Autoboxing and Unboxing

You cant create ArrayList of integers

```
ArrayList<int> x=new ArrayList<int>();
```

Type argument cannot be of primitive type

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

Solution:

```
public static void main(String[] args) {
    ArrayList<MyInt> x =new ArrayList<MyInt>();
    x.add(new MyInt( x: 50));

}
```

-Arraylist creates a list of objects of classes, so we need to pass a class.

-Easier way to do above things is Autoboxing, java supports primitive types by using a class or an object wrapper. Kinda like above for MyInt, we create a wrapper.

Wrapper is very similar to encapsulation.

Java has these in built

for int primitive type we have Integer class

```
for( int i=0;i<=10;i++){
    x.add(Integer.valueOf(i));/*
    Autoboxing
    Method valueof is taking the value of i as the
    primitive type and converting into integer class
    */
}
for( int i =0; i<=10;i++){
    System.out.println(i + "---> "+ x.get(i).intValue());
            /*
            Unboxing
        Converting back to primitve type
             */
}
```

```java
Integer myintvalue = 56;
/* SO what happens when this code gets compile this code will
*  be executed
*/
Integer myintvalue1= Integer.valueOf(56);
```

```java
Integer myintvalue = 56.56;
```

Required type: Integer

Provided: double

Similarly

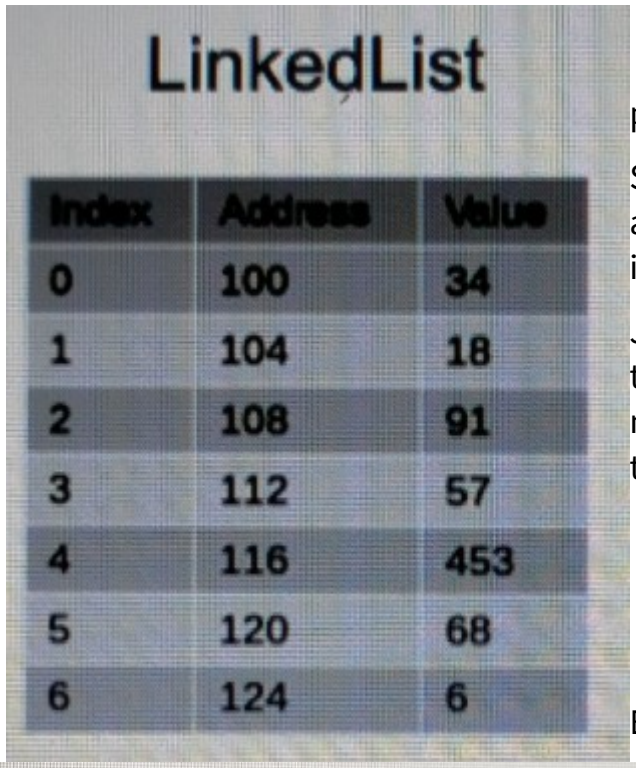
```java
int y=myintvalue;
int z= myintvalue1.intValue();
```

```java
ArrayList<Double> myDouble = new ArrayList<Double>();
for( double  i=0.0; i<=10.0;i++){
    myDouble.add(Double.valueOf(i));
}
for( int i=0;i<10;i++){
    System.out.print("\n"+myDouble.get(i).doubleValue());
}
```

```java
ArrayList<Double> myDouble = new ArrayList<Double>();
for( double  i=0.0; i<=10.0;i++){
    myDouble.add(i);
}
for( int i=0;i<10;i++){
    System.out.print("\n"+myDouble.get(i));
}
```

So basically u dont need to learn Autoboxing and Unboxing .

