del objeto entero correspondiente a un determinado valor entero de todo el programa. No es necesario decir que este enfoque sería incómodo y propenso a errores.

El contrato de interfaz para el objeto requiere que si dos objetos son iguales de acuerdo a equals, entonces ellos deben tener el mismo valor de hashCode. ¿Por qué la clase Object necesita hashCode , cuando su capacidad de discriminar es totalmente es equals ? El método hashCode existe exclusivamente por motivos de eficiencia. Los arquitectos de la plataforma Java anticiparon la importancia de las clases de colección basadas en algoritmos hash, como Hashtable, HashMap y HashSet, en las aplicaciones típicas de Java, y comparar entre sí muchos objetos con equals puede ser costoso. Tener todos los objetos de Java como soporte a hashCode permite el almacenamiento y la recuperación eficiente utilizando colecciones basadas en algoritmos hash.

La especificación de Object ofrece una directriz vaga acerca de equals y hashCode para que sea consistente "sus resultados serán los mismos para las subsiguientes invocaciones", dado que "ninguna información utilizada en la comparación del objeto mediante equals se modificará". Esto suena algo así como "el resultado del cálculo no debe cambiar, a menos que lo haga". Esta declaración vaga generalmente se interpreta en el sentido de que los cálculos de la igualdad y el valor hash deben ser una función determinista de estado de un objeto y nada más.

¿Qué es igualdad?

Los requisitos para equals y hashCode impuestos por la especificación de la clase Object son bastante fáciles de seguir. Decidir si, y cómo, sobrescribir equals requiere un poco más de criterio. En el caso de clases de valores inmutables simples, como Integer (y de hecho, para casi todas las clases inmutables), la elección es bastante obvia - la igualdad debe basarse en la igualdad de el estado del objeto subyacente (el valor en su interior que determina su estado). En el caso de Integer, el único estado del objeto es el valor entero subyacente.

Para los objetos que pueden mutar, la respuesta no es siempre tan clara. ¿Se debe basar la igualdad de un objeto en la igualdad de referencias como en la implementación por defecto o se debe basar en el estado del objeto (como Integer y String)? No hay una respuesta fácil - depende

del uso previsto de la clase. Para los contenedores como List y Map, se podría haber hecho un argumento razonable de cualquiera. La mayoría de las clases en la biblioteca de clases de Java, incluyendo clases de contenedores sólo ofrecen un equals y hashCode basados en Object.

Si el valor hashCode de un objeto puede cambiar en función de su estado, entonces hay que tener cuidado al usar tales objetos como claves en las colecciones basadas en hash para asegurarse de no permitir que su estado cambie cuando se utilizan como claves hash. Todas las colecciones basadas en hash suponen que el valor de hash de un objeto no cambia mientras está en uso como clave de una colección. Si el código hash de la clave fuera a cambiar mientras se encuentra en una colección, podrían surgir consecuencias imprevisibles y confusas. Esto no suele ser un problema en la práctica - no es una práctica común el uso de un objeto mutable, como una lista, para que sea una clave en un HashMap.

Un ejemplo de una clase simple mutable que define equals y hashCode en función de su estado es Point. Dos objetos del tipo Point son iguales si hacen referencia a la mismas coordenadas (x, y), y el valor hash de un punto se deriva de la representación IEEE 754-bit de los valores X e Y de las coordenadas.

Para las clases más complejas, el comportamiento de equals y hashCode, pueden incluso ser impuestas por la especificación de una superclase o interfaz. Por ejemplo, la interfaz List requiere que un objeto del tipo List es igual a otro objeto si y sólo si el otro objeto también de una lista y contienen los mismos elementos (definido por Object.equals en dichos elementos) en el mismo orden. Los requisitos para hashCode se definen con aún más especificidad - el valor de hashCode de una lista deben cumplir con el siguiente cálculo:

```
hashCode = 1;
  Iterator i = list.iterator();
  while (i.hasNext()) {
      Object obj = i.next();
      hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
  }
```

No sólo es el valor de hash depende de los contenidos de la lista, sino que se especifica también el algoritmo específico para combinar los valores hash de los elementos individuales. (La clase String especifica un algoritmo similar para ser utilizado para calcular el valor hash de una cadena.)

Escribiendo métodos equals y hashCode propios

Remplazar el valor por defecto de equals es bastante fácil, pero remplazar un método equals que ya se ha sobrescrito puede ser extremadamente difícil de hacer sin violar la simetría o la exigencia de la transitividad. Al remplazar equals, siempre se debe incluir algunos comentarios Javadoc sobre éste para ayudar a aquellos que quieran extender la clase para que lo hagan correctamente.

Ejemplo

```
package igualdades;
```

```
public class A {
       final B valorNoNulo = new B();
       C otrovalorNoNulo = new C();
       int valorQueNoDeterminaEstado;

       public boolean equals(Object otro) {
           if (this == otro)
             return true;
           if (!(otro instanceof A))
             return false;
           A otroA = (A) otro;
           return
             (valorNoNulo.equals(otroA.valorNoNulo))
               && ((otrovalorNoNulo == null)
                   ? otroA.otrovalorNoNulo == null
                   : otrovalorNoNulo.equals(otroA.otrovalorNoNulo));
         }
}
```

Ahora que se ha definido equals, hay que definir hashCode () de una manera compatible. Una forma compatible, pero no tan útil para definir hashCode () es la siguiente:

```
public int hashCode() { return 0; }
```

Este enfoque produce un rendimiento horrible para HashMaps con un gran número de entradas, pero se ajusta a la especificación. Una aplicación más razonable de hashCode de A sería el siguiente.

Ejemplo

```
       public int hashCode() {
              int hash = 1;
              hash = hash * 31 + valorNoNulo.hashCode();
              hash = hash * 31
                     + (otrovalorNoNulo == null ? 0 : otrovalorNoNulo.hashCode());
              return hash;
       }
```

Nótese que ambas implementaciones delegan una porción del cálculo de equals o hashCode en los campos de estado de la clase (sus variables de instancia). Dependiendo de la clase, es posible que también delegar una parte del cálculo de equals o hashCode a la superclase. Para los campos de estado de tipo base, hay funciones de ayuda en las clases de envoltorios asociadas que pueden ayudar en la creación de valores de hash, como Float.floatToIntBits.

