# Generics

- Lists
- Arrays vs Lists
- Creating a list class
- Introduction to Generics (Continued next week)

## Lists

- A list is a collection that stores data like an array
- Like an array, every element is stored in an integer index that stores its position in the sequence
- However, unlike an array it can store any number of elements and the number of elements do not have to be declared when the list is created

## Array reminder

- An array must have its size declared when it's created:

```
int[] intArray = new int[4];
```

- The same applies to the shorthand syntax, although the size is worked out by the compiler

```
int[] intArray2 = {1,2,3,4};
```

- Both arrays have a fixed length of 4 (a maximum index of 3). This length can never be altered after the array is created. You can never write to intArray[4]

## Arrays vs lists

- An array has a special syntax using square brackets
- Lists are classes/objects so use the object syntax with dots and method names.
- **Lists are similar to arrays but do not use the square bracket syntax**
- **Lists are just classes and have no special syntax**

## Predecessors and Successors

- Every element in the list has a numerical index
- The first element in the list has index = 0
- Every element in the list (other than the first) has a <u>predecessor</u>: The element immediately before it in the list
- Every element in the list (other than the last) has a <u>successor</u>: the element immediate after it in the list

## An example list  for Strings

- You can easily write a class to act as a list
- It will need at minimum, methods for the following:
  - **add(String element)** – to add an item to the list
  - **length()** to get the length of the list
  - **get(int index)** – Retrieve the element at a given index

# Building the StringList Class

```
public class StringList {
    private String[] strings = new String[999];
    private int length = 0;


    public void add(String item) {
        this.strings[this.length] = item;
        this.length++;
    }

    public String get(int index) {
        return this.strings[index];
    }

    public int length() {
        return this.length + 1;
    }
}
```

An array larger
than ever needed

An integer to store the
number of elements in the
list

When an item is added,
Add it to the next available
Index and increment the
counter

# StringList

- It's now possible to use the list like this:

```java
public static void main(String[] args) {
    StringList list = new StringList();

    list.add("A");
    list.add("B");
    list.add("C");
    list.add("D");

    for (int i = 0; i < list.length(); i++) {
        System.out.println(list.get(i));
    }

}
```

```
Output:
A
B
C
D
```

## StringList

- You can then add additional features to the original class. Some methods that may be useful:
  - **<u>remove(int index)</u>** remove an item at a given index
  - **<u>contains(String str)</u>** returns true/false depending on whether the String *str* is stored in the list
  - **<u>clear()</u>** empty the list

## Implementing contains

- To implement the contains method, all of the current elements need to be looped though and compared against the value:

```java
public boolean contains(String str) {
    for (int i = 0; i < length; i++) {
        if (strings[i].equals(str)) return true;
    }
    return false;
}
```

```java
StringList list = new StringList();

list.add("A");
list.add("B");
list.add("C");
list.add("D");

if (list.contains("C")) {
System.out.println("List contains C");
}
```

## Implementing clear

- To clear the list, the string array must be re-initialised and the length reset to zero

```
public void clear() {
    this.strings = new String[999];
    this.length = 0;
}
```

```
StringList list = new StringList();

list.add("A");
list.add("B");
list.add("C");
list.add("D");

System.out.println(list.length()); //4

list.clear();

System.out.println(list.length()); //0
```

# Removing an element

- To remove an element by its index, all elements in the list after the one being removed must be shifted up. Consider the list:
  - 0: A
  - 1: B
  - 2: C
  - 3: D
- If “B” is removed then “C” must be moved into position 1, and “D” must be moved into position
- Finally the number of elements must be reduced by 1

## Removing an element

```java
public void remove(int index) {
    for (int i = index+1; i < length; i++) {
        strings[i-1] = strings[i];
    }
    strings[length] = null;
    length--;
}
```

```java
StringList list = new StringList();

list.add("A");
list.add("B");
list.add("C");
list.add("D");


for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}

list.remove(1);

for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}
```

```
Output:
A
B
C
D

A
C
D
```

## StringList

- The list now includes most of the functionality that will be useful. However it has two major limitations
  - 1) It cannot store more than 999 elements because it's based on an array with a fixed size
  - 2) It can only store strings

## Problem 1: Limited values

- The problem of only storing up to 999 values can be overcome.
- An array cannot have its size changed
- However, a new array can be created with a larger size then the old array values can be copied into the new array
- A resize method can be added to the StringList to resize the array when it reaches its capacity

## Problem 1: Limited size

```
private void resize() {
    String[] newArray = new String[strings.length*2];

    for (int i = 0; i < length; i++) {
        newArray[i] = strings[i];
    }

    strings = newArray;
}
```

Create a new array twice as large as the old one

Copy all the elements from the old array into the new one

Overwrite the old array with the new one

```
public class StringList {
    private String[] strings = new String[2];
    private int length = 0;


    public void add(String item) {
        if (length == strings.length) resize();
        this.strings[this.length] = item;
        this.length++;
    }
```

