Core Java

-AMRUTA NADGONDE

Chapter 4

OOPS CONCEPTS

Inheritance

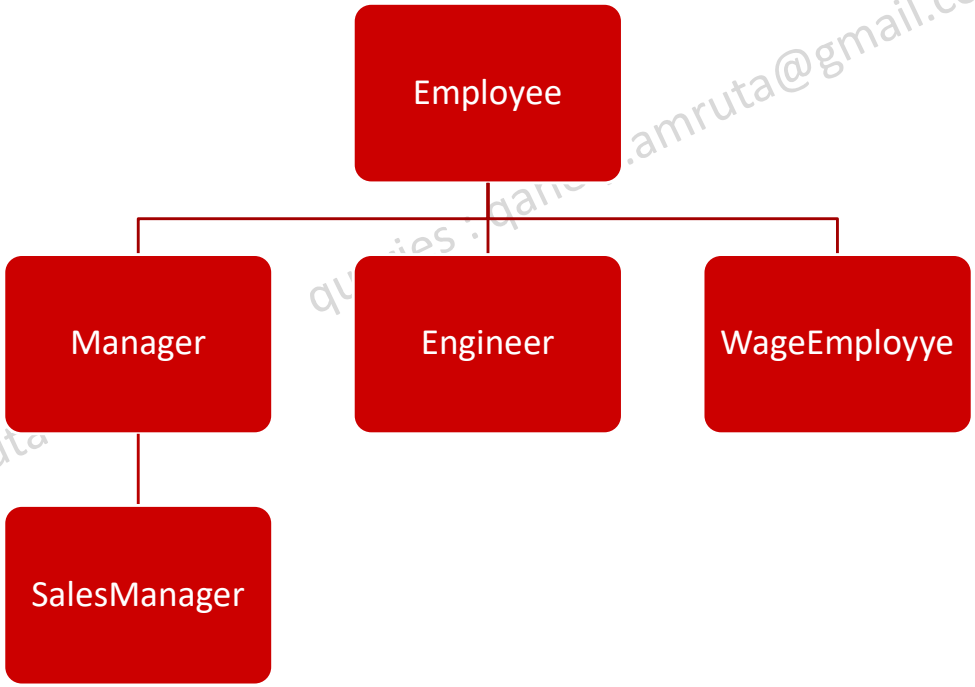
- One of the powerful features of object oriented programming.
- It is an “is-a” kind of a relationship.
- To inherit means to receive properties of already existing class.
- All data members and methods are inherited except the private fields.
- Hierarchy of classes is created using inheritance; new class (derived) is derived from an existing(base) class.

Advantages:

- Reusability: Once a class written and tested, it can be used to create new classes.
- Extensibility : As new classes can be derived from existing class ;this provides the extensibility of adding and removing classes in a hierarchy as and when needed.

