Agenda:

1) Wrapper class
2) Object class
3) Generics
4) Collection framework
5) Iterable v/s Iterator.

WRAPPER CLASSES

Java → OOP language

primitive data types → int, boolean, char etc.

Wrapper classes are classes which encapsulate primitive data types and provides lots of utility functions

Primitive	Wrapper
int	Integer
long	Long
char	Character
boolean	Boolean
double	Double

Example

parseInt( )
obj ⟶ toString( )
longValue( )
etc.

int i = 3; ⟶ stack

Integer a = new Integer(3); ⟶ Heap

Autoboxing and Unboxing

Integer wrappedNum = 5;

(Autoboxing) [new Integer(5);]

int num = wrappedNum;

(Unboxing)

Objects default value = null

GENERICS

```
class IntegerPrinter {
    Integer intToPrint;
    public IntegerPrinter(Integer num) {
        this.intToPrint = num;
    }

    public void print() {
        sout(intToPrint);
    }
}
```

What if you want to do same for Double, String?

Duplication ↑

Class Explosion → one class for each type.

In Java, Object class is the root of all classes. Implicitly all classes inherit from Object class.

Approach-1

↓

```
class ObjectPrinter {
    → Object objToPrint;
    public ObjectPrinter(ObjectPrinter op) {
        this.objToPrint = op;
```

(Animal) objToPrint.walk()

```
[ ObjectPrinter op = new ObjectPrinter( ↑ )
```

any type obj.

↑

Object → ROOT of class hierarchy

Animal
- walk()

Cons →
1. Lack of type safety
2. No compile time checks
3. Performance overhead → Runtime checks and explicit type-casting

Approach - 2

```
int num = 5 ;
```

int → fixed

num = 5 → can change

With generics → data type can vary based on client

```
class Printer <T> {
    T intToPrint;
    public Printer(T num) {
        this.intToPrint = num;
    }
    public void print() {
        sout(intToPrint);
        T.walk()
    }
}
```

T extends Animal

<ABC> ✓

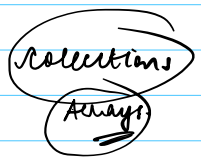
```
Printer<Double> dp = new Printer<>(23.0)
dp.print();

Printer<String> sp = new Printer<>("Rahul")
sp.print()
```

(*) No typecasting to object class.

COLLECTION FRAMEWORK IN JAVA

COLLECTION → group of objects

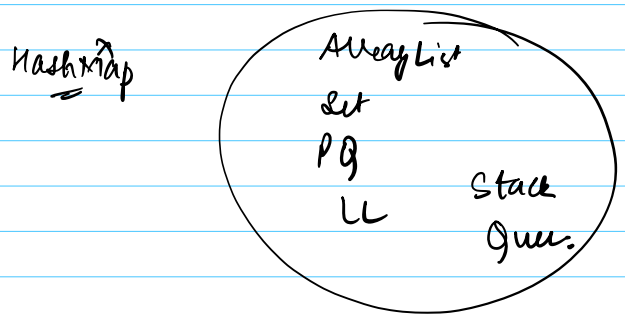
FRAMEWORK → Ready to use set of tools, templates for fast application development



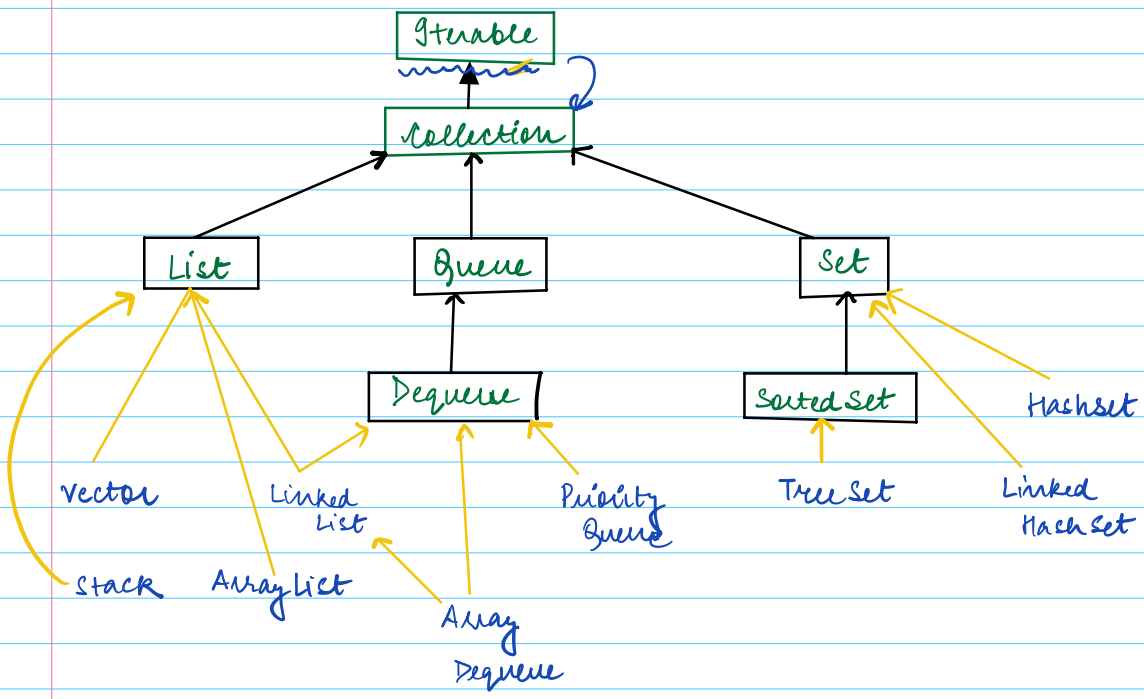
* Set of classes and interfaces to manage data
  ↳ Built-in data structures ready to use