Posibles mejoras

La construcción de hash en la clase Object de la biblioteca de clases Java era un compromiso de diseño muy sensible - se facilita el uso de contenedores basados en hash mucho más fácil y más eficiente. Sin embargo, varios han hecho críticas del enfoque y la aplicación de algoritmos hash y la igualdad en la biblioteca de clases Java. Los contenedores de hash basados en java.util son muy

convenientes y fáciles de usar, pero puede no ser adecuado para aplicaciones que requieren un rendimiento muy alto. Mientras que la mayoría de ellos nunca será cambiada, vale la pena tener en cuenta al diseñar las aplicaciones que dependen en gran medida de la eficiencia de los contenedores basados en hash. Estas críticas incluyen:

- Rango muy chico para el uso de hash. El uso de `int`, en lugar de `long`, para el tipo de retorno de hashCode () aumenta la posibilidad de colisiones de hash.
- La mala distribución de los valores hash. Los valores hash para cadenas cortas y enteros pequeños en sí son números enteros pequeños, y están cerca los valores hash unos de otros. Una función más adecuada hash sería distribuir los valores hash de manera más uniforme en toda el rango de hash posible.
- No se definen las operaciones de hash. Mientras que algunas clases, como String y List, definen un algoritmo de hash que se utiliza en combinación con los valores hash de sus elementos constitutivos como un valor hash único, la especificación del lenguaje no define ningún medio aprobado de la combinación de los valores hash de varios objetos en un nuevo valor de hash. El truco utilizado por List, String, o la clase de ejemplo que se discutió anteriormente es simple, pero muy lejos de las matemáticas ideales. Tampoco las implementaciones de biblioteca de clases ofrecen cualquier algoritmo conveniente de hash que simplifique la creación de implementaciones más sofisticadas de hashCode ().
- Dificultad para escribir equals cuando se extiende una clase instanciable que ya rescribió equals. Las maneras "obvias" de definir equals () cuando se extiende una clase concreta que ya rescribió equals, fallan todas al no cumplir en general con los requisitos de la simetría o la transitividad que exige el método. Esto significa que se debe entender los detalles de la estructura y la implementación de clases que se extienden al rescribir equals, e inclusive se puede tener que exponer a los campos privados de la clase base como protegidos para hacerlo, lo cual viola los principios de buen diseño orientado a objetos.

Uso de Interfaces para hacer comparaciones

Las interfaces Comparable y Comparator resultan útiles para ordenar colecciones. La interfaz Comparable define un orden natural para las clases que la implementan. La interfaz Comparator se emplea para especificar la relación de orden. También permite anular el orden natural. Estas interfaces resultan útiles para ordenar los elementos de una colección.

Interfaz Comparable

La interfaz Comparable pertenece al paquete java.lang. Cuando se declara una clase, la implementación de JVM no tiene manera de determinar el orden que debe aplicarse a los objetos de la clase.

La interfaz Comparable permite definir el orden de los objetos de cualquier clase. Es posible ordenar las colecciones que contienen objetos de clases que implementan la interfaz Comparable.

Algunos ejemplos de clases de Java que implementan la interfaz Comparable son Byte, Long, String, Date y Float. Las clases numéricas emplean una implementación numérica, la clase String utiliza una implementación alfabética y la clase Date emplea una implementación cronológica. Al pasar una lista de tipo ArrayList que contiene elementos de tipo String al método estático sort de la clase Collections, se genera una lista en orden alfabético. Una lista que contiene elementos de tipo Date se clasificará por orden cronológico, mientras que una lista con elementos de tipo Integer se clasificará por orden numérico.

Para escribir tipos Comparable personalizados, es necesario implementar el método compareTo de la interfaz Comparable. El siguiente código muestra cómo implementar la interfaz Comparable. La clase Horario implementa la interfaz Comparable para que los objetos de esta clase puedan compararse entre sí.

Ejemplo

```
package interfaces.comparable;
public class Horario implements Comparable<Horario> {
	private int dia = 0;
	private int horaComienzo = 0;
	private int minutosComienzo = 0;
	private int horaFin = 0;
	private int minutosFin = 0;
	private int turnosPorHora = 0;

	public Horario(int dia, int horaComienzo, int minutosComienzo,
			int horaFin, int minutosFin, int turnosPorHora) {
		super();
		this.dia = dia;
		this.horaComienzo = horaComienzo;
		this.minutosComienzo = minutosComienzo;
		this.horaFin = horaFin;
		this.minutosFin = minutosFin;
		this.turnosPorHora = turnosPorHora;
	}

	public Horario(Horario f) {
		this.dia = f.dia;
		this.horaComienzo = f.horaComienzo;
		this.minutosComienzo = f.minutosComienzo;
		this.horaFin = f.horaFin;
		this.minutosFin = f.minutosFin;
	}

	public Horario agregar(int masDias) {
		Horario nuevaFecha = new Horario(this);
		nuevaFecha.dia += nuevaFecha.dia;
		return nuevaFecha;
	}

	public void imprimir() {
		System.out.println("Horario: ");
		System.out.println("Día: " + dia);
```

```
		System.out.println("Hora de comienzo: " + horaComienzo);
		System.out.println("Minutos de comienzo: " + minutosComienzo);
		System.out.println("Hora de fin: " + horaFin);
		System.out.println("Minutos de fin: " + minutosFin);
		System.out.println("Turnos por hora:" + turnosPorHora);
	}

	public int getDia() {
		return dia;
	}

	public int getHoraComienzo() {
		return horaComienzo;
	}

	public int getMinutosComienzo() {
		return minutosComienzo;
	}

	public int getHoraFin() {
		return horaFin;
	}

	public int getMinutosFin() {
		return minutosFin;
	}

	public int getTurnosPorHora() {
		return turnosPorHora;
	}

	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + dia;
		result = prime * result + horaComienzo;
		result = prime * result + horaFin;
		result = prime * result + minutosComienzo;
		result = prime * result + minutosFin;
		result = prime * result + turnosPorHora;
		return result;
	}

	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Horario other = (Horario) obj;
		if (dia != other.dia)
			return false;
		if (horaComienzo != other.horaComienzo)
			return false;
```

```
			if (horaFin != other.horaFin)
				return false;
			if (minutosComienzo != other.minutosComienzo)
				return false;
			if (minutosFin != other.minutosFin)
				return false;
			if (turnosPorHora != other.turnosPorHora)
				return false;
			return true;
		}

		public String toString() {
			return "Horario [dia=" + dia + ", horaComienzo=" + horaComienzo
				+ ", minutosComienzo=" + minutosComienzo + ", horaFin="
				+ horaFin + ", minutosFin=" + minutosFin + ", turnosPorHora="
				+ turnosPorHora + "]";
		}

		public int compareTo(Horario o) {
			if (dia > o.dia || horaComienzo > o.horaComienzo
				|| minutosComienzo > o.minutosComienzo || horaFin > o.horaFin
				|| minutosFin > o.minutosFin)
				return 1;
			else if (dia < o.dia || horaComienzo < o.horaComienzo
				|| minutosComienzo < o.minutosComienzo || horaFin < o.horaFin
				|| minutosFin < o.minutosFin)
				return -1;
			else
				return 0;
		}
}
```

