INFO1113 Object-Oriented Programming

Week 7B: Collection interfaces
Type checking

Copyright Warning

Topics

Type parameter constraints

We saw in the previous lecture how we are able to create generic containers and utilise a type parameter within our class. However there is more we can add to this.

extends with type parameters

We can enforce constraints on what types can be used within the container. The rational we have for this is that we may want to utilise explicit functionality of a **super** type.

For example, we want to be able to create a class that contain any **Shape** class or any class that implements **Drawable**.

Since the type parameter contains extra type data associated with it, we are able to guarantee access to methods from the super type with objects associated with the type parameter.

Type Parameters

Looking back on our syntax with generics, we just simply specified an identifier for the type parameter. Now we can specify an upper bound type.

Syntax:

```
[public] class ClassName<Param0 [extends SuperType]>
```

Example:

```
public class ShoppingCart<T extends Item>
```

Type Parameters

Looking back on our syntax with generics, we just simply specified an identifier for the type parameter. Now we can specify an upper bound type.

Syntax:

```
[public] class ClassName<Param0 [extends SuperType]>
```

Example:

```
public class ShoppingCart<T extends Item>
```

All types stored within **ShoppingCart** must extend from **Item**. We are then able to use methods defined in **Item**.

Type Parameters

So let's break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayList<T>();
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double maxLitres = 0.0;
        for(T e : liquids) {
            maxLitres += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + maxLitres + "L\n");

    }
}
```

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
}

public class Water extends Liquid {

    public Water(double litres) {
        super(litres);
    }

    public String toString() { return "Water";
}

}

public class Oil extends Liquid {

    public Oil(double litres) {
        super(litres);
    }

    public String toString() { return "Oil"; }
}
```

Type Parameters

So let's break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double maxLitres = 0.0;
        for(T e : liquids) {
            maxLitres += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + maxLitres + "L\n");

    }

}
```

As part of our class definition we have included **Liquid** as our bounded type with the parameter. This infers that all types in this class must have a super type which is **Liquid**.

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }

}

public class Water extends Liquid {

    public Water(double litres) {
        super(litres);
    }

    public String toString() { return "Water";
}

}

public class Oil extends Liquid {

    public Oil(double litres) {
        super(litres);
    }

    public String toString() { return "Oil"; }
}
```

Type Parameters

So let’s break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayList<T>();
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double maxLitres = 0.0;
        for(T e : liquids) {
            maxLitres += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + maxLitres + "L\n");

    }

}
```

This allows us to store any **T** type that is specified, this means that this barrel may only be used for **Water, Oil or Both** but this is defined by the user.

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
}

                   tends Liquid {

                   le litres) {
                   ;

    public String toString() { return "Water";
}

}

public class Oil extends Liquid {

    public Oil(double litres) {
        super(litres);
    }

    public String toString() { return "Oil"; }
}
```

Type Parameters

So let's break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double maxLitres = 0.0;
        for(T e : liquids) {
            maxLitres += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + maxLitres + "L\n");

    }

}
```

Since we have a **bounded type** we are able to infer that all types have a super type **Liquid** therefore we are able to utilise the methods defined in liquid.

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }

}

public class Water extends Liquid {

    public Water(double litres) {
        super(litres);
    }

    public String toString() { return "Water";
}

}

public class Oil extends Liquid {

    public Oil(double litres) {
        super(litres);
    }

    public String toString() { return "Oil"; }
}
```

Type Parameters

So let's break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }
```

As we can demonstrate here, we hve three separate instances that strictly contain each type or with the mixed one, any type that extends from Liquid.

```java
public static void main(String[] args) {
    Barrel<Water> waterBarrel = new Barrel<Water>();
    Barrel<Oil> oilBarrel = new Barrel<Oil>();
    Barrel<Liquid> mixedBarrel = new Barrel<Liquid>();

    waterBarrel.add(new Water(1.0));
    waterBarrel.add(new Water(2.0));

    waterBarrel.outputVolume();

    oilBarrel.add(new Oil(1.0));
    oilBarrel.add(new Oil(2.0));

    oilBarrel.outputVolume();

    mixedBarrel.add(new Oil(1.0));
    mixedBarrel.add(new Water(2.0));

    mixedBarrel.outputVolume();
}
```

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
```

```
> javac BarrelProgram
```

What if I was to add oil to the water barrel?

Type Parameters

So let's break this down!