Initialise the array with a more sensible starting size

Call the resize method if the initial array is full

## Problem 2) The list can only store Strings

- The StringList class has been designed around Strings to store strings
- If I wanted to make an IntegerList I'd have to copy/paste the entire class and replace all occurrences of String with Integer
- The underlying logic for the list is the same for any type it's only the type that is different

# Problem 2) The list can only store strings

```java
public class StringList {
    private String[] strings = new String[2];
    private int length = 0;

    public void add(String item) {
        if (length == strings.length) resize();
        this.strings[this.length] = item;
        this.length++;
    }

    public String get(int index) {
        return this.strings[index];
    }

    public boolean contains(String str) {
        for (int i = 0; i < this.length; i++) {
            if (this.strings[i].equals(str)) return true;
        }
        return false;
    }

    public void clear() {
        this.strings = new String[999];
        this.length = 0;
    }
```

```java
public class IntList {
    private int[] ints = new int[2];
    private int length = 0;

    public void add(int item) {
        if (length == ints.length) resize();
        this.ints[this.length] = item;
        this.length++;
    }

    public int get(int index) {
        return this.ints[index];
    }

    public boolean contains(int str) {
        for (int i = 0; i < this.length; i++)
            if (this.ints[i].equals(str)) ret
        }
        return false;
    }

    public void clear() {
        this.ints = new int[999];
        this.length = 0;
    }
```

## Problem 2) The list can only store Strings

- This would have to be done for any type I wanted to store
  - Double
  - Boolean
  - Char
  - Float
  - Person
  - Employee
  - Vehicle
  - Etc!
- This would involve a lot of repeated code. To add a feature to the list it would have to be added to every list class.

## Generics

- Java provides a mechanism for allowing classes to work with any type
- This is called a *Generic*
- Generics can be used to overcome the problem of repeated code and allow methods to be used on any type

## Defining Generics

- When declaring a class as a a Generic you use Angle Brackets.
- The class is defined using the header

```
public class List<T> {
```

- The <T> in brackets can be thought of as a variable name. You can call this anything you like

## Defining Generics

- You can then use the variable name in your class anywhere you would normally use a type:

```
public class List<T> {
    private Object[] items = new Object[2];
    private int length = 0;

    public void add(T item) {
        if (length == items.length) resize();
        this.items[this.length] = item;
        this.length++;
    }


    public T get(int index) {
        return (T) this.items[index];
    }
```

# Generics

- This allows you to define the type when you initialise the list inside the angle brackets:

```java
List list = new List<String>();

list.add("A");
list.add("B");
list.add("C");
list.add("D");

for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}
```

```java
List list = new List<Integer>();

list.add(1);
list.add(2);
list.add(3);
list.add(4);

for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}
```

```java
List list = new List<Boolean>();

list.add(false);
list.add(true);
list.add(true);
list.add(false);

for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}
```

```java
List list = new List<Double>();

list.add(12.3);
list.add(9.887);
list.add(22.3);
list.add(1.093);

for (int i = 0; i < list.length(); i++) {
    System.out.println(list.get(i));
}
```

## Generics

- This allows you to define the class once and reuse it with any type
- This has the advantage of making your class more reusable
- However there is one caveat: Generics only support objects! Note that the type in the angle brackets start with uppercase letters!

```
List list = new List<String>();
List list = new List<Integer>();
List list = new List<Boolean>();
List list = new List<Double>();
```

## Generics

- When you declare an primitive in java you use lowercase "int", "double", "boolean", etc

```
int myInt = 9;
double myDobule = 34.2;
boolean myBoolean = false;
```

- When using Generics you need to use the upper-case variant!

```
List list = new List<Integer>();
List list = new List<Boolean>();
List list = new List<Double>();
```

## Generics

- This is a poorly designed part of Java created to overcome some of the problems with generics.
- Java supplies a class for each primitive
- These are *wrapper* classes that can be used to store primitives as objects
- You can (usually!) treat them interchangeably so you can think of them as different names for the same thing
- This conversion is done transparently and requires no specific code by the programmer

## Wrappers

```
double x = 99.9;
Double wrapper = x;

Double wrapper2 = 77.8;
double y = wrapper2;
```

- This is known as "unboxing" and "boxing". When java does this itself it is known as "Autoboxing"
- You should never need to declare the wrapper class other than when defining a generic

# Wrappers

- Java provides a wrapper class for each primitive

| Primitive Type | Wrapper Class |
|---|---|
| byte | Byte |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

## Generics

- Consider a box class that represents a rectangle with a width and a height
- You may not know the dimensions being used up front

## Generic box class

```
public class Box<T> {
    private T height;
    private T width;

    public Box(T width, T height) {
        this.width = width;
        this.height = height;
    }
    public T getWidth() {
        return width;
    }
    public T getHeight() {
       return height;
    }
    public String toString() {
        return "Width: " + width + " height: " + height;
    }
}
```

The constructor uses the Generic type

## Generic box class

- This way, the box class can be used with any type:

```
Box<Integer> box1 = new Box<Integer>(4, 5);

System.out.println(box1.toString());
```

```
Output:
Width: 4 height: 5
```