La clase que verifica el funcionamiento es la siguiente

```
package interfaces.comparable;
import java.util.TreeSet;

public class ConjuntoHorarios {

		public static void main(String[] args) {
			TreeSet conjuntoDeHorarios = new TreeSet();
			conjuntoDeHorarios.add(new Horario(2, 10, 30, 15, 0, 4));
			conjuntoDeHorarios.add(new Horario(2, 8, 0, 12, 0, 4));
			conjuntoDeHorarios.add(new Horario(3, 9, 0, 13, 0, 4));
			conjuntoDeHorarios.add(new Horario(3, 8, 0, 13, 0, 3));
			Object[] estudianteArray = conjuntoDeHorarios.toArray();
			Horario s;
			for (Object obj : estudianteArray) {
				s = (Horario) obj;
				System.out.printf(
				"Día = %s Comienza a las %s:%s y termina a las %s:%s\n",
				s.getDia(),  s.getHoraComienzo(), s.getMinutosComienzo(),
				s.getHoraFin(), s.getMinutosFin());
			}
		}
```

```
}
```

La salida generada por el programa ConjuntoHorarios es

```
Día = 2 Comienza a las 8:0 y termina a las 12:0
Día = 2 Comienza a las 10:30 y termina a las 15:0
Día = 3 Comienza a las 8:0 y termina a las 13:0
Día = 3 Comienza a las 9:0 y termina a las 13:0
```

El ejemplo ordena los objetos del tipo Horario por día, hora de comienzo, minutos de comienzo, hora de fin, y minutos de fin. Se debe notar que la variable de instancia que almacena la cantidad de turnos no se la considera significativa para ordenar los horarios. Esta es una decisión de diseño que establece el "orden natural" de los objetos de tipo horario.

Observe que los horarios se comparan según sus variables de instancia, que definen su estado. Esto ocurre porque al ordenar los elementos de la colección TreeSet, TreeSet comprueba si los objetos se encuentran ordenados usando el método compareTo para compararlos.

Algunas colecciones, como TreeSet, se ordenan. La implementación de TreeSet necesita saber cómo ordenar los elementos. Si los elementos tienen un orden natural, TreeSet emplea el orden natural. En caso contrario, es necesario ayudarle. Por ejemplo, la clase TreeSet incluye el siguiente constructor, que admite Comparator como parámetro.

TreeSet(Comparator comparator)

Este constructor crea un conjunto de árbol vacío, ordenado según el parámetro Comparator especificado. La siguiente sección ofrece una explicación detallada del uso de la interfaz Comparator.

Interfaz Comparator

La interfaz Comparator ofrece una mayor flexibilidad a la hora de ordenar. Por ejemplo, en el caso de la clase Horario descrita anteriormente, los horarios sólo se ordenaron mediante una comparación de sus días, horas y minutos en los cuales están comprendidos. No se ordenó de ninguna otra forma ni se usó algún otro criterio. Esta sección muestra cómo aumentar la flexibilidad de la ordenación con la interfaz Comparator.

La interfaz Comparator forma parte del paquete java.util. Se utiliza para comparar los objetos según el orden personalizado en lugar del orden natural. Por ejemplo, permite ordenar los objetos según un orden distinto del orden natural.

También se emplea para ordenar objetos que no implementan la interfaz Comparable.

Para escribir un comparador Comparator personalizado, es necesario agregar una implementación del método compare a la interfaz:

```
int compare(Object o1, Object o2)
```

Este método compara dos argumentos de orden y devuelve un entero negativo si el primer argumento es menor que el segundo. Si ambos son iguales, devuelve un cero y si el primer argumento es mayor que el segundo, devuelve un entero positivo, como lo hace la interfaz Comparable.

El en caso supuesto que se quiera generar una forma de comparar los mismos horarios pero sólo teniendo en cuenta las horas, pero nada más, habría que cambiar la implementación de Comparable. En lugar de hacer esto se crea una clase que manejará la comparación a través de implementar Comparator.

Ejemplo

```
package interfaces.comparator;
import java.util.Comparator;

public class ComparaHorasHorario implements Comparator {

	public int compare(Object o1, Object o2) {
		if (((Horario) o1).getHoraComienzo() == ((Horario)o2)
				.getHoraComienzo())
			if (((Horario) o1).getHoraFin() == ((Horario) o2)
					.getHoraFin())
				return 0;
			else if (((Horario) o1).getHoraFin() < ((Horario) o2)
					.getHoraFin())
				return 1;
			else
				return -1;
		else if (((Horario) o1).getHoraComienzo() < ((Horario) o2)
				.getHoraComienzo())
			return -1;
		else
			return 1;
	}
}
```

Es posible crear varias clases para comparar los horarios. Por ejemplo, se puede volver a utilizar la interfaz para crear otro criterio de comparación, en este caso, por horas y minutos, lo cual cambiaría el criterio para ordenar los elementos internamente.

Ejemplo

```
package interfaces.comparator;
import java.util.Comparator;

public class ComparaHorasYMinutosHorario implements Comparator {

	public int compare(Object o1, Object o2) {

		if (((Horario) o1).getHoraComienzo() == ((Horario) o2)
				.getHoraComienzo()) {
			if (((Horario) o1).getHoraFin() == ((Horario) o2)
```

```
                        .getHoraFin()) {
                        if (((Horario) o1).getMinutosComienzo() == ((Horario)
                              o2).getMinutosComienzo()) {
                              if (((Horario) o1).getMinutosFin() == ((Horario)
                                    o2).getMinutosFin())
                                    return 0;
                              else {
                                    if (((Horario) o1).getMinutosFin() <
                                          ((Horario) o2).getMinutosFin())
                                          return -1;
                                    else
                                          return 1;
                              }
                        } else {
                              if (((Horario) o1).getMinutosComienzo() <
                                    ((Horario) o2).getMinutosComienzo())
                                    return -1;
                              else
                                    return 1;
                        }
                  } else {
                        if (((Horario) o1).getHoraFin() < ((Horario) o2)
                              .getHoraFin())
                              return -1;
                        else
                              return 1;
                  }
            } else {
                  if (((Horario) o1).getHoraComienzo() < ((Horario) o2)
                              .getHoraComienzo())
                        return -1;
                  else
                        return 1;
            }
      }
}
```

Los efectos de utilizar diferentes criterios de ordenamiento se pueden ver inmediatamente si se crea una colección que admita a Comparator como argumento en el constructor (lo cual indica el criterio de ordenamiento que utilizará la colección). Por ejemplo, si se utilizan las dos clases que se mostraron anteriormente como ejemplos, se obtendría dos formas diferentes de ordenar la colección.