Linked List

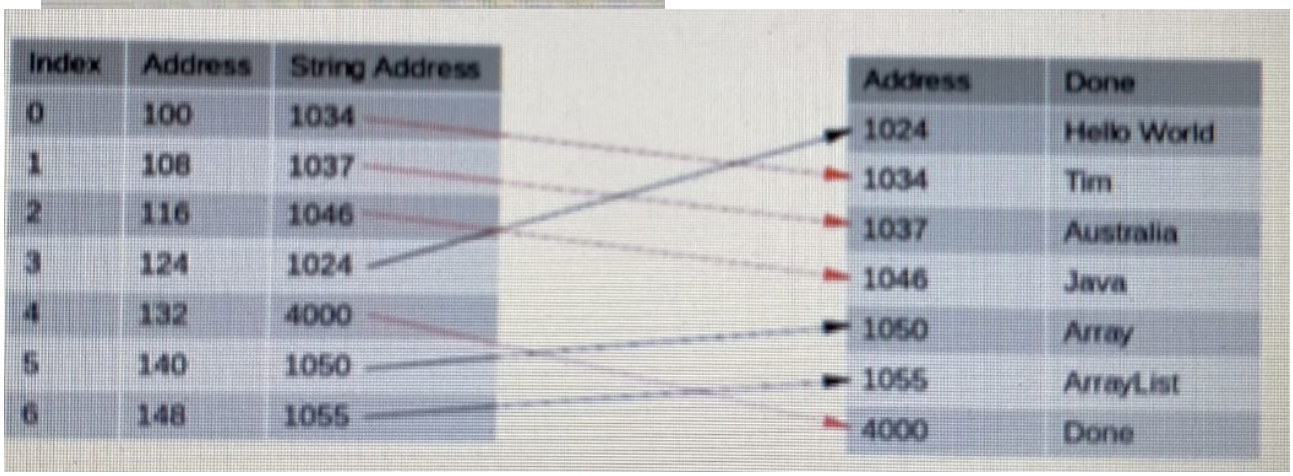


Index: is what we use to access a particular value.

So what happens internally. Java would allocate 4 bytes of memory for each integer.

Java would try to do this contigiously. In this case the formula, which physical memory address java could use to grab the next value.

But strings can be of variable size



So for strings, the memory allocated itself is 8 bytes, but that points to another location in memroy where the string is.

Here string addresses do not need to be contigious.

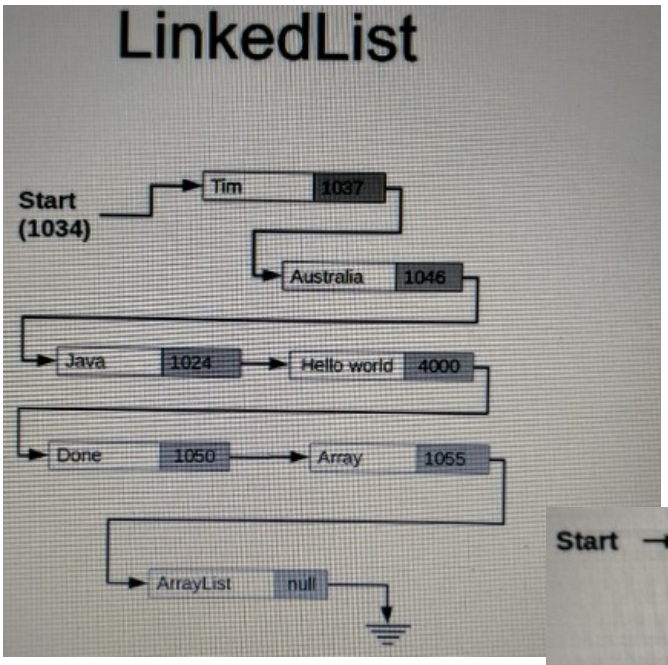
Important Note:-

When

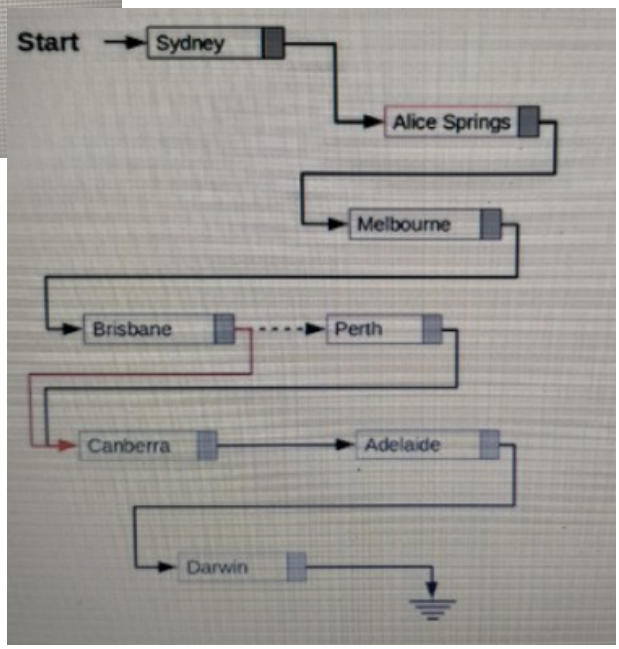
```java
ArrayList<Integer> x= new ArrayList<Integer>();
x.add(1);
x.add(12);
x.add(13);
x.add(14);

/* When deleting or adding an item into ArrayList */
x.add( index: 1, element: 2);
```

In ArrayList all the other entries need to be moved down, considering an Integer List.

