BiTS
from collecting... to connecting...

BITS College

Undergraduate Program

Course Code: SE132

Course Title: Object Oriented Programming

BITS
from collecting... to connecting...

Chapter 4: Polymorphism

Contents

- What is Polymorphism ?
- Types of Polymorphism
- Why polymorphism
- Advantage of polymorphism

BITS
from collecting... to connecting...

What is polymorphism ?

- is the ability of an object to take many forms.
  - It allows us to perform the same action in many different ways.

- **Polymorphism occurs when there is inheritance**, i.e. there are many classes that are related to each other.

Cont...

Example 1: A person's relationships with other people

```
class Shapes {
  public void area() {
    System.out.println("The formula for area of ");
  }
}
class Triangle extends Shapes {
  public void area() {
    System.out.println("Triangle is ½ * base * height ");
  }
}
class Circle extends Shapes {
  public void area() {
    System.out.println("Circle is 3.14 * radius * radius ");
  }
}
```

Cont...


```
class Main {
  public static void main(String[] args) {
    Shapes myShape = new Shapes();  // Create a Shapes object
    Shapes myTriangle = new Triangle();  // Create a Triangle object
    Shapes myCircle = new Circle();  // Create a Circle object
    myShape.area();
    myTriangle.area();
    myShape.area();
    myCircle.area();
  }
}
```

Types of polymorphism

BITS
from collecting... to connecting...

- Method Overloading
  - is a process that can create multiple methods of the same name in the same class, and all the methods work in different ways.
  - Method overloading occurs when there is more than one method of the same name in the class.
- Method Overriding
  - is a process when the subclass or a child class has the same method as declared in the parent class.

Cont...

Compile Time

- The call to the method is resolved at compile-time
- It is achieved through Method Overloading

```
public class Addition {
void sum(int a, int b) {
        int c = a+b;
        System.out.println(" Addition of two numbers :" +c);
 }
void sum(int a, int b, int e) {
        int c = a+b+e;
        System.out.println(" Addition of three numbers :" +c);
 }
public static void main(String[] args) {
        Addition obj = new Addition();
        obj.sum ( 30,90);
        obj.sum(45, 80, 22);
}
}
```

Cont...

- Run time
  - The call to an overridden method is resolved dynamically at runtime rather than at compile-time
  - It is achieved through Method Overriding
  - Overriding is done by using a reference variable of the superclass
  - Which method to be called is determined based on the object which is being referred to by the reference variable. This is also known as Upcasting
  - Upcasting is done when the Parent class's reference variable refers to the object of the child class

Cont...

Example

```
class Animal{
  void eat(){
      System.out.println("Animals Eat");
   }
 }
class herbivores extends Animal{
  void eat(){
      System.out.println("Herbivores Eat Plants");
   }
 }
class omnivores extends Animal{
  void eat(){
      System.out.println("Omnivores Eat Plants and meat");
    }
  }
```