Ejemplo

```
package interfaces.comparator;
import java.util.TreeSet;

public class ConjuntoHorarios {

      public static void main(String[] args) {
            ComparaHorasHorario ch = new ComparaHorasHorario();
            ComparaHorasYMinutosHorario cm = new ComparaHorasYMinutosHorario();
            TreeSet horarioSetPorHora = new TreeSet(ch);
```

```
            TreeSet horarioSetPorMinuto = new TreeSet(cm);

            Horario h1 = new Horario(2, 10, 30, 15, 0, 4);
            Horario h2 = new Horario(2, 8, 0, 12, 0, 4);
            Horario h3 = new Horario(3, 9, 0, 13, 0, 4);
            Horario h4 = new Horario(3, 8, 0, 13, 0, 3);

            horarioSetPorHora.add(h1);
            horarioSetPorHora.add(h2);
            horarioSetPorHora.add(h3);
            horarioSetPorHora.add(h4);

            System.out.println("--- Ordenando sólo por hora ---");
            Object[] vectorHorarios = horarioSetPorHora.toArray();
            Horario s;
            for (Object obj : vectorHorarios) {
                  s = (Horario) obj;
                  System.out.printf(
                  "Día = %s Comienza a las %s:%s y termina a las %s:%s\n",
                  s.getDia(), s.getHoraComienzo(), s.getMinutosComienzo(),
                  s.getHoraFin(), s.getMinutosFin());
            }

            System.out.println("--- Ordenando por horas y minutos ---");

            horarioSetPorMinuto.add(h1);
            horarioSetPorMinuto.add(h2);
            horarioSetPorMinuto.add(h3);
            horarioSetPorMinuto.add(h4);

            vectorHorarios = horarioSetPorMinuto.toArray();
            for (Object obj : vectorHorarios) {
                  s = (Horario) obj;
                  System.out.printf(
                  "Día = %s Comienza a las %s:%s y termina a las %s:%s\n",
                  s.getDia(), s.getHoraComienzo(), s.getMinutosComienzo(),
                  s.getHoraFin(), s.getMinutosFin());
            }
      }
}
```

En la salida que se obtiene al ejecutar el programa se pueden apreciar ambas formas de ordenar los elementos de la colección dado que se crean dos TreeSet con las clases que implementan los criterios de comparación a través de Comparator.

```
--- Ordenando sólo por hora ---
Día = 3 Comienza a las 8:0 y termina a las 13:0
Día = 2 Comienza a las 8:0 y termina a las 12:0
Día = 3 Comienza a las 9:0 y termina a las 13:0
Día = 2 Comienza a las 10:30 y termina a las 15:0
--- Ordenando por horas y minutos ---
Día = 2 Comienza a las 8:0 y termina a las 12:0
Día = 3 Comienza a las 8:0 y termina a las 13:0
Día = 3 Comienza a las 9:0 y termina a las 13:0
```

```
Día = 2 Comienza a las 10:30 y termina a las 15:0
```

Genéricos y colecciones

Las clases de las colecciones tradicionales utilizan los tipos Object para admitir contener diferentes tipos de entradas y salidas. Es necesario convertir el tipo de objeto explícitamente para poder recuperarlo y esto no garantiza la seguridad del tipo.

Aunque la infraestructura de colecciones existente admite las colecciones homogéneas (es decir, colecciones con un tipo de objeto específico, por ejemplo Date), no había ningún mecanismo para evitar la inserción de otros tipos de objetos en la colección. Para poder recuperar un objeto, casi siempre había que convertirlo.

Este problema se ha resuelto con la funcionalidad de los genéricos. Dicha funcionalidad se ha introducido a partir de la plataforma Java SE 5.0. Para ello, se proporciona información para el compilador sobre el tipo de colección utilizado mediante el parámetro genérico con el que se va a utilizar la colección. Así, la comprobación del tipo se resuelve de forma automática antes del tiempo de ejecución. Esto elimina la conversión explícita de los tipos de datos para su uso en la colección durante la ejecución del programa. Gracias al autoboxing de los tipos primitivos, es posible utilizar tipos genéricos para escribir código más sencillo y comprensible. Sin embargo, se debe evitar en lo posible usar autoboxing porque es costoso a nivel de recursos cada vez que convierte un objeto o lo hace seleccionable para el recolector de basura.

Antes de los tipos genéricos, el código podría ser similar al siguiente

```
ArrayList lista = new ArrayList();
lista.add(0, new Integer(42));
int total = ((Integer)lista.get(0)).intValue();
```

En este código, se necesita una clase de envoltorio Integer para la conversión del tipo mientras se recupera el valor del número entero que existe en lista. En el momento de la ejecución, el programa debe controlar el tipo para lista.

Al aplicar los tipos genéricos, ArrayList debe declararse como ArrayList<Integer> para informar al compilador sobre el tipo de colección que se va a utilizar. Al recuperar el valor, no se necesita una clase envoltorio Integer.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

La función de autoboxing encaja muy bien con el API de tipos genéricos. Con autoboxing, el código de ejemplo podría escribirse tal como se muestra en el siguiente código.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Este código utiliza la declaración e instanciación genérica para declarar y crear una instancia del tipo ArrayList con objetos del tipo Integer. Por tanto, al agregar un tipo que no sea Integer a la lista se genera un error de compilación.

Nota: Los tipos genéricos se habilitan de forma predeterminada a partir de la plataforma Java SE 5.0. Es posible deshabilitar los tipos genéricos mediante la opción -source 1.4 del compilador javac.

