# INFO1113 / COMP9003
# Object-Oriented Programming

## Lecture 7

THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

# Acknowledgement of Country

*I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.*

*I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.*

# Copyright Warning

# Topics: Part A

- **Generics**
- **Generics in Data Structures**
- **Generic class and UML**

```
ArrayList<Integer> aList = new ArrayList<Integer>();

ArrayList<Double> dList = new ArrayList<Double>();

ArrayList<String> sList = new ArrayList<String>();
```

# Generics

**Motivation:**

Without generics, we will need to duplicate the class multiple times for different types.

Or we would be required to perform casting between objects (if we were to store them as an **Object** type within the data structure).

Refer to Chapter 12.1, pages 905-914, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# Generics (Case 1)

**Let's say we live in a world without generics.** Let's go through two scenarios where we want to implement an **ArrayList** for every type we use in our program.

It becomes apparent that we will be duplicating code for every data type, we would need to create:

- **IntArrayList**
- **DoubleArrayList**
- **FloatArrayList**
- **StringArrayList**

This is awful, now we're maintaining duplicate code for a change in data type.

# Generics (Case 2)

So as we have learned from previous lectures that all **reference types** inherit from **Object**. We could treat all instances as **Object.**

**Cool, we only need to write it once.**

We effectively generalise all instances to an **Object[]** that will contain each **type.**

**Hang on! If everything is Object, how do we know what types is stored?**

# Generics (Case 2)

The compiler wouldn't know what is stored in the **Container**, it could be any kind of element, however this isn't the only issue.

**Let's use the container as an example:**

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

So, looking at set, how would we know what type is being added?

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

```
public class Container {

  private Object element;

  public Container(Object element) {
    this.element = element;
  }

  public Object set(Object element) {
    Object oldElement = this.element;
    this.element = element;
    return oldElement;
  }

  public Object get() {
    return element;
  }

}
```

# Generics (Case 2)

The compiler wouldn't know what is stored in the **Container**, it could be any kind of element, however this isn't the only issue.

**Let's use the container as an example:**

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

1. Container c = new Container("Hello Box!");
2. String s1 = c.get();
3. c.set("New string here!");
4. String s2 = c.get();

5. System.out.println(s1);
6. System.out.println(s2);

}
```

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

```
public class Container {

   private Object element;

   public Container(Object element) {
      this.element = element;
   }

   public Object set(Object element) {
      Object oldElement = this.element;
      this.element = element;
      return oldElement;
   }

   public Object get() {
      return element;
   }

}
```

# Generics (Case 2)

The compiler wouldn't know what is stored in the **Container**, it could be any kind of element, however this isn't the only issue.

**Let's use the container as an example:**

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

    Container c = new Container("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);
```

```
Container.java:26: error: incompatible types: Object cannot be converted to String
        String s1 = c.get();
                         ^
Container.java:28: error: incompatible types: Object cannot be converted to String
        String s2 = c.get();
                         ^
2 errors
```

```java
public class Container {

  private Object element;

  public Container(Object element) {
    this.element = element;
  }

  public Object set(Object element) {
    Object oldElement = this.element;
    this.element = element;
    return oldElement;
  }

  public Object get() {
    return element;
  }

}
```

# Generics (Case 2)

The compiler wouldn’t know what is stored in the **Container**, it could be any kind of element, however this isn’t the only issue.

**Let’s use the container as an example:**

```
public class Container {

  private Object element;

  public Container(Object element) {
    this.element = element;
  }

  public Object set(Object element) {
    Object oldElement = this.element;
    this.element = element;
    return oldElement;
  }

  public Object get() {
    return element;
  }

}
```

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

  Container c = new Container("Hello Box!");
  String s1 = c.get();
  c.set("New string here!");
  String s2 = c.get();

  System.out.println(s1);
  System.out.println(s2);
```

Since this is a **String** variable and the return type is **Object**. The compiler cannot guarantee type correctness here.

