Object Oriented Programming using Java

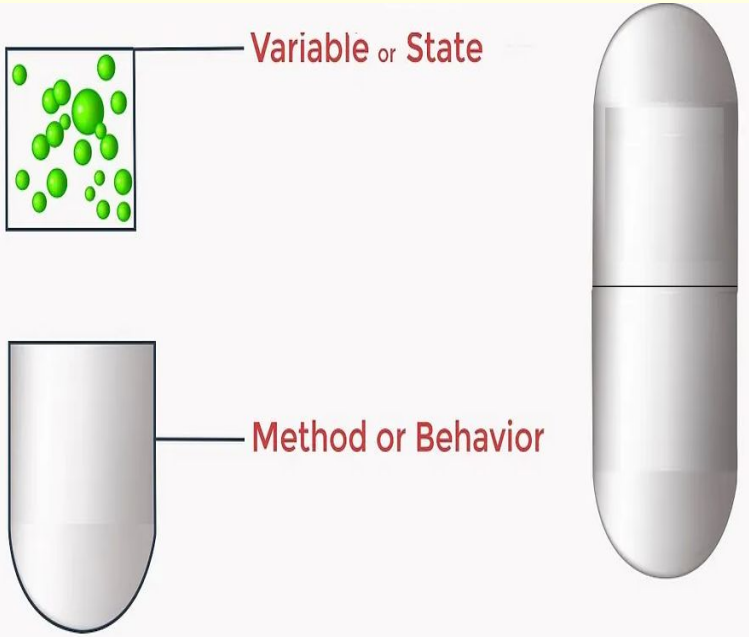
By

Dr. Sunirmal Khatua,

Visvesvaraya Young Faculty Fellow, Govt. of India

Assistant Professor, University of Calcutta

Encapsulation




```
class Student {
      private String name;
      private int marks;

     public String getName(){
           return name;
     }

     public void setName(String n){
           name = n;
     }

    public intgetMarks(){
           return marks;
     }

     public void int m){
           marks = m;
     }

}
```

Abstraction

- The process of hiding the details and showing only the essential information to users.
- Abstraction is achieved through either abstract classes or interfaces.

Generics in Java

```
public class ArrayUtil {




}
```

Print elements of an Array : Overloading

```
public class ArrayUtil {

    public void print(Integer[]  a){
        for(Integer i : a)
            System.out.println(i);
    }

    public void print(String[]  a){
        for(String s : a)
            System.out.println(s);
    }

}
```

```
public class Test {

    public static void main(String args[]){
        ArrayUtil au = new ArrayUtil();
        Integer[] a = {10,20,30,40};
        au.print(a);
        String[] s = {“a”, “b”, “c”, “d”};
        au.print(s);
    }

}
```

Print elements of an Array : A better way

```
public class ArrayUtil {

    public void print(Integer[]  a){
        for(Integer i : a)
            System.out.println(i);
    }

    public void print(String[]  a){
        for(String s : a)
            System.out.println(s);
    }

}
```

```
public class ArrayUtil<T> {

    public void print(T[]  a){
        for(T i : a)
            System.out.println(i);
    }

}
```