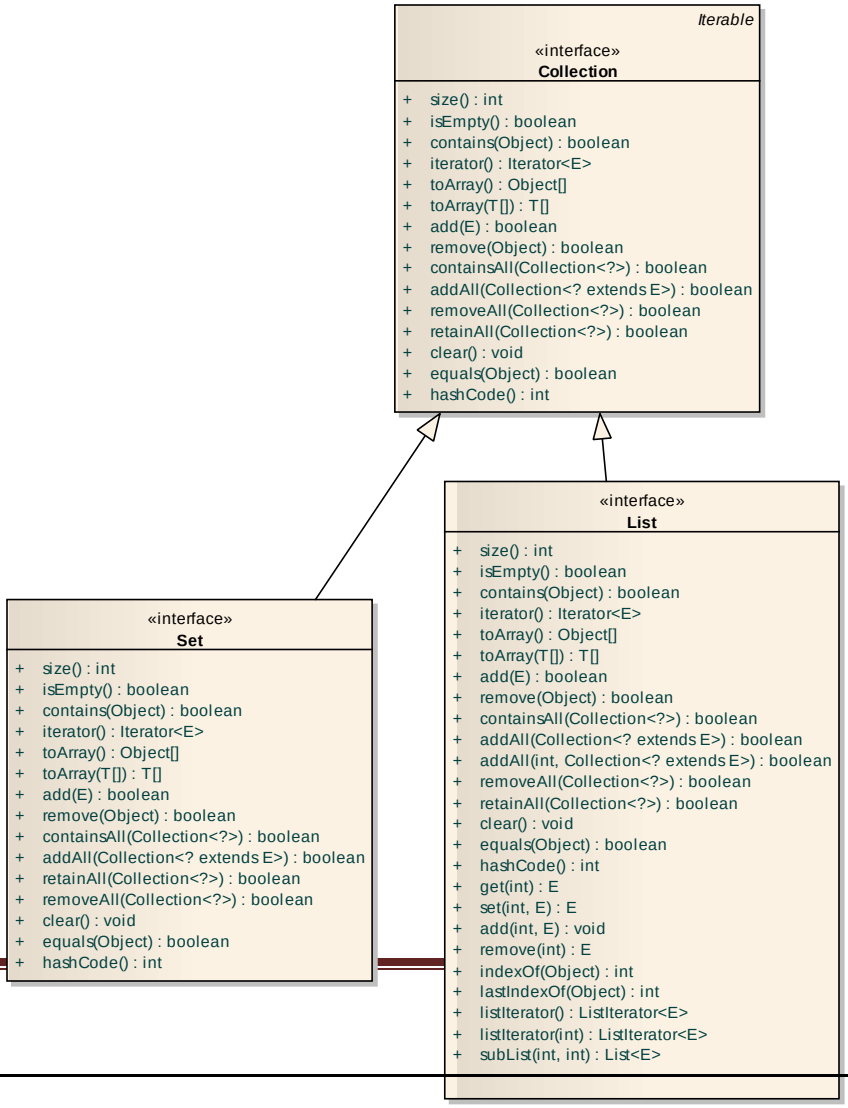
Los tipos genéricos son ideales para utilizar en cualquier clase que no tenga dependencia con su estado para prestar sus servicios. Todos los algoritmos fundamentales son candidatos ideales para esto.

La API de colecciones genéricas

Las colecciones tradicionales fueron actualizadas a partir de Java 1.5 para utilizar genéricos. Los parámetros se definen para las nuevas colecciones según la siguiente convención:

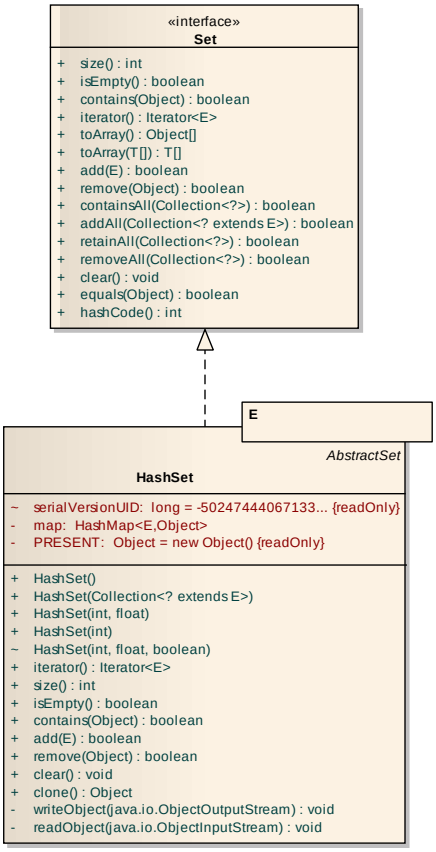
- E - Elemento (usado extensamente por el Framework de colecciones de Java)
- K - Clave
- N - Número
- T - Tipo
- V - Valor
- S,U,V etc. – tipos 2º, 3º, 4º…

De esta manera, se pueden apreciar en los diagramas UML también como se definen las clases que tienen parámetros genéricos. Cuando se trata de interfaces se suele indicar el uso del genérico dentro de las declaraciones. En cambio, cuando se trata de clases concretas, se suele indicar resaltando en un rectángulo superpuesto el parámetro a recibir



Nota acerca de los diagramas: Se puede observar en los diagramas UML el uso de parámetros genéricos con letras como E, K o T. La razón es que los diagramas implementan la versión moderna utilizando genéricos en las colecciones.

Un ejemplo de como queda el diagrama en la clase concreta HashSet es el siguiente.



Set genérico

En el siguiente ejemplo es una modificación del utilizado para explicar las colecciones tradicionales. El programa declara una variable (set) de tipo Set<String> y la asigna a un objeto HashSet<String> nuevo. A continuación, agrega unos elementos de tipo String e imprime el conjunto como salida estándar. Las líneas que producen un error de compilación (porque, por

ejemplo, un número entero no es una cadena) están marcadas con comentarios. El código muestra la implementación de Set con tipos genéricos.

Ejemplo

```
package conjuntos;

import java.util.*;
public class EjemploDeSet {
	public static void main(String[] args) {
		Set<String> set = new HashSet<String>();
		set.add("uno");
		set.add("segundo");
		set.add("3ro");
		// La siguiente línea da error de compilación
		//		set.add(new Integer(4));
		// La siguiente línea da error de compilación
		//		set.add(new Float(5.0F));
		set.add("segundo"); // duplicado, no se agrega
		// La siguiente línea da error de compilación
		//		set.add(new Integer(4)); // duplicado, no se agrega
		System.out.println(set);
	}
}
```

La salida producida es la siguiente

```
[3ro, segundo, uno]
```

List genérico

Análogamente, se presenta el mismo ejemplo para el manejo de listas. Notar nuevamente que las líneas que provocan errores de compilación a causa de la violación del tipo establecido en el parámetro genérico fueron marcadas como comentarios.