It becomes slower to process as input increases.



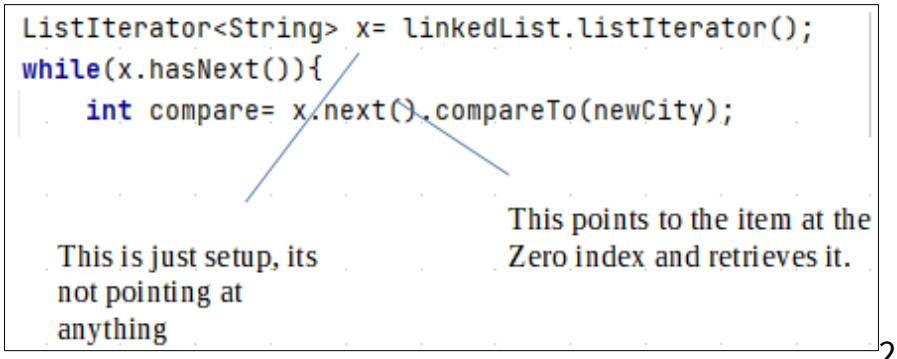
Since perth is removed, and we dont have any pointers left, it will be removed by java garbage collection.




```java
private static void printList(LinkedList<String> linkedList ){
    Iterator<String> i=linkedList.iterator();/* Its like for-loop
    while(i.hasNext()){
        System.out.println("Now Visiting "+i.next());
    }
    System.out.print("xxxxxxxxxx");

}
```

Tips to learn stringListIterator

1.



2. Whenever u use next

Imagine its a state or (Steps in a ladder)

x.next() you are pointing to the object u just retrieve

Again()

x.next() you are pointing to the object u just retrieve

So to insert before to what u pointing at, u need to

x.previous()

x.next()

3) Java implement linkedlist as a double linkedlist.

```java
private static boolean addInOrder(LinkedList<String> linkedList, String newCity) {
    ListIterator<String> x = linkedList.listIterator();
    while (x.hasNext()) {
        int compare = x.next().compareTo(newCity);

        /* result is negative if string precedes argument
         *
         * -1 when String Precedes
         * 1 when Argument Precedes */
        if (compare == 0) {
            System.out.print(newCity + " already exist");
            return false;
        } else if (compare > 0) {
            x.previous();
            x.add(newCity);
            return true;
        } else if (compare < 0) {

            /*System.out.print("Move on to the next city");*/
        }



    }
    x.add(newCity);
    return true;

}
```

```
Welcome toMelbourne
Enter ur number: 2

Welcome toPerth
Enter ur number: 3

Welcome toPerth
Enter ur number:
```

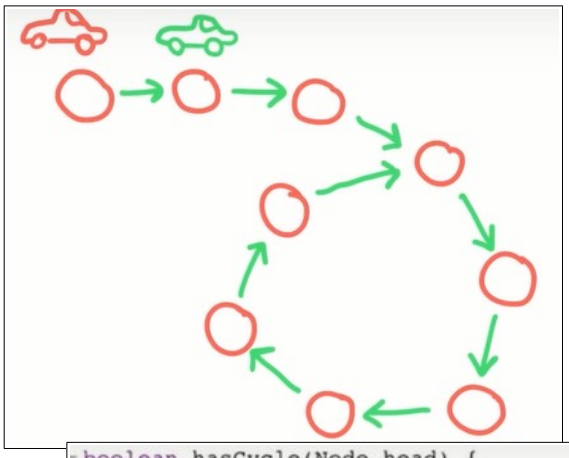
Be careful with this error.

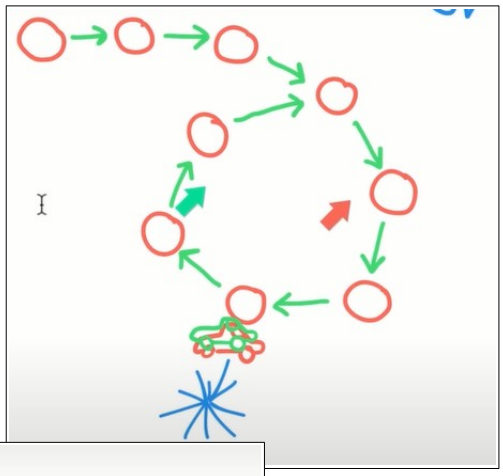
Basically when u go forward in linkedlist, to go backward u need to run backwards twice to go one step back and vice versa.