Because the constructor is defined as requiring type T, You must pass arguments that match the Generic type (in this case Integer)

## Generic box class

- This way, the box class can be used with any type:

```
Box<Double> box1 = new Box<Double>(4.2, 5.3);

System.out.println(box1.toString());
```

```
Output:
Width: 4.2 height: 5.3
```

By changing the generic
Type, the arguments
Must also be changed

## Generic box class

- This way, the box class can be used with any type:

```
Box<String> box1 = new Box<String>("14cm", "22cm");

System.out.println(box1.toString());
```

```
Output:
Width: 14cm height: 22cm
```

You could do the same with Strings

## Generic names

- The type T is used by convention but you can use anything.
- The conventions are as follows:

| Name | Usual Meaning |
|---|---|
| T | Used for a generic type. |
| S | Used for a generic type. |
| E | Used to represent generic type of an element in a collection. |
| K | Used to represent generic type of a key for a collection that maintains key/value pairs. |
| V | Used to represent generic type of a value for collection that maintains key/value pairs. |

## Generic objects as arguments

- When declaring a method argument that takes a generic type, you **must** provide the full class name including the type

```
public static void main(String[] args) {
        Box<String> box1 = new Box<String>("12cm",  "4cm");
        showBoxWidth(box1);
    }

    public static void showBoxWidth(Box<String> box) {
        System.out.println("Box has a width of " +  box.getWidth());
    }
```

## Generic objects as arguments

- However, arguments only work with the provided types, this will cause an error

```
public static void main(String[] args) {
        Box<String> box1 = new Box<String>("12cm",  "4cm");
        showBoxWidth(box1);
    }

    public static void showBoxWidth(Box<Integer> box) {
        System.out.println("Box has a width of " +  box.getWidth());
    }
```

## Wildcard parameters

- There is a wildcard parameter which allows you to pass in a generic of any type.
- Using a ? character inside the angle brackets allows parameters of any generic type to be used

```java
public static void main(String[] args) {
    Box<String> box1 = new Box<String>("12cm", "4cm");
    showBoxWidth(box1);
    Box<Integer> box2 = new Box<Integer>(12, 4);
    showBoxWidth(box2);
}

public static void showBoxWidth(Box<?> box) {
    System.out.println("Box has a width of " + box.getWidth());
}
```

## Wildcard parameters

- When using wildcards you need to be careful.

```
public static void showBoxWidth(Box<?> box) {
    int width = box.getWidth();
}
```

This will cause an error because getWidth() could return a string or a double instead of an int

## Wildcard parameters

- You can cast the type to force a conversion

```
public static void showBoxWidth(Box<?> box) {
    int width = (int) box.getWidth();
}
```

This will attempt to convert
the return value into
an integer. However,
it may cause a runtime
error if getWidth() returns
a string

## Wildcard parameters

- Because of this, wildcard  parameters are generally discouraged unless you include try/catch blocks to deal with potential errors

```
public static void showBoxWidth(Box<?> box) {
         try {
              int width = (int) box.getWidth();
         }
         catch (Exception e) {

         }
    }
```

## Inbuilt Generics

- Java provides several inbuilt generic classes that solve some of the problems from earlier.
- One of use to us is ArrayList
- This is essentially the List implementation from the beginning of this lecture inbuilt in java.
- To use the inbuilt ArrayList class, import the class

```
import java.util.ArrayList;
```

## ArrayList

- You can use the ArrayList class by specifying the type you want to store in the angle brackets
- To add elements to the list, call the *add* method on it

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

list.add("Zero");
list.add("One");
list.add("Two");
list.add("Three");
```

# ArrayList

- You can then read back items from the list using the .**get** method
- Like arrays, indexes start from Zero

```java
ArrayList<String> list = new ArrayList<String>();

list.add("Zero");
list.add("One");
list.add("Two");
list.add("Three");

System.out.println(list.get(2)); //"Two"
```

## Using ArrayList

- To get the length of the array list call the **size()** method.
- Note: This is another inconsistency in Java. A string's length is a method called length: str.length(); An array's length is a property called length: array.length; and an ArrayList's length is a method called size()

```java
ArrayList<String> list = new ArrayList<String>();

list.add("Zero");
list.add("One");
list.add("Two");
list.add("Three");

System.out.println(list.size()); // Prints “4”
```

## Using ArrayLists

- To loop through an ArrayList you can use its size in a for loop:

```java
ArrayList<String> list = new ArrayList<String>();

list.add("Zero");
list.add("One");
list.add("Two");
list.add("Three");


for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

- *Note: There is another method using iterators which will be covered next week*

# Using an ArrayList

- ArrayLists have a *contains* method that returns true/false depending on whether the list contains the value being searched for

```java
ArrayList<String> list = new ArrayList<String>();

list.add("Zero");
list.add("One");
list.add("Two");
list.add("Three");

if (list.contains("One")) {
    System.out.println("List contains One");
}
else {
    System.out.println("List does not contain One");
}
```