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
```

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }

public static void main(String[] args) {
    Barrel<Water> waterBarrel = new Barrel<Water>();
    Barrel<Oil> oilBarrel = new Barrel<Oil>();
    Barrel<Liquid> mixedBarrel = new Barrel<Liquid>();

    waterBarrel.add(new Water(1.0));
    waterBarrel.add(new Oil(2.0));

    waterBarrel.outputVolume();

    oilBarrel.add(new Oil(1.0));
    oilBarrel.add(new Oil(2.0));

    oilBarrel.outputVolume();

    mixedBarrel.add(new Oil(1.0));
    mixedBarrel.add(new Water(2.0));

    mixedBarrel.outputVolume();
}
```

When we attempt to compile the compiler will refuse to do this as the type safety is being violated here.

```
> javac BarrelProgram
Liquid.java:21: error: incompatible types: Oil
cannot be converted to Water
     waterBarrel.add(new Oil(2.0));
                     ^
Note: Some messages have been simplified;
recompile with -Xdiags:verbose to get full output
1 error
```

Generic Arrays

We have seen how we can use type parameters with single variables but how does it work with arrays?

Not very well as we will need to perform an unsafe operation to construct an array.

Small talk on safe abstractions

Generic Arrays

Let's see what happens if we were to declare a generic array?

```
public class DynamicArray<T, K extends T> {

    private T[] array;

    public DynamicArray() {

    }

}
```

```
> javac DynamicArray
>
```

Generic Arrays

Let's see what happens if we were to declare a generic array?

```
public class DynamicArray<T, K extends T> {

    private T[] array;

    public DynamicArray() {
        array = new K[4];
    }

}
```

Okay, so it appears there is nothing to worry about. Let's write the rest of the code.

```
> javac DynamicArray
>
```

Generic Arrays

Let’s see what happens if we were to **instantiate** a generic array?

```
public class DynamicArray<T, K extends T> {

    private T[] array;

    public DynamicArray() {
       array = new K[4];
    }

}
```

```
> javac DynamicArray
DynamicArray.java:8: error: generic array
creation
    array = new K[4];
                ^

1 errors
```

Drat! We are unable to instantiate a generic array.

Let’s take a detour into arrays

So let's take a look at this small program.

```java
public static void main(String[] args) {

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

Generic Arrays

So let's take a look at this small program.

```
public static void main(String[] args) {

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

Initialising an array of strings, completely valid operation, it will be of length 2.

Generic Arrays

So let's take a look at this small program.

```
public static void main(String[] args) {

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

We are able to set **objects** to **strings** since arrays are **covariant**.

Generic Arrays

So let's take a look at this small program.

```
public static void main(String[] args) {

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

objects[0] is to be assigned an **Integer** type. Since Integer is a subtype of **Object** this operation is considered valid.

Generic Arrays

So let's take a look at this small program.

```java
public static void main(String[] args) {

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

```
> javac BrokenArrays.java
> java BrokenArrays
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
    at BrokenArrays.main(BrokenArrays.java:7)
```

However, since the original type is **String[]** we have violated a type constraint at **Runtime.**

Arrays are covariant, Generics are not!

Generic Arrays

With the following program which looks functionally similar to the Array version, we are unable to break this type constraint.

```
public static void main(String[] args) {

    List<String> strings = new ArrayList<String>();
    List<Object> objects = strings;
    objects[0] = new Integer(1); //Don't have to consider this

}
```

Generic Arrays

With the following program which looks functionally similar to the Array version, we are unable to break this type constraint.

```
public static void main(String[] args) {

    List<String> strings = new ArrayList<String>();
    List<Object> objects = strings;
    objects[0] = new Integer(1); //Don't have to consider this

}
```

Specifically here, because **generics are invariant** we are unable to perform a similar operation to arrays, this will result in a compilation error.

Okay, so how do we get around this?

Generic Arrays

Let’s dive into building a **DynamicArray** with generics.

```java
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

Generic Arrays

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

So just like week 4's DynamicArray, however we will be replacing our **int[]** type with **T[].**

Generic Arrays

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

There is a difference between declaring a generic array and instantiating a generic array.

Generic Arrays

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

There is a difference between declaring a generic array and instantiating a generic array.

Simply, this is because Arrays in java can break type safety where generics are aimed at enforcing type safety.

Generic Arrays

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

Hrmm.. okay, so we need to do something **unsafe** now.

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new (T[]) Object[4];
        size = 0;
    }

    private void resize() {
        T[] temp = (T[])new Object[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

We will need to instantiate an **Object[]** to use with array and **cast.**

Iterators

We have seen an example of the for-each loop in prior lectures. We will be looking into how we are able to implement the same behaviour on our own data structures.

An iterator is object that allows reading through a collection. It maintains state within the collection and where to go next.