LinkedList problems:

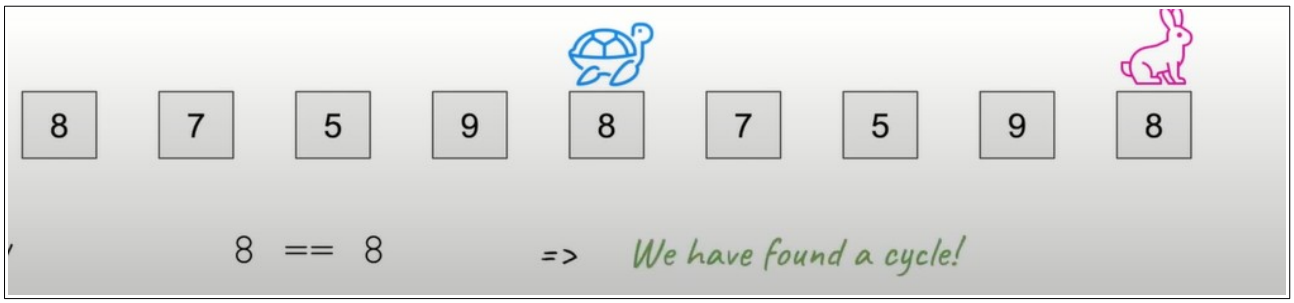
Cycles in a linkedlist: solved using Floyd tortoise and hare algorithm

links: [Hackerank](),





```
boolean hasCycle(Node head) {
    if (head == null) return false;
    Node fast = head.next;
    Node slow = head;
    while (fast != null && fast.next != null && slow != null) {
        if (fast == slow) {
            return true;
        }
        fast = fast.next.next;
        slow = slow.next;
    }
    return false;
}
```



Linkedlist challenge skipped

```
private Song findSong(String title){
    for(Song e : this.songs){
        if(e.getTitle().equals(title)){
            return e;
        }
    }
    return null;
}
```

This is known as for-each cmd

Inner Abstract classes and interfaces

Interfaces specifies method that a particular class that implements the interface must implement. Interface is actual abstract, that means there is no actual code for any of the methods, you only supply the actual method names and parameters.

The actual code still goes in the actual class that you are creating.

But the idea is to use an interface to provide a common behavior that can be used by several classes by having them all implement the same interface. Basically to standardize the way a particular class is used.

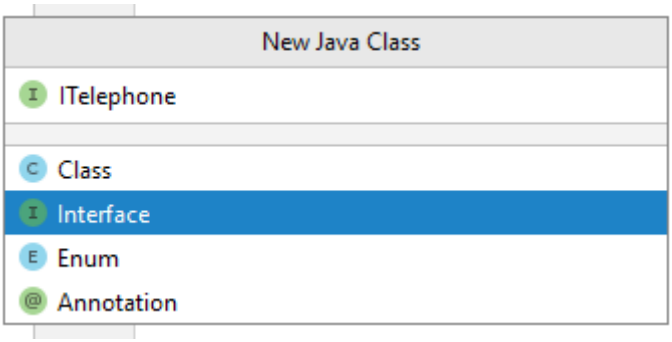
To make sure

1.

The signature doesnt change(cuz if u got another code that rely on the fact that this parameter is a string, so we dont broke the other code).

```
public void openClutch(String in_out){
    this.clutchIsIn=in_out.equalsIgnoreCase( anotherString: "in");
}
```

2.



The “I” while naming an Interface in start is conventional.

3.

```
public interface ITelephone {
    void powerOn();
    void dial(int phoneNumber);
    void answer();
    boolean callphone(int phoneNumber);
    public boolean isRinging();
}
```

keyword public is redundant cuz we implementing the method in the class

4.

```
public class DeskPhone implements ITelephone{
}
```

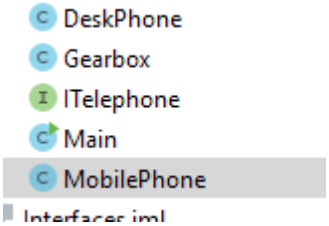
Class 'DeskPhone' must either be declared abstract or implement abstract method 'powerOn()' in 'ITelephone'

Implement methods Alt+Shift+Enter More actions... Alt+Enter

5.

```
public class Main {
    public static void main(String[] args) {
        ITelephone kazimPhone=new DeskPhone();
        DeskPhone sahilPhone=new DeskPhone();
        kazimPhone.powerOn();
        kazimPhone.callphone( phoneNumber: 1111111);
        kazimPhone.answer();


    }
}
```

6.



They are colored differently.

7.

You can use generic List interface instead of ArrayList or Linkedlist.

And do polymorphism.

And make the code suitable for both ArrayList and Linkedlist and Vector.