Parameterized / Generic Class

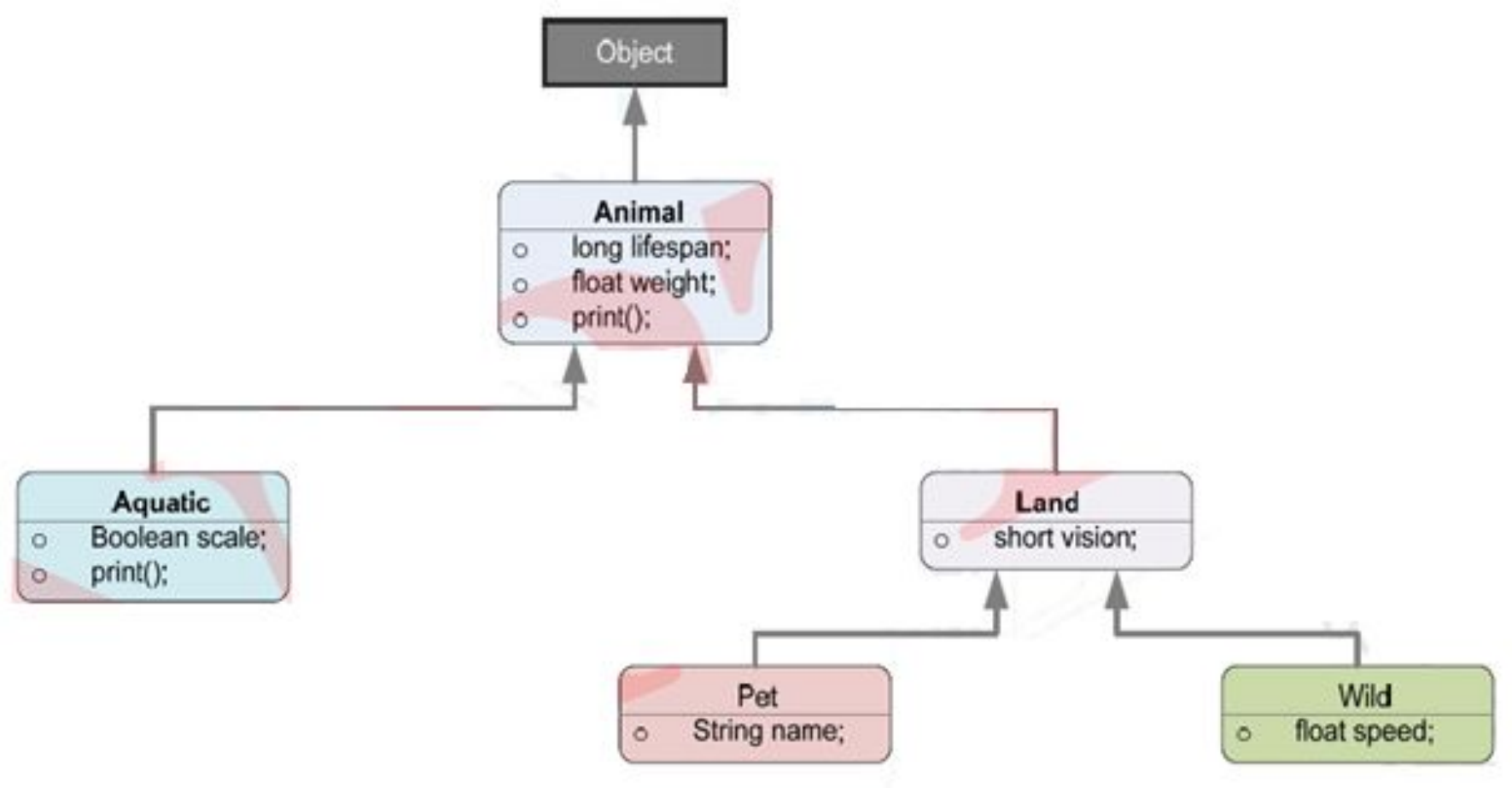
```
public class Test {

    public static void main(String args[]){
        ArrayUtil au = new ArrayUtil();
        Integer[] a = {10,20,30,40};
        au.print(a);
        String[] s = {“a”, “b”, “c”, “d”};
        au.print(s);
    }

}
```

Generic Class: Some Examples

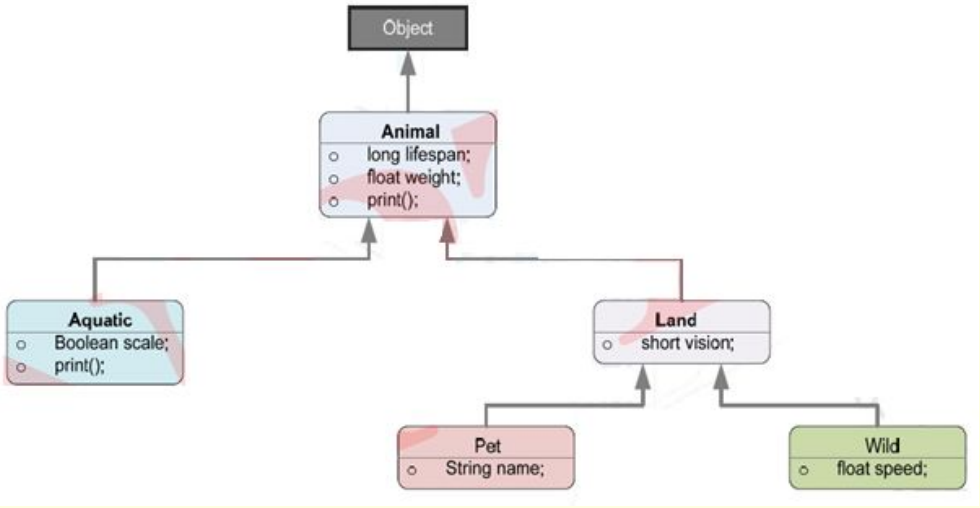
```
public class Sample<T>{
    private T data;

    public void setData(T newData){
        data = newData;
    }

    public T getData(){
        return data;
    }
}

public class  Pair<T>{

    private T firstData;

    private T secondData;

    public pair(T firstData, T secondData){
        this.firstData = firstData;
        this.secondData = secondData;
    }
}
```

```
public class  Pair<T1,T2>{

    private T1 firstData;

    private T2 secondData;

    public pair(T1 firstData, T2 secondData){
        this.firstData = firstData;
        this.secondData = secondData;
    }

    public void setFirstData(T1 firstData){
        this.firstData = firstData;
    }

    public T1 getFirstData(){
        return firstData;
    }
    public void setSecondData(T2 firstData){
        this.secondData = secondData;
    }

    public T2 getSecondData(){
        return secondData;
    }

}
```