Cont…

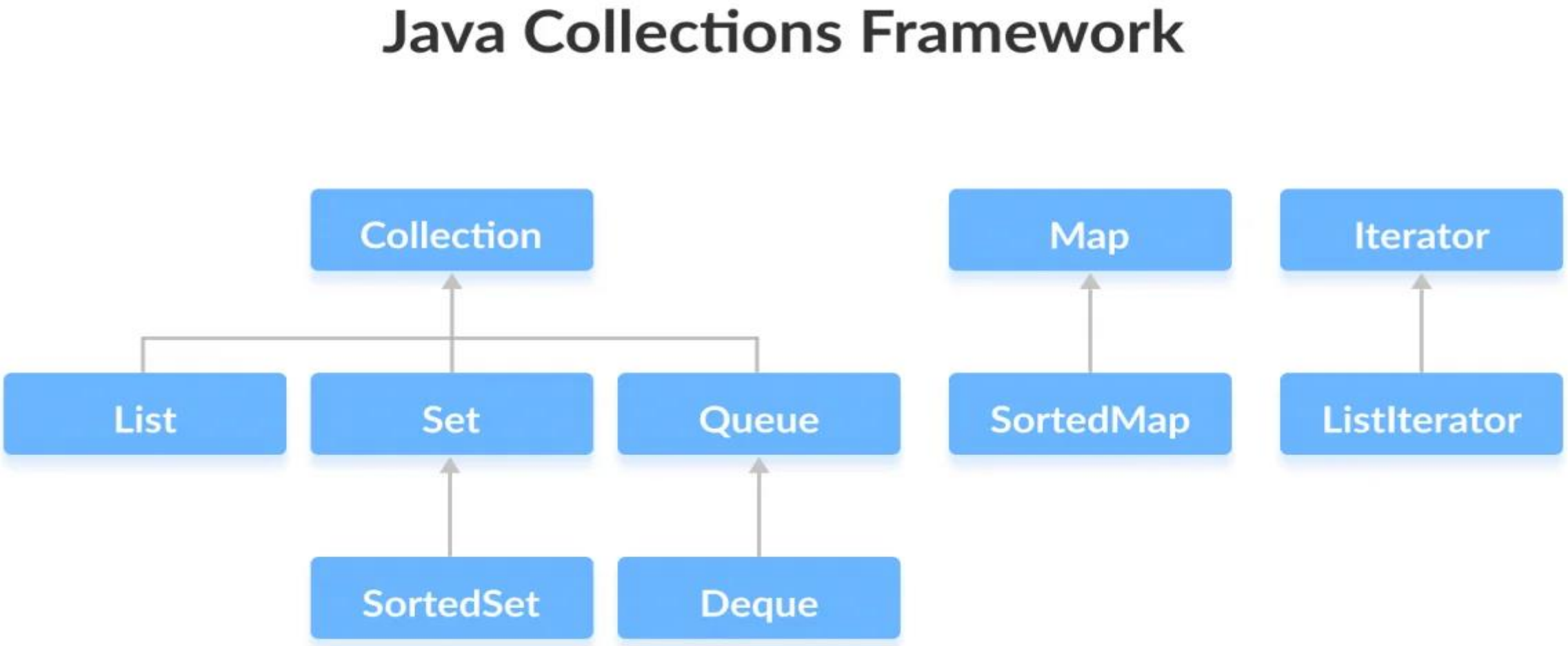

```
class carnivores extends Animal{
  void eat(){
System.out.println("Carnivores Eat meat");
   }
}
class main{
  public static void main(String args[]){
    Animal A = new Animal();
    Animal h = new herbivores(); //upcasting
    Animal o = new omnivores(); //upcasting
    Animal c = new carnivores(); //upcasting
    A.eat();
    h.eat();
    o.eat();
    c.eat();
  }

}
```

Java Collection Framework

BITS
from collecting... to connecting...

- Collections are containers that groups multiple items in a single unit
  - Are dynamic in the sense that data can be added or removed at run time
- It provides the architecture to store and manipulate a group of objects
- Java collections allow us to do various operations, through methods, on the data stored in the collection
- It provides us a set of interfaces and classes to store and manipulate data
- Found in java.util package

Cont...



Why the Collection Framework

BITS
from collecting... to connecting...

- We do not have to write code to implement these data structures and algorithms manually
- Our code will be much more efficient as the collections framework is highly optimized.
- Moreover, the collections framework allows us to use a specific data structure for a particular type of data. Here are a few examples,
  - If we want our data to be unique, then we can use the Set interface provided by the collections framework.
  - To store data in key/value pairs, we can use the Map interface
  - The ArrayList class provides the functionality of resizable arrays.

Methods of Collection

- **add()** - inserts the specified element to the collection
- **size()** - returns the size of the collection
- **remove()** - removes the specified element from the collection
- **iterator()** - returns an iterator to access elements of the collection
- **addAll()** - adds all the elements of a specified collection to the collection
- **removeAll()** - removes all the elements of the specified collection from the collection
- **clear()** - removes all the elements of the collection

List

- Is an interface that extends the collection interface
- Stores elements in an indexed approach
- Types of list (classes implementing list interface)
  - Array list
    - Allows dynamic addition or removal of data
  - Linked list
    - A sequence of links which contains item
    - Each list a connection to another list
  - Vector
    - Same as array list, but it is thread safe
  - Stack

Array List

BITS
from collecting... to connecting...