So basically when u initialize the List interface variable u specify the type, whether is it a Vector, ArrayList , LinkedList.

So its a good idea to make methods to use generic List and in the program decide the actual type that you want in your program.

Deciding Whether implement an interface or inherit from a base class. Sometimes it can be difficult to decide which one to do.

--→ The way to decide that generally is to really consider the relationship of the final class to the object that its extending or implementing.

So for our ITelephone example both of the device have a telephone, but deskphone doesnt have anything else and MobilePhone(Play games and more).

So a mobilePhone is actually a computer that has a phone interface.

So mobilePhone can telephone and DeskPhone can telephone, which indicates that they should both implement.

In java, a class can only inherit from one super class, unlike other languages for e.g

c++ , But you can actually implement from many interfaces.

Some languages allow moddable inheritance by allowing a class to inherit from several base classes.

In java, multiple inheritance is only available by implementing several interfaces.

For e.g a dog can extend animal and implemet walk.

A bird can extend animal and implement walk and fly.

```
ISaveable kazim= new Player( name: "kazim", hitpoints: 100, strength: 100, weapon: "Valyrian Steel");
System.out.print(kazim.toString());
saveObject(kazim);

kazim.setHitpoints(8);
System.out.print(kazim);
kazim.setWeapon("Dragon Glass");
saveObject(kazim);
loadObject(kazim);
System.out.print(kazim);
```

```
( (Player) kazim).setHitpoints(8);
System.out.print(kazim);
((Player)kazim).setWeapon("Dragon Glass");
saveObject(kazim);
loadObject(kazim);
System.out.print(kazim);
```

Here we are casting.

Interface challenge part 2

It can be confusing when to declare an instance as a class or using an interface:

Based on do u wanna use polymorphism.

Tip:8:32 You can always cast the instance to the interface or class type as necessary.

Also notice the parameters to methods that have been declared as the interface type are automatically cast from the class type for us.

Inner classes

To nest a class inside another class. **Link**:Inner classes Oracle doc

Four types of nested classes

- Static nested classess
- non static nested classes /inner class
- local class: thats an inner class thats defined inside of a scope block which is usually a method
- anonymous class which is a nested class without a class name

Static nested class is mainly used to associate a class with its outer class.

In terms of behavior, you'd think of it as being identical to a top level class, but the difference is that its packaged in its outer class rather than in the package.

So that really means that it cannot access the nonstatic methods or members of its outer class without first creating an instance of that class.

Non static nested class: it doesnt make sense to refer to a class without its outer class.

So if we are modeling an engine, we might actually have a gearbox class and a gear class.

```java
public class Gearbox {
    //base class
    private ArrayList<Gear> gears;
    private int maxGears;
    private int gearNumber=0;

    public Gearbox(int maxGears){
        this.maxGears = maxGears;
        this.gears=new ArrayList<Gear>();
        Gear neutral=new Gear( gearNumber: 0, ratio: 0.0);
        this.gears.add(neutral);
    }
    //inner class
    public class Gear{
        //when it doesnt makes sense to refer to something
        //except within the context
        private int gearNumber;
        private double ratio;

        public Gear(int gearNumber,double ratio){
            this.gearNumber=gearNumber;
            this.ratio=ratio;
        }
        public double driveSpeed(int revs){
            return revs*this.ratio;
        }
    }
}
```

- Now instance of the gear class have got access to all the methods or fields of the outer gearbox class, even those mark as private.

-Notice the **this** keyword in the inner class refers to the inner class and not the outer one.

-Above both Gear class and Gearbox have a attribute ‘gearNumber’ with same name;

Inside Gear, to access Gearbox gearNumber ,

gearbox.this.gearNumber

so if a word or variable or parameter , if name is same as the one in outer class, its said to shadow the declaration of the outer class or in the outer class.

So pls do rename one of the variable

```
public class Main {
    public static void main(String[] args) {
        Gearbox mclaren=new Gearbox( maxGears: 6);
        Gearbox.Gear first= mclaren.new Gear( gearNumber: 1, ratio: 12.3);
        System.out.print(first.driveSpeed( revs: 1000));
    }
}
```

- Often though the inner class is gonna be private anyway so IOW instances will only be created by the outer class.

-So if u got Gearbox as the outer class, the inner class u dont want people to directly access. So IOW, the instance is gonna be created by the outer class.

And making this class private here, because we are not actually interested in the individual gear objects we are driving, we wanna change the gear, we do that by interacting with the gearbox not with the individual gears.

We dont have any reference to the Gear class. Gear class is local to that Gearbox.