Inheritance

is-a

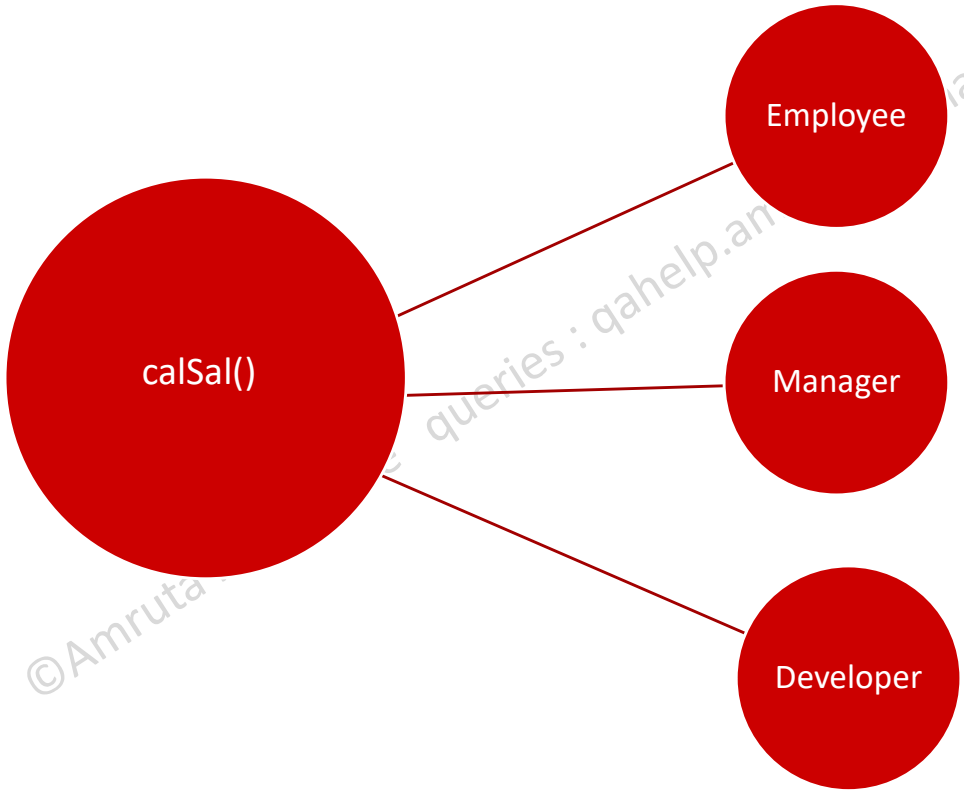


‘super’ Keyword

- When a class is derived from another class; member initialization of base class in derived class is achieved by using ‘***super’*** keyword.
- Constructor of the derived class first invokes the constructor of super class.
- If explicit call to parameterized constructor is not given default no argument constructor is called.
- ‘***super’*** keyword is used to call any non static method of the base class from the derived class.
- When ‘***super’*** *is used* for method invocation, it explicitly means base class implementation of the method.

Polymorphism

- "Poly" means many and "morph" means form.
- Ability to make more than one form is called polymorphism.
- In programming terminology, one command may invoke different implementation for different related objects.
- Allows different objects to share the same external interface although the implementation may be different.
- Can be achieved in two ways – Compile time polymorphism and run time polymorphism.



Compile time and Late Binding

- Binding is the process of associating a function call to an object
- **Compile time polymorphism (early binding)** – binding occurs at compile time. The argument passed to the method decides the method to be invoked.
- Achieved using **method overloading**.
- **Run time polymorphism** (late binding) - binding occurs at run time. The method to be invoked is decided based on the object used to invoke it.
- Achieved using **method overriding**

Method overriding

- Overriding is directly related to sub classing.
- Overriding is required to modify super class methods as required in sub class.
- Overridden methods should have same argument list of identical type and order else they are considered as overloaded methods.
- Return type of overridden method must be same else compiler generates an error.

Method Overloading vs Overriding

	Overloading	**Overriding**
Scope	In the same class	In the inherited classes
Purpose	Handy for program design	Specific implementation of Derived class
Signature of Methods	Different for each method	Has to be same in all the classes
Return Type	Can be same or different	Has to be same in all the classes

Up-casting and down-casting

- Up-casting: In inheritance super class reference can refer to a derived class object without needing a cast.

```
Employee e = new WageEmployee();

int sal = e.calSal(); //invokes derived class implementation
```

- Down-casting : In situations where specific methods of derived class need to be invoked using base class reference explicit casting is required; called as down-casting

```
WageEmployee we = (WageEmployee) e

int wages = we.calWages();          //specific method of WageEmployee
```

Covariant Return Types

- In method overriding, overridden method in subclass should have signature same as that of the super class.
- In Java 5 return type of a overridden method can be changed as long as the new return type is the subtype of the declared type.

```
class Employee{
        public Number getEmployees()
        {
                return EmployeeCount();
}

class SalesExe extend Employee{
        public Integer getEmployees()
        {
                return SalesExeCount();
        }  }
```