Wildcards in Generics



Wildcards in Generics




```
class AnimalWorld<T extends Animal> {

    T [ ] listOfAnimals;

    AnimalWorld(T [ ] list)
        listOfAnimals = list;
    }
}
```

Wildcards in Generics

```
class BoundedWildcards {
    //Case 1: Unbound wildcard:
    static void showAnimals(AnimalWorld<?> animals) {
        for(Animal a : animals)
            a.print();
    }
    // Case 2: Lower bounded wildcard:
    static void showAnimal(AnimalWorld<? super Animal> animals) {
        for(Object a: animals)
        a.print();
    }

    // Case 3a: Upper bounded wildcard:
    static void showPet(AnimalWorld<? extends Pet> animals) {
        for(Object a: animals)
        a.print();
    }
}
```

Enum in Java

- Enum is a special data type or special class in Java that allows variables to be defined as Symbolic Constants

```
public enum Day{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
Day d = Day.MONDAY;
```

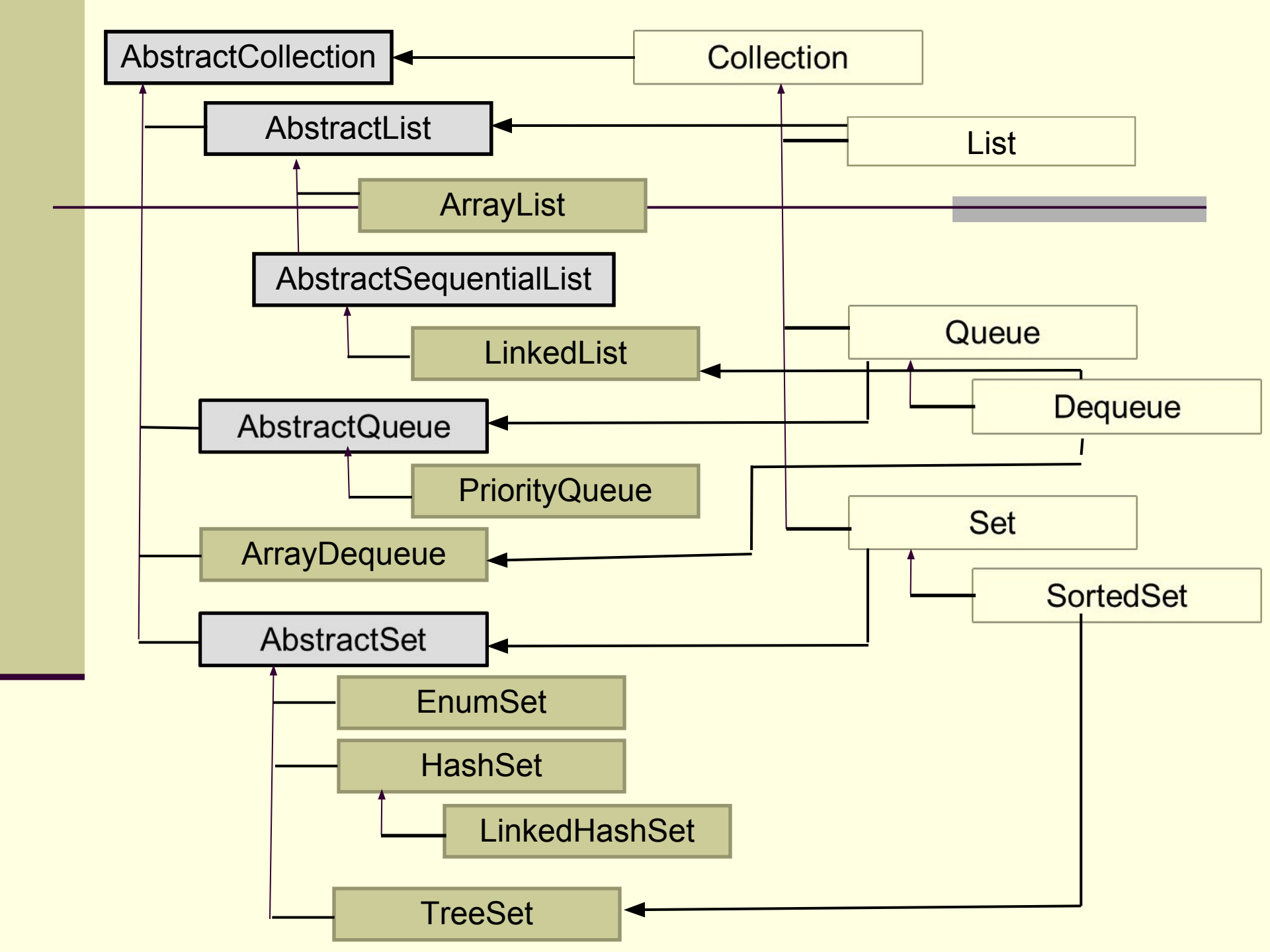
System.out.println(d) MONDAY
System.out.println(d.ordinal()) 1