Ejemplo

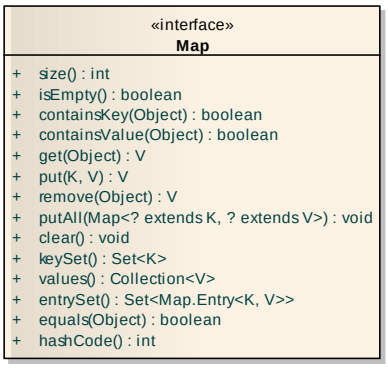
```
package listas;

import java.util.*;
public class EjemploDeList {
	public static void main(String[] args) {
	List<String> list = new ArrayList<String>();
	list.add("uno");
	list.add("segundo");
	list.add("3ro");
	// La siguiente línea da error de compilación
	//    list.add(new Integer(4));
	// La siguiente línea da error de compilación
	//    list.add(new Float(5.0F));
	list.add("segundo"); // duplicado, se agrega
	// La siguiente línea da error de compilación
	//    list.add(new Integer(4)); // duplicado, se agrega
```

```
        System.out.println(list);
        }
}
```

La salida producida por el programa es la siguiente

```
[uno, segundo, 3ro, segundo]
```

Map genérico



Siguiendo el mismo estilo, se presenta para su comparación con el ejemplo de colección tradicional, el uso de mapas genéricos. El diagrama UML de la interfaz Map muestra la aplicación de los parámetros genéricos.

Ejemplo

```
package mapas;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

public class EjemploMap {
        public static void main(String args[]) {
                Map<String, String> mapa = new HashMap<String, String>();
                mapa.put("uno", "1ro");
                // La siguiente línea da error de compilación
                //           mapa.put("segundo", new Integer(2));
                mapa.put("tercero", "3º");
                // Sobrescribe la asignación anterior
                mapa.put("tercero", "III");
                // Devuelve el conjunto de las claves
                Set<String> conjunto1 = mapa.keySet();
                // Devuelve la vista Collection de los valores
                Collection<String> coleccion = mapa.values();
                // Devuelve el conjunto de las asignaciones de claves a valores
```

```
            Set<Entry<String,String>> conjunto2 = mapa.entrySet();
            System.out.println(conjunto1 + "\n" +
                  coleccion + "\n" + conjunto2);
      }
}
```

El siguiente diagrama muestra la implementación de la interfaz Map en la clase HashMap utilizada en el programa


«interface»
Map
+ size() : int
+ isEmpty() : boolean
+ containsKey(Object) : boolean
+ containsValue(Object) : boolean
+ get(Object) : V
+ put(K, V) : V
+ remove(Object) : V
+ putAll(Map<? extends K, ? extends V>) : void
+ clear() : void
+ keySet() : Set<K>
+ values() : Collection<V>
+ entrySet() : Set<Map.Entry<K, V>>
+ equals(Object) : boolean
+ hashCode() : int
K
V
AbstractMap
HashMap
~ DEFAULT_INITIAL_CAPACITY: int = 16 {readOnly}
~ MAXIMUM_CAPACITY: int = 1 << 30 {readOnly}
~ DEFAULT_LOAD_FACTOR: float = 0.75f {readOnly}
~ table: Entry ([])
~ size: int
~ threshold: int
~ loadFactor: float {readOnly}
~ modCount: int
- entrySet: Set<Map.Entry<K,V>> = null
- serialVersionUID: long = 362498820763181265L {readOnly}
+ HashMap(int, float)
+ HashMap(int)
+ HashMap()
+ HashMap(Map<? extends K, ? extends V>)
~ init() : void
~ hash(int) : int
~ indexFor(int, int) : int
+ size() : int
+ isEmpty() : boolean
+ get(Object) : V
- getForNullKey() : V
+ containsKey(Object) : boolean
~ getEntry(Object) : Entry<K,V>
+ put(K, V) : V
- putForNullKey(V) : V
- putForCreate(K, V) : void
- putAllForCreate(Map<? extends K, ? extends V>) : void
~ resize(int) : void
~ transfer(Entry[]) : void
+ putAll(Map<? extends K, ? extends V>) : void
+ remove(Object) : V
~ removeEntryForKey(Object) : Entry<K,V>
~ removeMapping(Object) : Entry<K,V>
+ clear() : void
+ containsValue(Object) : boolean
- containsNullValue() : boolean
+ clone() : Object
~ addEntry(int, K, V, int) : void
~ createEntry(int, K, V, int) : void
~ newKeyIterator() : Iterator<K>
~ newValueIterator() : Iterator<V>
~ newEntryIterator() : Iterator<Map.Entry<K,V>>
+ keySet() : Set<K>
+ values() : Collection<V>
+ entrySet() : Set<Map.Entry<K,V>>
- entrySet0() : Set<Map.Entry<K,V>>
- writeObject(java.io.ObjectOutputStream) : void
- readObject(java.io.ObjectInputStream) : void
~ capacity() : int
~ loadFactor() : float