```
Container.java:26: error: incompatible types: Object cannot be converted to String
        String s1 = c.get();
                         ^
Container.java:28: error: incompatible types: Object cannot be converted to String
        String s2 = c.get();
                         ^
2 errors
```

# Generics (Case 2)

The compiler wouldn't know what is stored in the **Container**, it could be any kind of element, however this isn't the only issue.

**Let's use the container as an example:**

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

   Container c = new Container("Hello Box!");
   String s1 = (String) c.get();
   c.set("New string here!");
   String s2 = (String) c.get();

   System.out.println(s1);
   System.out.println(s2);
```

```
> javac ContainerProgram.java
>
```

We will need to **cast** the object to the correct type. This is a **runtime check** and can lead to unsafe assumptions.

```
public class Container {

   private Object element;

   public Container(Object element) {
      this.element = element;
   }

   public Object set(Object element) {
      Object oldElement = this.element;
      this.element = element;
      return oldElement;

         Object get() {
         rn element;
   }
}
```

# Generics

As part of usage with collection classes, you have been able to specify a type that will be contained within the collection.

Generics gives us the ability to handle multiple different types without needing to rewrite the same code.

Refer to Chapter 12.1, pages 905-914, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# Generics

The advantages of generics

- Stronger type checks at compile time
- Elimination of casts
- Enabling programmers to implement generic algorithms

# Generics

**Generics** are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class.

**Syntax:**

[public] class <u>ClassName</u><Param0[,Param1..]>

**Example:**

```java
public class Container<T>
```

# Generics

**Generics** are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class.

**Syntax:**

[public] class <u>ClassName</u><Param0[,Param1..]>

**Example:**

```
public class Container<T>
```

We have specified a type parameter here. This allows us to create a variable to represent the type within our class.

We are not limited to just one type variable as we can specify many as we want. **However**, only utilise generics when necessary.

# Generics in classes

We will use the generic identifier within our class so we can annotate methods and variables with it. This allows the method to be annotated with the generic variable.

**Syntax:**

[public] [static] **T** methodName()

[public] [static] **void** methodName(**T** parameter)

**T** variable

Type<T> variable

# Generic Container

In the example below we will be writing a container that will store any type we want.

```
public class Container {

  private int element;

  public Container(int element) {
    this.element = element;
  }

  public int set(int element) {
    int oldElement = this.element;
    this.element = element;
    return oldElement;
  }

  public int get() {
    return element;
  }
}
```

```
public class Container<T> {

  private T element;

  public Container(T element) {
    this.element = element;
  }

  public T set(T element) {
    T oldElement = this.element;
    this.element = element;
    return oldElement;
  }

  public T get() {
    return element;
  }
}
```

We have defined T as our **type parameter**. This allows us to use this identifier through out our class

We can use the type parameter as a **data type** for our variable.

We can use the type parameter as the return type and data type of the parameter

# Generic Container

In the example below we will be writing a container that will store any type we want.

```java
public class Container<T> {

  private T element;

  public Container(T element) {
    this.element = element;
  }

  public T set(T element) {
    T oldElement = this.element;
```

We are now utilising our generic **Container** class with a **String** type.

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

  Container<String> c = new Container<String>("Hello Box!");
  String s1 = c.get();
  c.set("New string here!");
  String s2 = c.get();

  System.out.println(s1);
  System.out.println(s2);
}
```

```
> java ContainerProgram
Hello Box!
New string here!
<Program End>
```

# Generic Container

In the example below we will be writing a container that will store any type we want.

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
```

We are able to infer what type is going to be used through the generic identifier.

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

    Container<String> c = new Container<String>("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);
}
```

```
> java ContainerProgram
Hello Box!
New string here!
<Program End>
```

# Generic container demo

# Data Structures

We explored collections in week 4. Generics are typically used within this area as the operations and patterns involved do not differ based on the type that is used.

A linked list that contains **integers** does not have different operations to a linked list that contains **doubles**.

# Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```
public class 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;
  }
}
```

```
public class Node<T> {

  private T value;
  private Node<T> next;

  public Node(T v) {
    value = v;
    next = null;
  }

  public Node<T> getNext() {
    return next;
  }

  public void setNext(Node<T> n) {
    next = n;
  }

  public T getValue() {
    return value;
  }

  public void setValue(T v) {
    value = v;
  }

}
```

# Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```
public class 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;
  }
}
```

We typically **provide a type argument** when we use a **collection.** The T parameter is used through the class.