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6), VENUS (4.869e+24, 6.0518e6), EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6), JUPITER (1.9e+27,  7.1492e7), SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),  NEPTUNE (1.024e+26, 2.4746e7);

    final double mass;
    final double radius;
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
}
```

```
Planet p = Planet.EARTH;

System.out.println("For "+p+
    "mass="+p.mass+" radius="+p.radius)
```

Data Structure in Java

- Java 2 has a set of well defined Data Structures in the package java.util
- The set of Classes and Interfaces in java.util package are popularly known as Java Collection Framework (JCF).
- Prior to Java 2, Java supported some ad hoc classes for data structure such as Dictionary, Hashtable, Vector, Stack and Properties
- However, these old classes are not deprecated after JCF. Rather they are fully compatible and still used along with JCF classes. These old classes are now called Java Legacy Classes.
- JCF has two important parts: Collections and Map


AbstractCollection
Collection
AbstractList
List
ArrayList
AbstractSequentialList
Queue
LinkedList
Dequeue
AbstractQueue
PriorityQueue
Set
ArrayDequeue
SortedSet
AbstractSet
EnumSet
HashSet
LinkedHashSet
TreeSet

Collection Interface

`add(o)`	Add a new element
`addAll(c)`	Add a collection
`clear()`	Remove all elements
`contains(o)`	Membership checking.
`containsAll(c)`	Inclusion checking
`isEmpty()`	Whether it is empty
`iterator()`	Return an iterator
`remove(o)`	Remove an element
`removeAll(c)`	Remove a collection
`retainAll(c)`	Keep the elements
`size()`	The number of elements

List Interface

- `add(i,o)` Insert `o` at position `i`
- *`add(o)`* Append `o` to the end
- `get(i)` Return the `i`-th element
- `remove(i)` Remove the `i`-th element
- `set(i,o)` Replace the `i`-th element with `o`
- `indexOf(o)`
- `lastIndexOf(o)`
- `listIterator()`
- `sublist(i,j)`

Ordering and Sorting a Collection

- There are two ways to define orders on objects.
  - Each class can define a *natural order* among its instances by implementing the **`Comparable`** interface.

```
int compareTo(Object o)
```

  - Arbitrary orders among different objects can be defined by *comparators*, classes that implement the **`Comparator`** interface.

```
int compare(Object o1, Object o2)
```

Example

```
public class Student implements Comparable<Student>{
    String name;
    int marks;
    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }
    @Override
    public String toString() {
        return "["+name+","+marks+"]";
    }
    @Override
    public int compareTo(Student o) {
       int t = marks - o.marks;
       if(t==0)  return o.name.compareTo(name);
       else      return t;
    }
}
```