Interfaces and Generics + Intro to ArrayList

Objectives

- Create your own interfaces
- Create multiple implementations of those interfaces
- Understand how generics are used in Java
- Be able to create and implement generic interfaces
- Implement generic classes
- (Maybe) how to implement ArrayList

Objectives

```
List<String> strList = new ArrayList<>();
List<Integers> intList = new LinkedList<>();

class NewClass implements Comparable<NewClass>{
    @Override
    public int compareTo(OtherClass oc){
        // ...
    }
}
```

- You've seen two interfaces previously:
  - The List interface in week 1.
  - Implementing the Comparable interface in week 2.
- We'll be drawing many comparisons to List in particular
- We'll build towards a simplified version of how the List interface and it's implementations are build so you can make your own!

Defining an Interface

```
interface List{
    public boolean add(Object value);
    public Object get(int index);
    public boolean remove(int index);
}
```

- Here's how you make an interface!
- It's just like a class but it lacks two things:
  - We only provide the method header.
  - They don't have attributes.

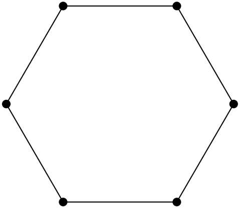
Implementing Interfaces

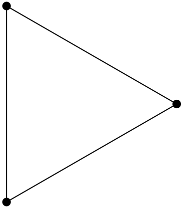
```java
class ArrayList implements List{

    public int add(Object value){
        /*rest of definition here*/
    }

    public Object get(int index){
        /*rest of definition here*/
    }

    public boolean remove(int index){
        /*rest of definition here*/
    }

    // Continue impelmenting ArrayList
        specific methods

}
```

```java
class LinkedList implements List{

    public int add(Object value){
        /*rest of definition here*/
    }

    public Object get(int index){
        /*rest of definition here*/
    }

    public boolean remove(int index){
        /*rest of definition here*/
    }

    // Continue impelmenting LinkedList
        specific methods

}
```

- Here, both classes **must** implement all methods in the list interface.
- The behaviour should be the same but the implementation can be different!

Worksheet: Implementing the Shape Interface





Off to the worksheet to implement our `Shape` interface.

Complete the following activities:

- Equilateral Triangle Class
- Hexagon Class

Why use this?

Example 1: Creating Collections of "Like" objects

```
List<Shape> shapeList = new ArrayList<>();
shapeList.add(new EquilateralTriangle(1.4));
shapeList.add(new Hexagon(0.25));
shapeList.add(new Hexagon(7.0));
shapeList.add(new EquilateralTriangle(7.25));
shapeList.add(new Hexagon(100.5));
shapeList.add(new EquilateralTriangle(75.456));
```

- All objects are of Shape type so we can create a collection of shapes.
- Like with the List interface, using the Shape interface limits us to the methods these classes have in common.

Worksheet: Implementing the Shape Interface

Off to the worksheet to implement our methods! Complete the following activities:

- sumAllShapeAreas
- sumAllShapePerims

Key Points

- By implementing an interface all method headers **must** be implemented.
- The former connects classes by some contractually obligated shared behaviour.
- This has the following benefits:
  - Reduces code complexity.
  - Allows us to rely on abstract methods rather than their implementation.

What are Generics?

```
class ArrayList<E> implements List<E>{
    public boolean add(E e){
        /* Code here */
    }
    public E get(int index){
        /* Code here */
    }
    public void remove(int index){
        /* Code here */
    }
}
```

- You've seen them before! You just didn't know it.
- `List<E> elems = new ArrayList<>()` allows us to substitute in the placeholder `E` for our type.
  - `List<String> elems = new ArrayList<>();`
  - `List<Integer> elems = new ArrayList<>();`
  - `List<Double> elems = new ArrayList<>();`

Generic Notation

- T - Type
- E - Element
- K - Key
- V - Value
- N - Number

```
//the list keeps elements
interface List<E>{
    //...
}

// the comparable interface
interface Comparable<T>{
    public int compareTo(T o);
}

//we'll cover this later in the course
interface Map<K, V>{
    //...
}
```

A Simplified List Interface

```
interface List<E>{

    public boolean add(E e);
    public E get(int index);
    public void remove(int index);

}
```

- The `E` is a placeholder for the type on which the list operates.
- We use `E` because lists have many elements stored in them.
- Each of the method parameters and returns have `E` rather than explicit types like `String`.
- As such `E` is a placeholder for when we instantiate an implementation of this interface:
    - `List<String> strList = new ArrayList<>();`

Comparable

```
public interface Comparable<T>{
    public int compareTo(T other);
}


public Animal implements Comparable<Animal>{
    // ...
    public int compareTo(Animal a){
        return name.compareTo(a.name);
    }
}
```

- Here the generic is `T`. Why?
- Compare to does a comparison between two types.
- The name attribute is a `String` so we just call the string compareTo between this and the passed in Animal instances.