```
public class Node<T> {

  private T value;

  }

  public Node<T> getNext() {
    return next;
  }

  public void setNext(Node<T> n) {
    next = n;
  }

  public T getValue() {
    return value;
  }

  public void setValue(T v) {
    value = v;
  }

}
```

# Usage of generics in data structures

Let's go over the use of generics with a **Linked List** that we saw in week 4.

```
public class 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;
  }
}
```

```
public class Node<T> {

  private T value;
  private Node<T> next;

  public Node(T v) {
    value = v;
    next = null;
  }

  public Node<T> getNext() {
    return next;
  }

  public void setNext(Node<T> n) {
    next = n;
  }

  public T getValue() {
    return value;
  }

  public void setValue(T v) {
    value = v;
  }

}
```

We can use it not only as part of class and instance attributes but part of method variable.

# Generics and collections demo

# What about static methods?

# Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

**Syntax:**

[public] static <Param0[,Param1..]> **return_type** methodName([,Param1..])

**Example:**

**public static** <T> T **find**(T needle, T[] haystack)

**Usage:**

Points.**<AbsolutePoint>**find(point, points);

# Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Define a type parameter to be used in our method, we can also provide a type bound here

**Syntax:**

[public] static <Param0[,Param1..]> **return_type** methodName([,Param1..])

Type parameter can be used as return type and method parameter type

**Example:**