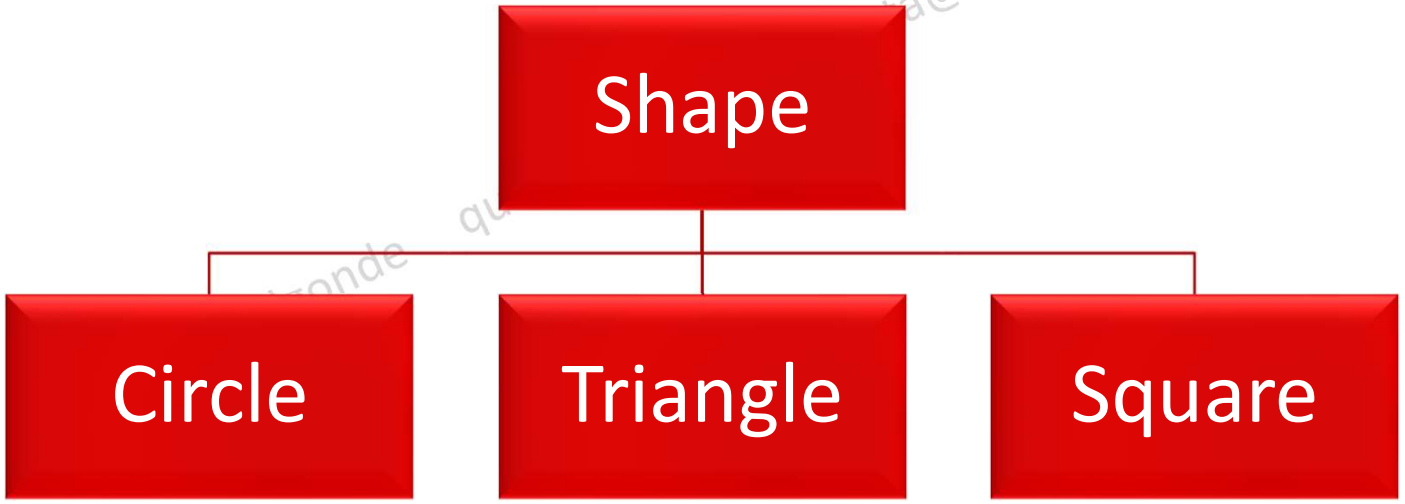
‘final’ Methods and Classes

- The main purpose of a final class/method is to take away inheritance feature from user so the class cannot be derived or the method cannot be overridden.
- ‘***final’*** keyword is used to make a class /method final
- Final method cannot be overridden in a subclass
- All methods in a final class are final by default. A final class cannot be sub classed.
- ‘private’ methods are final by default.
- In Java ‘String’ class is a final class.

Abstract Class

- Abstract Class: A class which contain generic/common features that multiple derived classes can share.
- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon) :
- A class which has at least one abstract method is called abstract class. Other methods of the class can be abstract / non abstract.
- Object of abstract class can not be created but reference can be.
- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Abstract Class



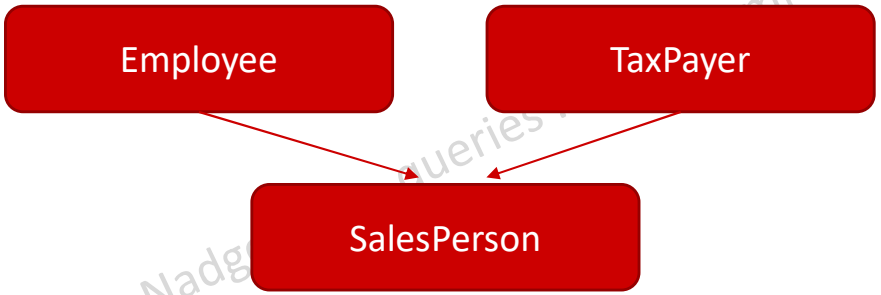
Abstract Class

- Abstract class cannot be instantiated
- Abstract class must contain at least one abstract method.
- Abstract methods do not have any implementation.
- One can create references so these classes support polymorphism
- Abstract classes are very useful during component creation because they allow specifying an invariant level of functionality in some method but leave the implementation of the other methods until a specific implementation of that class is needed.
- A class inheriting from an abstract class must provide implementation to all the abstract methods, else the class should be declared as abstract.
- Abstract modifier cannot be used for constructor

Interface

Need :

- It is possible for entities belonging to two different hierarchy sets to belong to same role:



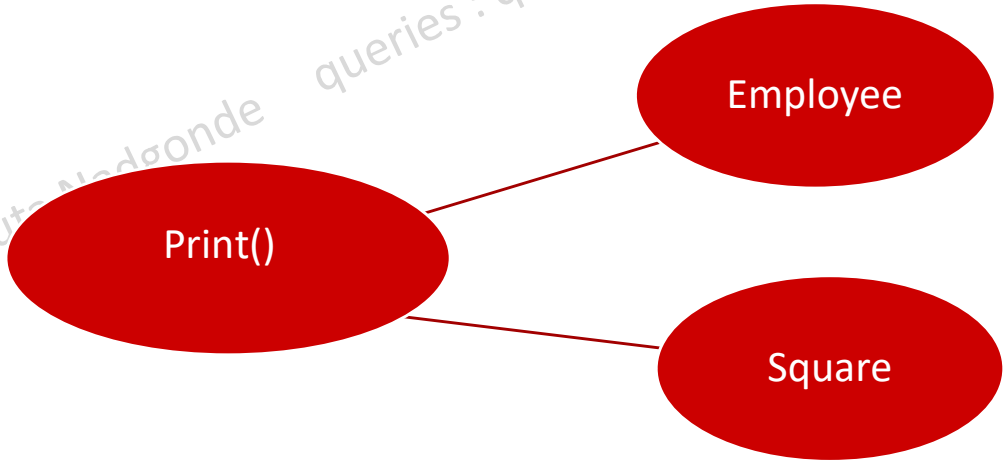
- Allows role-based inheritance.