Encapsulation

:time 5:44

There are two special cases of inner classes and thats local classes and anonymous classes. So local classes are declared inside of a block such as a method or if statements and their scope is restricted completely to that particular block.

Inner Class part 2:

```java
public class Button {
    private String title;
    private OnClickListener onClickLister;

    public Button(String title){
        this.title=title;
    }
    public String getTitle(){
        return title;
    }
    public void setOnClickLister(OnClickListener onClickLister){
        this.onClickLister=onClickLister;
    }
    public void onClick(){
        this.onClickLister.onClick(this.title);
    }
    public interface OnClickListener{
        public void onClick(String title);
    }
}
```

```java
public class Main {
    private static Scanner scanner=new Scanner(System.in);
    private static Button btnPrint = new Button( title: "Print");
    public static void main(String[] args) {


        class ClickListener implements Button.OnClickListener{

            public ClickListener(){
                System.out.println("I've been attached");
            }

            @Override
            public void onClick(String title) {
                System.out.print(title+" was clicked");
            }
        }

        btnPrint.setOnClickLister(new ClickListener());
    }
```

Above is a local class which is applicable just for the block only.

Anonymous class is also a local class, but its got no name.

They have to be declared and instantiated at the same time, because they havent got a name. They are use and throw.

They are very common for touching event handlers to button in an user interface.

For e.g if you are programming android apps.

So if u have several buttons and each require a different onclick method. So using a local class might not be the best solution.

So rather than define a class for each btn, we could use an anonymous inner class to declare an object.


```
btnPrint.setOnClickLister(new Button.OnClickListener() {
    @Override
    public void onClick(String title) {
        System.out.print(title+" was clicked");
    }
});
btnPrint.onClick();
```

Inner class challenge skipped

Tip: One of the beauty is about creating things properly the first time, we can make these changes internally without affecting any other part of the application.

Abstract Classes

Abstraction is when you define the required functionality for something without actually implementing the details. We are focusing in what needs to be done and not on how its to be done.

Interfaces you saw are purely abstract and I dont specify any actual aspect of the implementation. The actual implementation were left to the classes that implement the interface.

You could have each class which implement the interface might do completely do different thing with implement methods.

```
public interface ISaveable {
    1 related problem
    ArrayList<String> write();
    void read(List<String> saved_values);

}
```

-You cant instantiate a class that is abstract

if you look at this interface, its not complete abstraction. In this particular case, there is a specification that ArrayLists must be used. But in a typed language like java its hard to avoid that sometimes.

We can abstract this again.

```
public interface ISaveable {
    List<String> write();
    void read(List<String> saved_values);

}
```

Because we didnt force the Class that implement the interface to have to use an ArrayList.

Recommended, when ever u are using a list of some type in your programs, in terms of declaration.

```
public interface ISaveable {
    1 related problem
    List<> write();
    1 related problem
    void read(List<> saved_values);

}
```

We can abstract it again, and let the user decide what primitive datatype or datatype or class object to store.

Extra flexibility

Interfaces in java are by definition are abstract, cuz u cant instantiate an Interface.

You instantiate a class that implements the interface.

Java also allows abstract classes, which are classes that define methods but do not provide an implementation of those methods. The implementation itself is left to the classes that inherit from the abstract class.

```
public abstract class Animal {
}
```

```
public abstract class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }
    public abstract void eat();
    public abstract void breathe();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

So we gonna inherit from this abstract class and its gonna let us define behaviors that are necessary without specifying how they are to be performed.

There is no requirement for the other class, that is subclassing to actually implement those methods but by creating those abstract methods. We are actually forcing the class that is ultimately gonna implement from this abstract class to create those methods for us. Thats why we marked them as abstract.

Not all methods need to be abstract. Unlike an Interface.

```
public abstract class Dog extends Animal{
    public Dog(String name){
        super(name);
    }
    public void eat(){

    }
    public abstract void breathe();
}
```

e.g

```
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void eat() {
        System.out.print("\n Worms yummm");
    }

    @Override
    public void breathe() {
        System.out.print("Breathe in, Breathe out");
    }
}
```

Not all birds can fly. So implementing a fly method in Bird class is not a good idea.

Better idea would be to create an abstract Bird class that extends Animal and also has an abstract fly method that individual Bird objects can implement as they are able to.