```
public static <T> T find(T needle, T[] haystack)
```

**Usage:**

Since the method can be used without an instance, the type argument needs to known

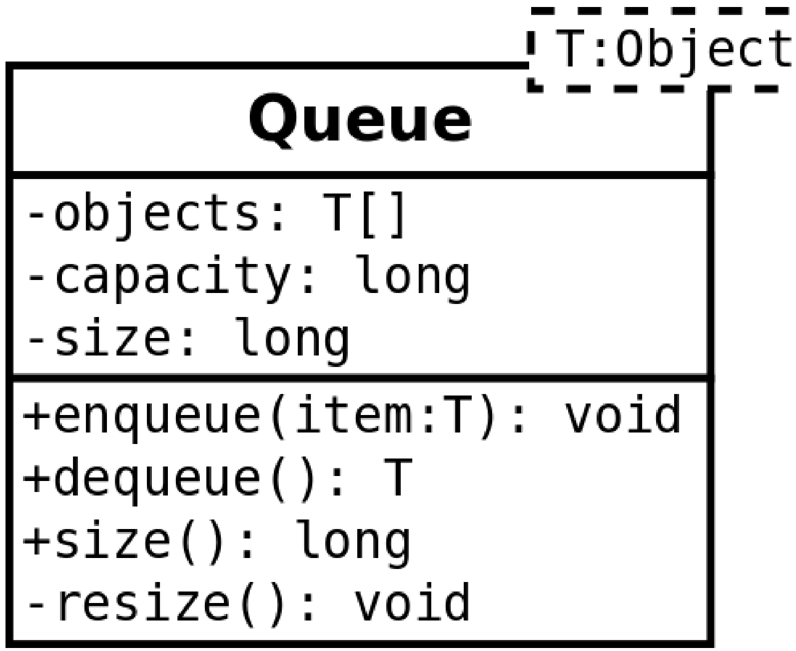
```
Points.<AbsolutePoint>find(point, points);
```

We pass the type argument to the static method when we want to invoke it. This type argument can be used to ensure a method parameter has a type association

# UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

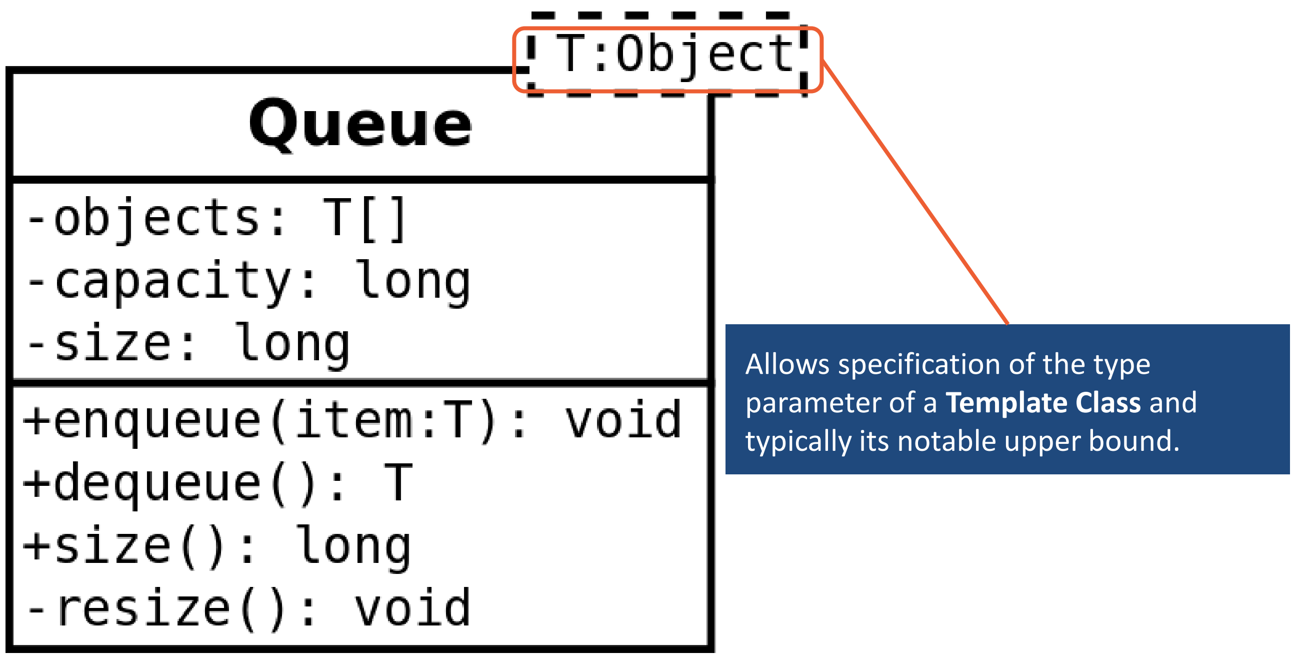
Within the visualisation it will contain annotation of the type parameter.



# UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

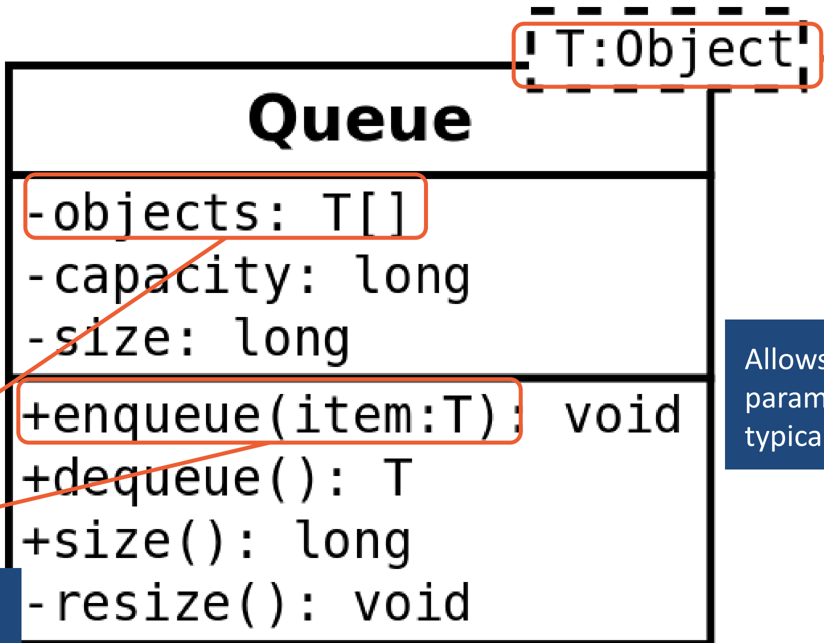
Within the visualisation it will contain annotation of the type parameter.



# UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

Within the visualisation it will contain annotation of the type parameter.




Allows specification of the type parameter of a **Template Class** and typically its notable upper bound.

Similar to our class definition, we use the type parameter for attributes and methods.

Let’s take a break!

# Topics

- **Bounded Type Parameters**
- **Generic Arrays**
- **Iterator and Iterable**

# Type parameter constraints

We saw in the previous section, how we can 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 **Talk()**.

# 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 <u>ClassName</u><Param0 [extends <u>SuperType</u>]>

**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 total = 0.0;
        for(T e : liquids) {
            total += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + total + "L\n");

    }

}
```

```java
public 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!