Iterators

Interestingly, prior to Java 5, the language did not have a language construct involving **for-each** (or enhanced for loop).

However, iterators existed prior and therefore there exists a pattern that the **for-each** loop just translates into.

Iterators

We have seen iterators in use within Java when we utilise the **for-each** loop.

```
ArrayList<String> list = new ArrayList<String>();
for(String s : list) {
    System.out.println(s);
}
```

But we need to consider the equivalence of this operation.

Iterators

We have seen iterators in use within Java when we utilise the **for-each** loop.

```
ArrayList<String> list = new ArrayList<String>();
for(String s : list) {
    System.out.println(s);
}
```

If we were to grab an item outside of the for-each loop, we would need to use **get(). How is the for-each loop doing this?**

But we need to consider the equivalence of this operation.

Breakdown of iterators

Here is our translation of a **for-each** loop, it is returning an iterator, using **hasNext()** and **next()** methods.

```
Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String s = iterator.next(); //Returns element and moves it
    System.out.println(s);
}
```

We have access to an iterator object which in turn is used within a while loop. It will check that there is an element it can access next before iterating.

But this is Java, these methods must exist somewhere!

We can see the iterator has access to both hasNext() and next(), hasNext() checks, next() will return the next element in the collection.

Iterable

The iterable interface primarily declares an **iterator()** method to be defined by the collection type. The returned iterator will be an object that can be utilised in a **for-each** loop.

The iterator returned typically reflects the type that is utilised within the collection type. As per the language specification, the compiler will check if the collection has implemented **iterable** interface.

Iterable

Let’s take a look at the iterable interface.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default void`	`forEach(Consumer<? super T> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	`iterator()` Returns an iterator over elements of type `T`.
`default Spliterator<T>`	`spliterator()` Creates a `Spliterator` over the elements described by this `Iterable`.

We can see that it requires implementing a method called **iterator()** which will return **Iterator<T>** object.

Iterable

Let's take a look at the iterable interface.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
default void	forEach(Consumer<? super T> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
Iterator<T>	iterator() Returns an iterator over elements of type T.
default Spliterator<T>	spliterator() Creates a Spliterator over the elements described by this Iterable.

We can see that it requires implementing a method called **iterator()** which will return **Iterator<T>** object.

Considering any class can create an iterator we will need to specifically mark types with **Iterable** for them to work with a for-each loop.

Hrmm… it requires another type!

Iterator type

We can check out the iterator type from the java documentation and understand what methods compose an **iterator.**

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`default void`	`forEachRemaining(Consumer<? super E> action)` Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`default void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Iterator type

We can check out the iterator type from the java documentation and understand what methods compose an **iterator.**

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`default void`	`forEachRemaining(Consumer<? super E> action)` Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`default void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

We can see that we have **hasNext()** and **next()** methods to define in our iterator class.

Iterator type

We can check out the iterator type from the java documentation and understand what methods compose an **iterator.**

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
default void	forEachRemaining(Consumer<? super E> action) Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.
default void	remove() Removes from the underlying collection the last element returned by this iterator (optional operation).

We can see that we have **hasNext()** and **next()** methods to define in our iterator class.

Simply, one is used for checking that there is an element and the other will return the element.

Okay, but why would I prefer this over a for-loop and indexes?

Let’s transform our Linked List

Welcome back Linked List! We’re going to make it iterable!

```java
public LinkedList<T>{

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

Let's transform our Linked List

Welcome back Linked List! We're going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

We have specified that this **Linked List** will implement Iterable.

Welcome back Linked List! We’re going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

We have specified that this **Linked List** will implement Iterable.

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

Okay, but what iterator do we use?

Let's transform our Linked List

Welcome back Linked List! We're going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

We will create our own iterator that we will use in a for-each loop.

```
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public T next() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }

}
```

Let’s transform our Linked List

Welcome back Linked List! We’re going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

```
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public T next() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }

}
```

We contain a variable called cursor which will allow us to **move** through the collection.

Let's transform our Linked List

Welcome back Linked List! We're going to make it iterable!

```java
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v
        } else {
            Node<T> current = he
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

```java
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public T next() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }

}
```

As we saw before, we define out own **hasNext()** method

Let’s transform our Linked List

Welcome back Linked List! We’re going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

```
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public T next() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }
```

We write next() to return the **next value** while changing the cursor.

Let’s transform our Linked List

Welcome back Linked List! We’re going to make it iterable!

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator(head);
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

We update our **iterator()** method to return a **LinkedListIterator** object.

```
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        n cursor != null;


        ext() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }

}
```

See you next time!