Role → Interface

- To create loosely coupled applications

Interface

- Interface is a contract between the consumer and provider.
- It defines a standard set of properties, methods and events.
- Any class which implements the interface has to provide the implementation to all the members of the interface.



Interface

- All the methods declared in an interface are implicitly *abstract.*
- Methods in an interface are always *public*, they cannot be *static*. (static methods are always class specific).
- All fields in an interface are always *static* and *final*.
- A variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed using this variable.
- An interface can extend another interface just like a class can extend another class. However unlike inheritance classes can implement multiple interfaces.
- A class that implements the interface should give the implementation of all the methods declared in an interface otherwise, must declare as an abstract.
- '*implements'* is the keyword used to implement an interface. If class implements multiple interface, then it can be done using comma-delimited list.

Chapter 5

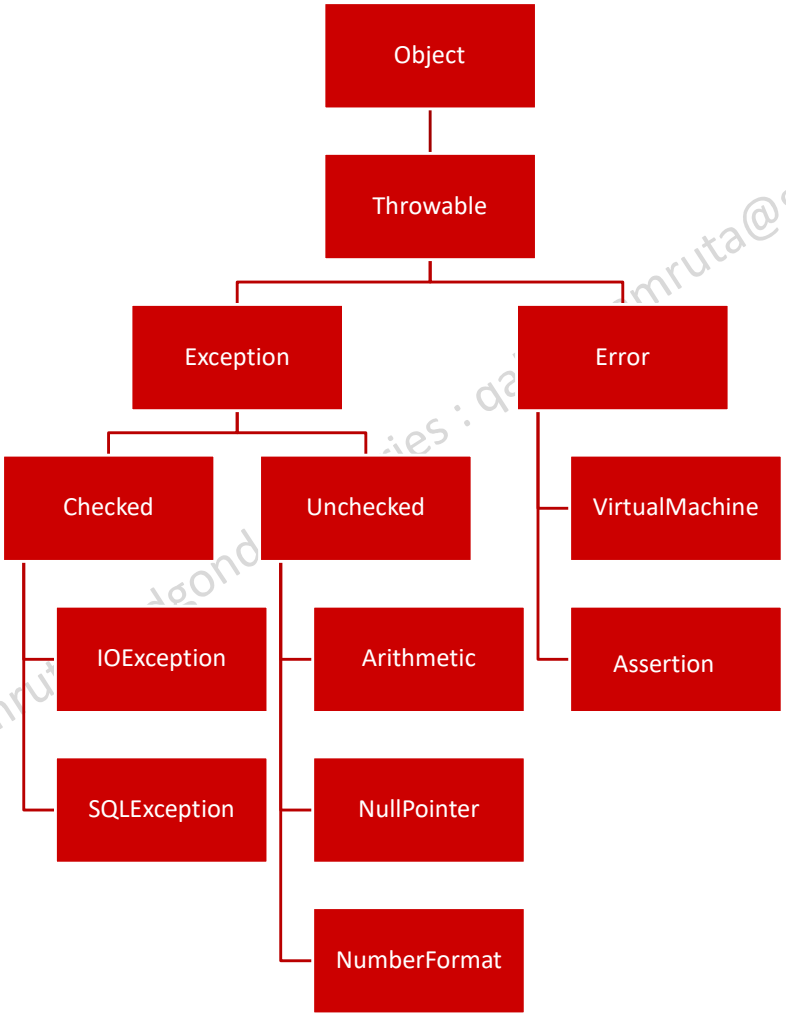
EXCEPTION HANDLING

Errors

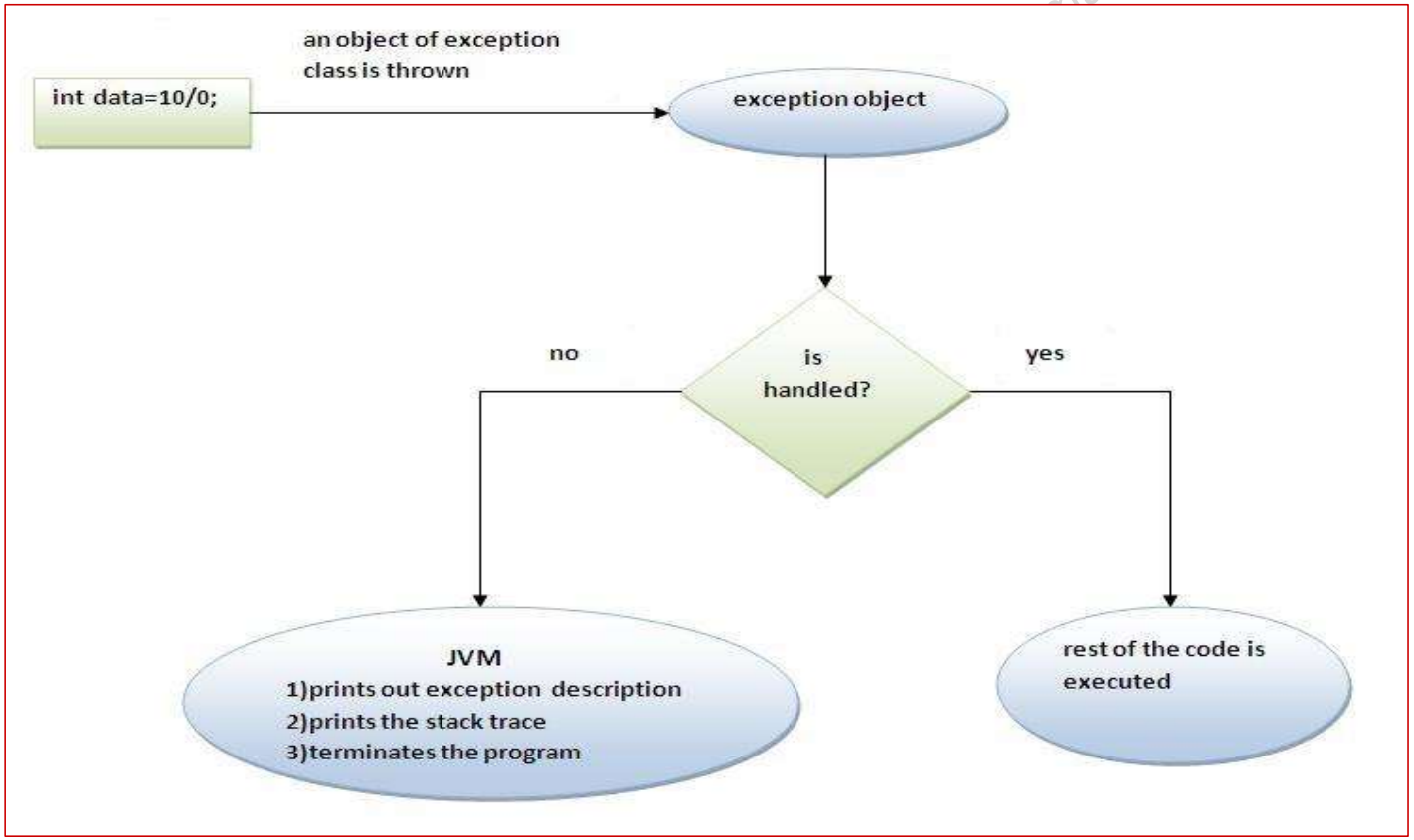
- Programs cannot be 100% error-free
- Logical errors:
  - Occur when there is something wrong with the logic used by the programmer.
  - Can be corrected by debugging and dry runs.
- Run Time errors
  - Occur at run-time, when the program is executed.
  - Can be controlled by predicting the error and checking appropriate condition.

Limitations of C-Style Error Handling