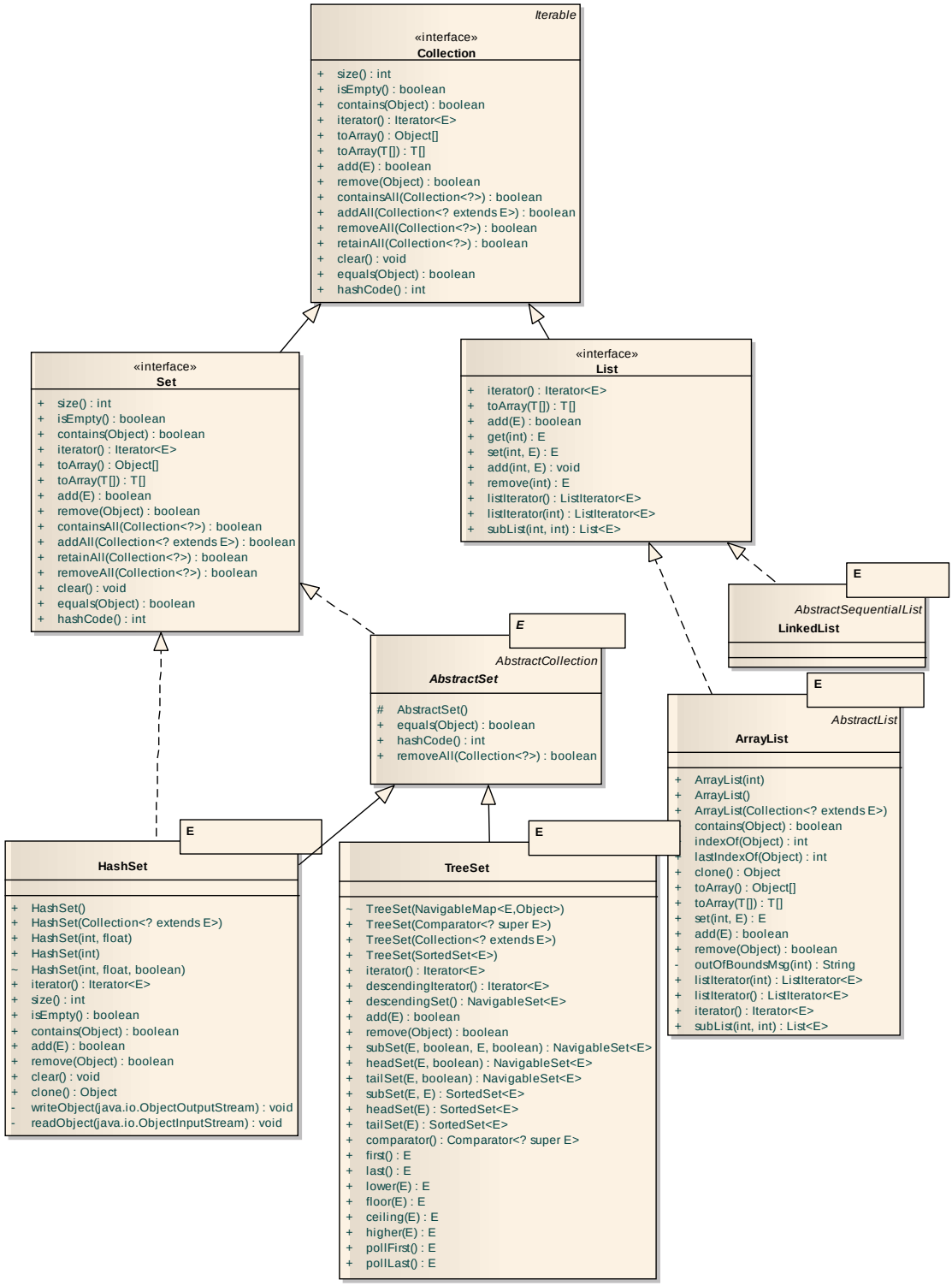
Análisis de los parámetros de tipo

La siguiente tabla muestra las declaraciones en la utilización de colecciones para comparar las que son previas a Java 1.5 con las posteriores. Notar como la introducción del parámetro genérico elimina la solución de compromiso al declarar Object como parámetro o argumento cuando se desconoce el tipo.

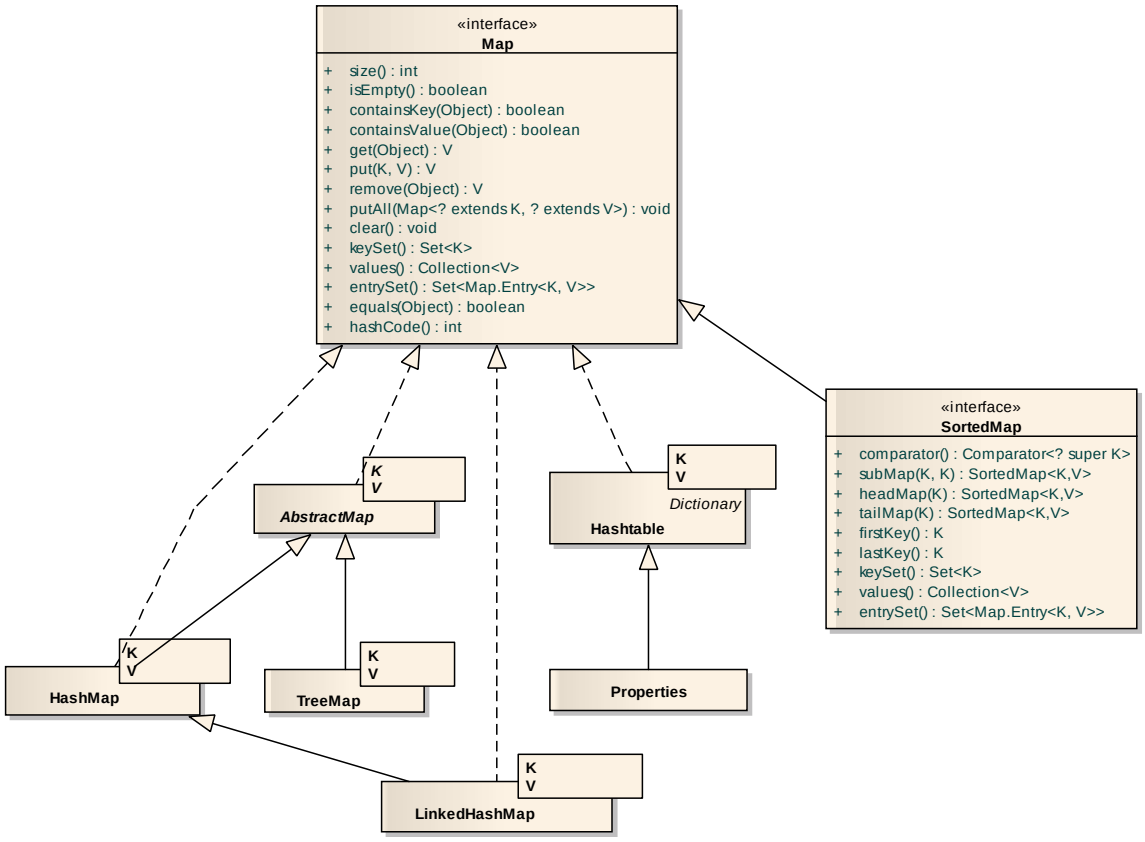
Categoría	Clase no genérica	Clase genérica
Declaración de clase	public class ArrayList extends AbstractList implements List	public class ArrayList<E> extends AbstractList<E> implements List <E>
Declaración de constructor	public ArrayList(int capacity)	public ArrayList(int capacity)
Declaración de método	public void add(Object o) public Object get(int index)	public void add(E o) public E get(int index)
Ejemplos de declaraciones de variables	ArrayList list1; ArrayList list2;	ArrayList <String> a3; ArrayList <Date> a4;
Ejemplos de declaraciones de instancias	list1 = new ArrayList(10); list2 = new ArrayList(10);	a3= new ArrayList<String> (10); a4= new ArrayList<Date> (10);

De la misma manera se puede apreciar la utilización del parámetro genérico cuando se los especializa (como se muestra para los tipos String y Date).