```
public abstract class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void eat() {
        System.out.print("\n Worms yummm");
    }

    @Override
    public void breathe() {
        System.out.print("Breathe in, Breathe out");
    }
    public abstract void fly();
}
```

-Abstract class part 2.

Relationships: Is-A, has-A, can

dog is-a Animal

parrot is-a Bird

a bat is not a bird, and dragonflies is not a bird, but they can fly.

So we could rename our bird class flying animals and have bats and bird both inherit from that.

But it should be easy to see the problems that could potentially cause,

a bat gave birth to young rather than laying eggs. This may lead to having flying animals being inherited by classes called mammals and birds.

So what we did wrong is that we put a fly methods in bird class. Previously we have placed an abstract method in the bird class.

A bird can fly and a bat can fly, a better design would be to have created CanFly interface, which both bats and birds can implement.

He take back using IcanFly to CanFly.

```
public abstract class Bird extends Animal implements ICanFly {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void eat() {
        System.out.print("\n Worms yummm");
    }

    @Override
    public void breathe() {
        System.out.print("Breathe in, Breathe out");
    }
    public void fly(){
      System.out.print(getName()+ "is flapping its wings");
    }
}
```

```java
public class Penguin extends Bird {
    public Penguin(String name){
        super(name);
    }

    @Override
    public void fly() {
        super.fly();
        System.out.print("I am not very good at that,
                "can i go for a swim instead?");
    }
}
```

So we have a bat class and dragon fly class that inherit from a suitable base class and both could then implement the can fly interface.

By adding canFly interface, we are actually adding more flexibility into the design to enable us to create other classess and we wouldnt be a victim trying everything to fit into the one class as we did earlier.

Now looking at animal you might actually argue that all animal can breath and can eat. That they should be implemented as interfaces and absolutely that could actually work. As these methods are common to all animals. It makes sense to include them in the base abstract class and let later classess to implement them.

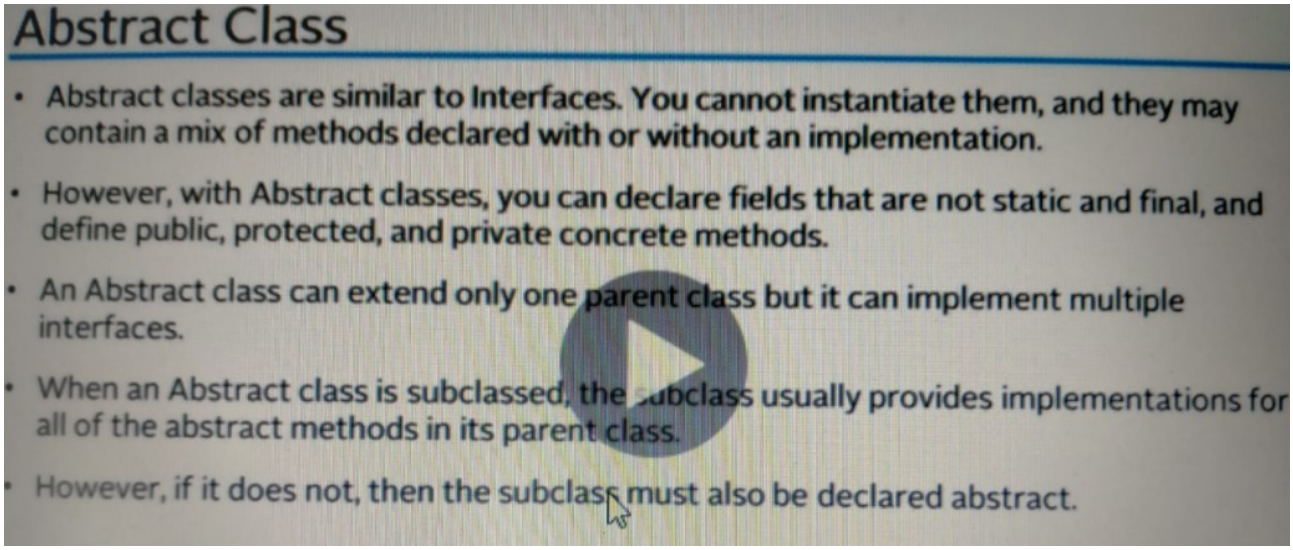
It basically ends up in deciding whether you create an interface for it or whether it should be an abstract method as the case here is for both the eat and breathe methods in the animal class.

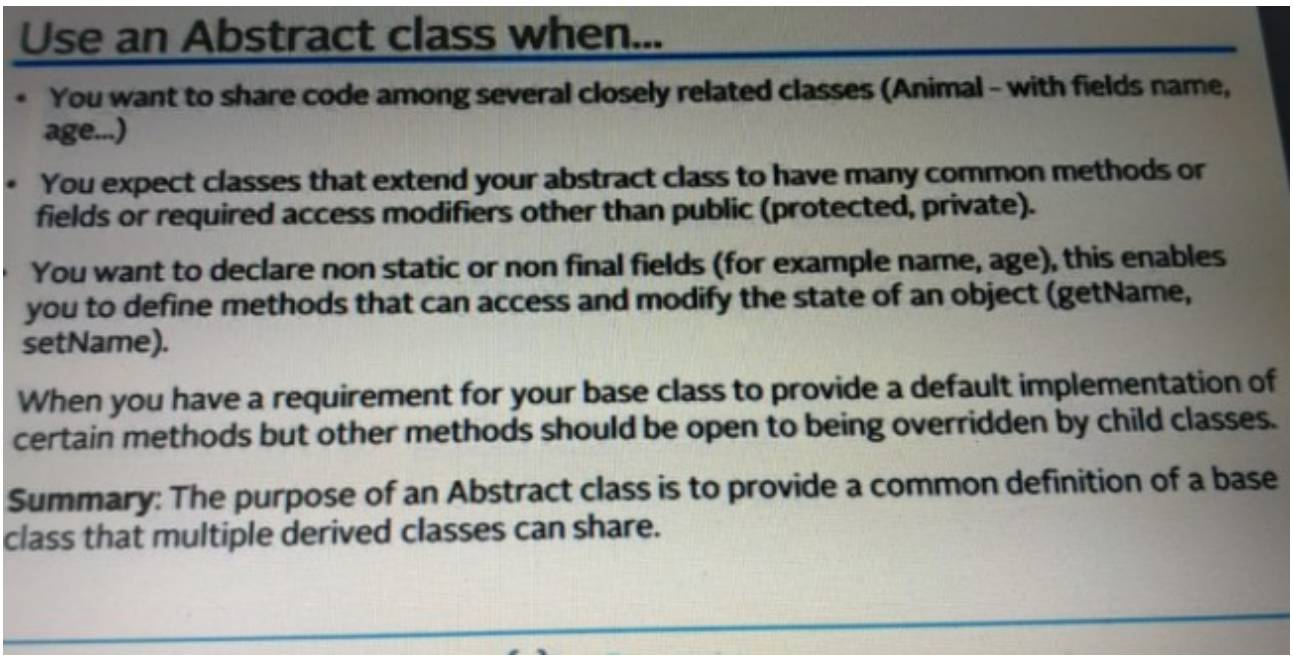