- It provides the functionality of resize-able array
- Usage
  - **ArrayList<Type> arrayList= new ArrayList<>();**
  - **e.g. ArrayList<Integer> arrayList = new ArrayList<>();**
- Common operations
  - Add
    - add(value), add(index, value)
  - Access
    - get(index), returns the value at the specified index
  - Change/ update
    - set(index, newValue)
  - Remove
    - remove(index), removeAll(), clear()
  - Other methods, contains(), sort(), size(), clone(), ...

Vector

- Allows us to create resize-able array like ArrayList class
- The difference b/n vector and array list is synchronization
- But, it is less efficient compared to array list
  - **Vector<Type> vector = new Vector<>();**
- Common operations
  - Add
    - add(value), add(index, element), add(vector)
  - Access
    - get(index)
  - Remove
    - remove(index), removeAll(), clear()

Stack

- Provides the functionality of stack data structure
- It extends vector class
- Elements are stored and accessed in last in first out manner
  - **Stack<Type> stacks = new Stack<>();**
- Common operations
  - Push – adds a value at the top a stack
    - push(value)
  - Pop - removes the top value from a stack
    - Pop()
  - Peek – returns an object from the top of the stack
    - Peek()
  - Search – returns the position an element from the top of the stack

Queue Interface

- Provides the functionality of the queue data structure
- Classes implementing queue interface are
  - ArrayDeque
  - PriorityQueue
  - LinkedList
- Elements are stored and accessed in First In, First Out manner

Priority Queue

BITS
from collecting... to connecting...

- Elements are retrieved in sorted order
  - **PriorityQueue<Integer> numbers = new PriorityQueue<>();**
- The elements may not be sorted
- Methods
  - Insert elements
    - add(value), offer(value)
  - Access
    - Peek() - returns the head of the queue
  - Remove
    - remove(element), poll() - returns and remove the head of the queue

ArrayDeque

BITS
from collecting... to connecting...

- Allows operation on both side of the queue
  - **ArrayDeque<Type> animal = new ArrayDeque<>();**
- Methods
  - Insert elements
    - add(value), addFirst(value), addLast(value), offer(value), offerFirst(value), offerLast(value)
  - Access
    - getFirst(), getLast(), peek(), peekFirst(), peekLast()
  - Remove
    - remove(element), remove(), removeFirst(), removeLast(), poll(), pollFirst(), pollLast(), clearAll()

Map Interface

- Elements of Map are stored in key/value pairs
- Keys are unique values associated with individual values
- We can access and modify values using the keys associated with them

Methods of Map Interface

- put(K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
- putAll() - Inserts all the entries from the specified map to this map.
- putIfAbsent(K, V) - Inserts the association if the key K is not already associated with the value V.
- get(K) - Returns the value associated with the specified key K. If the key is not found, it returns null.
- getOrDefault(K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.
- containsKey(K) - Checks if the specified key K is present in the map or not.
- containsValue(V) - Checks if the specified value V is present in the map or not.
- replace(K, V) - Replace the value of the key K with the new specified value V.
- replace(K, oldValue, newValue) - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
- remove(K) - Removes the entry from the map represented by the key K.
- remove(K, V) - Removes the entry from the map that has key K associated with value V.
- keySet() - Returns a set of all the keys present in a map.
- values() - Returns a set of all the values present in a map.
- entrySet() - Returns a set of all the key/value mapping present in a map.

Hash Map

- Creating map using HashMap class
  - HashMap<String, Integer> numbers = new HashMap<>();
- Adding values to the map
  - put(key,value)
- Accessing elements
  - get(key)
- Changing values
  - replace(key, newValue)
- Removing value
  - Remove(key)