El siguiente diagrama UML muestra las principales interfaces y clases de la implementación de la API de genéricos



El siguiente diagrama UML muestra las principales interfaces y clases derivadas de Map en su versión con genéricos.



Parámetros comodines en las colecciones

Tipos asegurados

Las colecciones que utilizan genéricos aseguran el tipo en las asignaciones de referencias.

Ejemplo

```
package invarianza;

import java.awt.List;
import java.util.ArrayList;

public class VerificaSeguridadDeTipo {

	public static void main(String[] args) {
		ArrayList<D> ld = new ArrayList<D>();

		ld.add(new D()); // Bien
		D d = ld.get(0); // Bien

		ld.add(new C()); // ¡Error de compilación!

		// por tanto esta asignación también
		C c = ld.get(0);
```

```
	}
}
```

Por lo tanto, sólo se pueden agregar y recuperar los tipos apropiados de una colección según fueran definidos en su creación. Esto garantiza el tipo que maneja la colección y no propaga errores.

Invarianza

Debido a que los tipos están asegurados, la pregunta siguiente es ¿qué pasa cuando los tipos son subtipos de los parámetros genéricos en la declaración de una colección?

Ejemplo

```
package invarianza;

import java.util.ArrayList;
import java.util.List;

public class VerificaInvarianza {
	public static void main(String[] args) {
		List<A> la;
		List<C> lc = new ArrayList<C>();
		List<D> ld = new ArrayList<D>();

		 //si lo siguiente fuese posible...
		 la = lc;
		 la.add(new C());

		 //lo siguiente también debería ser posible...
		 la = ld;
		 la.add(new D());

		 //por tanto...
		 C sa = ld.get(0); //¡No!!
	}
}
```

La asignación incorrecta en este caso es la = lc, porque como se mencionó anteriormente, si bien C y D son subclases de A, ArrayList<C> y ArrayList<D> no son subclases de List<A>. A lo sumo se puede afirmar que, por el principio de invarianza antes mencionado:

- ArrayList<A> es subclase de List<A>
- ArrayList<C> es subclase de List<C>
- ArrayList<D> es subclase de List<D>

Para que la garantía de seguridad de tipo siempre sea válida, debe ser imposible asignar una colección de un tipo a una colección de un tipo distinto, incluso si el segundo tipo es una subclase del primero.

Esto va en contra del polimorfismo tradicional y, a primera vista, parece restar flexibilidad a las colecciones genéricas.

Covarianza

Los comodines ofrecen una cierta flexibilidad al trabajar con colecciones genéricas. En el ejemplo a continuación se puede observar su uso.

Ejemplo

```
package covarianza;
import java.util.List;
import java.util.ArrayList;

public class VerificaCovarianza {
	public static void main(String[] args) {
		List<C> lc = new ArrayList<C>();
		List<D> ld = new ArrayList<D>();

		imprimeNombreClase(lc);
		imprimeNombreClase(ld);

		// pero...
		List<? extends Object> lo = lc; // Bien
		lo.add(new C());//¡Error de compilación!
	}

	public static void imprimeNombreClase(List<? extends A> la) {
		for (int i = 0; i < la.size(); i++) {
			System.out.println(la.get(i).getClass());
		}
	}
}
```

El método imprimeNombreClase se declara con un argumento que incluye un comodín. El comodín "**?**" de List <? **extends** A> podría interpretarse como "cualquier tipo de lista de elementos desconocidos que sean de tipo A o de una subclase de A". El límite superior (A) indica que los elementos de la colección pueden asignarse de forma segura a una variable A. Por tanto, las dos colecciones de los subtipos A pueden pasarse al método imprimeNombreClase.

Esta respuesta de la covarianza está designada para su lectura, no para que se escriba en ella. Debido al principio de la invarianza, es ilegal agregar a una colección que contenga un comodín con la palabra clave **extends**.

Genéricos: refactorización de código no genérico existente

Con las colecciones genéricas, es posible especificar tipos genéricos sin argumentos de tipo, denominados tipos básicos. Esta característica garantiza la compatibilidad con el código no genérico.

En el momento de la compilación, se elimina toda la información genérica del código genérico. Lo que queda es un tipo básico. Esto permite la interoperabilidad con el código tradicional, ya que los archivos de clase generados por el código genérico y por el código tradicional serían los mismos.

Durante la ejecución, ArrayList<String> y ArrayList<Integer> se traducen como ArrayList, lo cual es un tipo básico.

El uso del nuevo compilador Java SE 5.0 o posterior con código no genérico más antiguo generará una advertencia. El siguiente código muestra una clase que genera una advertencia durante la compilación.

Ejemplo

```
package advertencia;
import java.util.ArrayList;
import java.util.List;

public class AdvertenciaGenericos {
	public static void main(String[] args) {
		List lista = new ArrayList();
		lista.add(0, new Integer(42));
		int total = (Integer) lista.get(0);
	}
}
```

Si compila la clase AdvertenciaGenericos con el siguiente comando

```
javac AdvertenciaGenericos.java
```

Se observará la siguiente advertencia

```
Note: GenericsWarning.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
```

En cambio, si compila la clase con el siguiente comando

```
javac -Xlint:unchecked AdvertenciaGenericos.java
```

Se observará esta advertencia

```
GenericsWarning.java:5: warning: [unchecked] unchecked call
to add(int,E) as a member of the raw type java.util.List
list.add(0, new Integer(42));
^
1 warning
```

Aunque la clase se compila correctamente y las advertencias pueden ignorarse, es aconsejable modificar el código para que sea compatible con los genéricos. Para resolver esta advertencia en la clase AdvertenciaGenericos, es necesario cambiar la declaración para utilizar genéricos.

```
	List<Integer> lista = new ArrayList<Integer>();
```