```
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 total = 0.0;
      for(T e : liquids) {
         total += e.getLitres();
         System.out.println(e + ": " + e.getLitres() + "L");
      }
      System.out.println("Total: " + total + "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**.

```
public class Liquid {

   private double litres;

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

   public double getLitres() {
      return litres;



ic 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!

```
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 total = 0.0;
    for(T e : liquids) {
      total += e.getLitres();
      System.out.println(e + ": " + e.getLitres() + "L");
    }
    System.out.println("Total: " + total + "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.

```
public 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 total = 0.0;
        for(T e : liquids) {
            total += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + total + "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 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!

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

   private List<T> liquids;

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

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();
}
```

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

```
public class Liquid {

   private double litres;

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

   ublic 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!

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

    private List<T> liquids;

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

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();
}
```

```
public class Liquid {

    private double litres;

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

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

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
```

# Demo

# 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.**

# Generic Arrays

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

```
public class DynamicArray<T> {

    private T[] array;

    public DynamicArray() {

    }
}
```

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 declare a generic array?

```
public class DynamicArray<T> {

    private T[] array;

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

# Generic Arrays

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

```
public class DynamicArray<T> {

    private T[] array;

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

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

**oops!** We are unable to instantiate a generic array.

**Okay, so how do we get around this?**

# Generic Arrays

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

```
public class DynamicArray<T> {

    private T[] array;

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

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

```
> javac DynamicArray
```

# Demo

# Iterators

We have seen examples of **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.

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

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

# Iterators

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

```
LinkedList<String> list = new LinkedList<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 when we utilise the **for-each** loop.

```
LinkedList<String> list = new LinkedList<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

# 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)`<br>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()`<br>Returns an iterator over elements of type `T`. |
| `default Spliterator<T>` | `spliterator()`<br>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)<br>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()<br>Returns an iterator over elements of type T. |
| default Spliterator<T> | spliterator()<br>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.

**Hmm… 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)`<br>Performs the given action for each remaining element until all elements have been processed or the action throws an exception. |
| `boolean` | `hasNext()`<br>Returns `true` if the iteration has more elements. |
| `E` | `next()`<br>Returns the next element in the iteration. |
| `default void` | `remove()`<br>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)<br>Performs the given action for each remaining element until all elements have been processed or the action throws an exception. |
| boolean | hasNext()<br>Returns true if the iteration has more elements. |
| E | next()<br>Returns the next element in the iteration. |
| default void | remove()<br>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)`<br>Performs the given action for each remaining element until all elements have been processed or the action throws an exception. |
| `boolean` | `hasNext()`<br>Returns `true` if the iteration has more elements. |
| `E` | `next()`<br>Returns the next element in the iteration. |
| `default void` | `remove()`<br>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.

# Let’s transform our Linked List

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

```
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.

# 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;
    }
}
```

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?

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

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

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

  private Node<T> cursor;

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

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


  ublic 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;
   }

}
```

As we saw before, we define our 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<T>(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() {

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

    return element;
  }

}
```

# Demo

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

**See you next time!**

THE UNIVERSITY OF SYDNEY