↓

provides us various data structures and algorithms to efficiently store, retrieve and manipulate collections of objects



Class Hierarchy



1) Iterable : interface that has a method iterator

```
public interface Iterable <T> {

    Iterator <T> iterator;

    .
    .
    .

}
```

2) Collection : Interface defining basic methods for working with collections

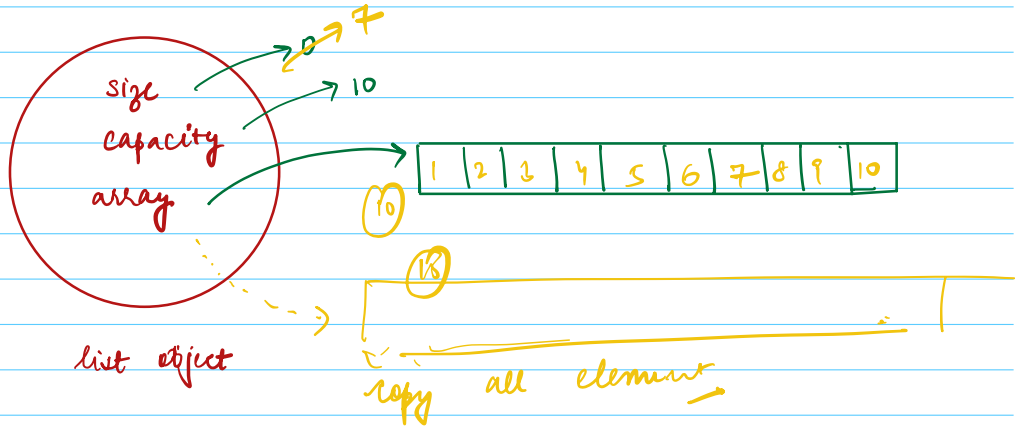
3) List : an ordered collection that allows duplicates and provides access based on indices

4) Set : a collection that does not allow duplicates and typically don't guarantee any specific order

ArrayList

→ Dynamic Array

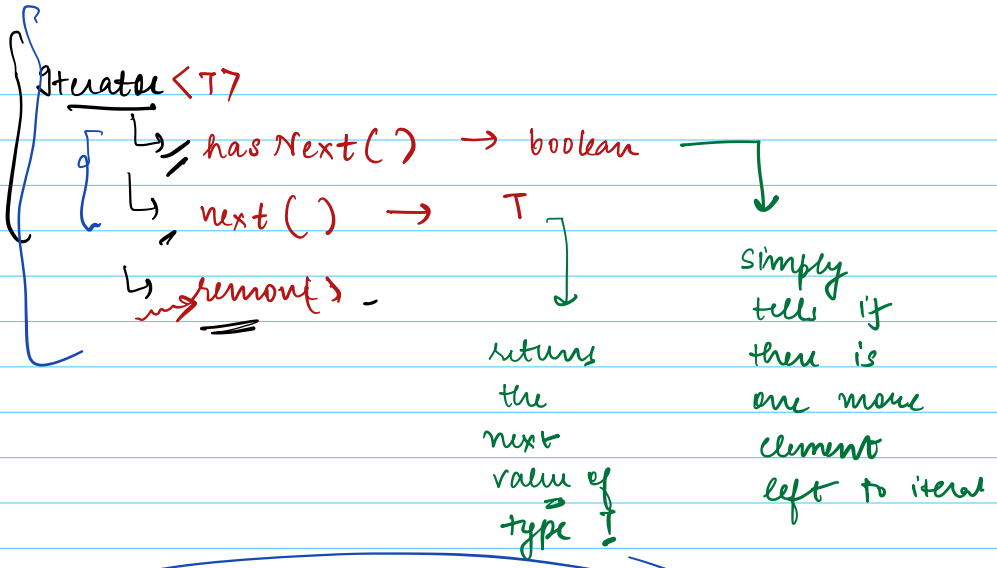
```
ArrayList<Integer> list = new ArrayList<>();
```



- .add(val) ⇝ add(1) (2) 1 ⇝ add(11)
- .add(pos, val)
- .set (pos, val)
- .get (pos) ⇝ arr[pos]

similar to AL conceptually

Vector	V/S	ArrayList
Synchronized and hence Thread Safe		NOT synchronized and hence not thread safe.
slower		faster




```
while ( Iterator.hasNext() )
    sout ( Iterator.next() )
```

Iterable : →

- iterator()
- { forEach()
  Split iterator() }

whoever implements me, will have an iterator

```
for ( int nums : numbers )
    sout(nu
```