An abstract class can have member variables that are inherited, something that can't be done in interface, interfaces can have variables, but they're all public static final variables, which essentially are gonna be constant values that should never change with a static scope.

They have to be static, because non static variable require an instance, and u cant instantiate an interface.

	Interface	Abstract
Constructor	No	Yes
Methods	Automatically public	Any visibility
Defined method	No	Yes

Abstract class vs Interface.





Interface

- An interface is just the declaration of methods of an Class, it's not the implementation.
- In an Interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement the Interface.
- Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.
- You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. All methods in interfaces are automatically public and abstract.
- An Interface can extend another interface.

Use an Interface when...

- You expect that unrelated classes will implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but you are not concerned about who implements its behavior.
- You want to separate different behavior.
- The Collections API is an excellent example, we have the List interface and implementations ArrayList and LinkedList (more on that later in the course).
- The JDBC API is another excellent example. It exist of almost only interfaces. The concrete implementations are provided as "JDBC drivers". This enables you to write all the JDBC code independent of the database (DB) vendor. You will learn more about JDBC later in the course.

Abstract class challenge

```java
public abstract class ListItem {
    protected ListItem righLink=null;
    protected ListItem leftLink=null;

    protected Object value;
    //3:00
    public ListItem(Object value){
        this.value=value;
    }
    abstract ListItem next();
    abstract ListItem setNext(ListItem item);
    abstract ListItem previous();
    abstract ListItem setPrevious(ListItem item);
    abstract int compareTo(ListItem item);

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

The reason fields are declared protected rather than private, because we need to be able to access them from our concrete subclass.

We could have left the access modifier off, which could have meant that the member variables are packaged private. So IOW, subclasses in the same package would be able to access them. But not subclasses in other packages.

New for-loop technique over an array.

```java
String stringData= "Darwin Brisbane Perth Melbourne Canberra A

String[] data= stringData.split( regex: " ");
for(String s: data){
    list.addItem(new Node(s));
}
list.traverse(list.getRoot());
```

Tip Regarding Recursion is to be careful with it, dont use it everywhere else it will overwhelm the call stack.

For e.g for binary tree, recursion would be fine,

but for linkedlist bad.

Skip: Abstract class challenge