Generic Class

```java
class StringC{

    String data;

    StringC(String data){
        this.data = data;
    }

    //Some methods to work
        with the data
}
```

```java
class IntegerC{

    Integer data;

    IntegerC(Integer data){
        this.data = data;
    }

    //Some methods to work
        with the data
}
```

```java
class DoubleC{

    Double data;

    DoubleC(Double data){
        this.data = data;
    }

    //Some methods to work
        with the data
}
```

- Notice how the only thing that differs is the data?

Generic Class

```java
class StringC{

    String data;

    StringC(String data){
        this.data = data;
    }

    //...
}
```

```java
class IntegerC{

    Integer data;

    IntegerC(Integer data){
        this.data = data;
    }

    //...
}
```

```java
class DoubleC{

    Double data;

    DoubleC(Double data){
        this.data = data;
    }

    //...
}
```

- Our Generic Class (GC) must be declared with:
  - Class declaration is Name<E, T, ...>.
  - Generic methods and attributes must use E, T, ... to in place of types.

```java
class GC<E>{

    E data; //generic data

    // generic constructor
    GC(E data){
        this.data = data;
    }

    //...
}

GC<String> str = new GC<>("hello");
GC<Integer> integer = new GC<>(3);
GC<Double> doub = new GC<>(3.21);
```

Generic Class

```
class GC<E>{

    E data; //generic data

    // generic constructor
    GC(E data){
        this.data = data;
    }

    // ...
}

GC<String> str = new GC<>("hello");
GC<Integer> integer = new GC<>(3);
GC<Double> doub = new GC<>(3.21);
```

- Generic classes are *very* similar to regular classes.
- You can then store generic data and have a generic stuff in them.
- Increases code reusability, decreases complexity!

Object vs E

```
class GC<E>{

    private Object[] dataArray;
        //generic data

    // generic constructor
    GC(){
        this.data = new Object[10];
    }

    //...
}
```

- Java doesn't allow for the `new` keyword to be used with `E`.
- Object is the "superclass" for all objects in Java (e.g., all Strings are Objects but not all Objects are Strings)

Storing our own objects with Generics

```java
class Dog{
    public String name;
    public String breed;
    public String sound;
    Dog(String name, String breed){
        this.name = name;
        this.breed = breed;
        sound = "dog";
    }
}

class Cat{
    public String name;
    public String breed;
    public String sound;
    Cat(String name, String breed){
        this.name = name;
        this.breed = breed;
        sound = "meow";
    }
}
```

```java
List<Dog> dogList = new ArrayList<>();
List<Cat> catList = new ArrayList<>();

Dog dog = new Dog("Jack", "Berner");
Cat cat = new Cat("Midnight", "Black");

dogList.add(dog);
catList.add(cat);
```

- Lists take a generic E as the type of data they store.
- So we can create our own classes and add instances of those classes to lists.

Storing our own objects with Generics

```
class GC<E>{

    E data; //generic data

    // generic constructor
    GC(E data){
        this.data = data;
    }

    //...
}

Dog dog = new Dog("Jack", "Berner")
Cat cat = new Cat("Midnight", "Black")

GC<Dog> str = new GC<>(dog);
GC<Cat> str = new GC<>(cat);
```

```
class Dog{
    public String name;
    public String breed;
    public String sound;
    Dog(String name, String breed){
        this.name = name;
        this.breed = breed;
        sound = "dog";
    }
}

class Cat{
    public String name;
    public String breed;
    public String sound;
    Cat(String name, String breed){
        this.name = name;
        this.breed = breed;
        sound = "meow";
    }
}
```

- We can also create our own classes and "pass" those into generic data structures.

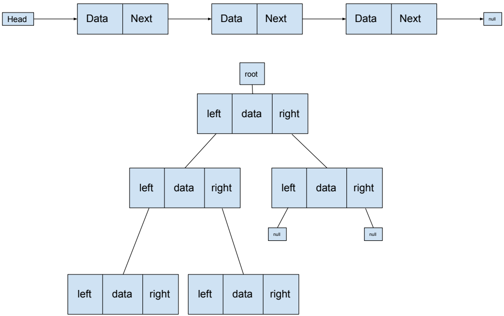
Implementing Generic Classes

```
class ArrayList<E> implements List<E>{

    public boolean add(E e){
        /* Code here */
    }

    public E get(int index){
        /* Code here */
    }

    public void remove(int index){
        /* Code here */
    }

}
```

- Again, very similar to how we implement classes without generics.
- Allows us to merge the affordance of interfaces with generics:
    - We can couple together multiple classes that implement the interface.
    - Ability to store and work with arbitrary data.

Why do we care about this? Looking forward

- Most of this class is about structuring data.
- Generics and interfaces allow us to create data structures that store *arbitrary data*.
- Generics are why we don't need a different implementation of List for every data type.



Structure of ArrayList

```
class ArrayList<E> implements List<E>{
  private static final INITIAL_SIZE = 10;
  private Object[] stuffList;
  int size;

  ArrayList(){
    stuffList = new Object[INITIAL_SIZE];
    size = 0;
  }

  public void add(E e){/*our implementation here*/}

  public boolean remove(E e){/*our implementation here*/}
}
```

Adding

null	null	null	null

Size = 0

```
SimpleList<Integer> nums = new SimpleArrayList<>();
```

.

Adding

1	null	null	null

Size = 1

```java
SimpleList<Integer> nums = new SimpleArrayList<>();
nums.add(1);
```

Adding

1	2	null	null

Size = 2

```java
SimpleList<Integer> nums = new SimpleArrayList<>();
nums.add(1);
nums.add(2);
```

.

Adding

1	2	3	null

Size = 3

```java
SimpleList<Integer> nums = new SimpleArrayList<>();
nums.add(1);
nums.add(2);
nums.add(3);
```

.

Adding

1	2	3	4

Size = 4

```java
SimpleList<Integer> nums = new SimpleArrayList<>();
nums.add(1);
nums.add(2);
nums.add(3);
nums.add(4);
//What happens when we add another element?
```

Adding

1	2	3	4	5	null	null	null

Size = 5

```
SimpleList<Integer> nums = new SimpleArrayList<>();
nums.add(1);
nums.add(2);
nums.add(3);
nums.add(4);
//What happens when we add another element?
nums.add(5);
```

Adding Method: Psudeocode

```
Add(NewElement)
    EnsureCapacity(size + 1)
    List[size] = NewElement
    Size++
EndAdd
```

Ensure Capacity

```
EnsureCapacity(MinSize)
    If(A.length < MinSize)
        A= CopyList(A, Size + AmountToIncreaseBy)
    EndIf
EndAdd
```

Removing

Condition #1: The Item we're removing is at the end (Good)

1	2	3	4	null	null	null	null

size = 4

```
primes[4] = null;
size = size - 1
```

Condition #2: The item is in the middle or front of the list (Bad)

1	null	3	4	null	null	null	null

size = 3

```
primes[1] = null;
size = size - 1
```

Why removing that way is bad

2	3	5	7	null	13	17	null	null	29	31	37	null	43	47	null

1. We can't use Size to find the true end of our list.
2. We have all these empty spaces.
3. Where do we insert?

How to remove 2?

Before:

1	2	3	4	null	null	null	null

size = 4

After:

1	3	4	4	null	null	null	null

size = 4

Step 1: Copy each element to the right of the element we want to remove 1 step to the left

How to remove 2?

Before:

1	2	4	4	null	null	null	null

size = 4

After:

1	3	4	null	null	null	null	null

size = 4

Step 2: Set end of the list (element at size - 1) to `null`.

How to remove 2?

Before:

1	3	4	null	null	null	null	null

size = 4

After:

1	3	4	null	null	null	null	null

size = 3

Step 3: Decrement size.

Remove Method: Basic Psudeocode

```
Remove(ElementForRemoval)
    i = Find(ElementForRemoval)

    If(ElementForRemoval not in List)
        Return False
    EndIf

    If(i == Size - 1)
        // Remove from the end
    Else
        // Remove from the middle
    EndIf

    Size = Size - 1
    Return True
EndAdd
```

This Week

1. This week you will be building a generic list interface and a generic list class that implements that interface.
2. Some videos are up, more to come.