- No separate mechanism for error handling.
- Error values are checked using if..else/switch statements to take appropriate action and generation of error messages.
- Business logic gets secondary importance whereas error handling is given primary importance
- Java - structured way of handling these runtime errors.
- Exception handling - mechanism that provides a way to respond to run time errors by transferring control to special code called handler.
- Achieved using ‘try-catch’ blocks.


Object
Throwable
Exception
Error
Checked
Unchecked
VirtualMachine
IOException
Arithmetic
Assertion
SQLException
NullPointer
NumberFormat

Internal working of ‘try-catch’



Multiple ‘try-catch’ Blocks

- Single try-Multiple catch blocks to handle individual errors.
- ‘try’ block executes normally unless exception occurs.
- When exception occurs, the system searches for the nearest catch block which matches the type of exception and handles it.
- Object of *Exception* class catches all exceptions that are not handled by any catch block.
- All catch blocks must be ordered from most specific to most general.

‘finally’ Block

- Exceptions occurring in the application before cleaning up (such as closing connection, stream etc) must be handled.
- Clean-up code cannot be written in ‘try’ block.
- Each ‘catch’ block needs to have own copy of clean-up.
- The solution is ‘finally’ block.
- A finally block is always executed where there is an exception or not

‘throw’ Keyword

- Used to explicitly throw an exception.
- Used for both checked or unchecked exceptions.
- Mainly used to throw custom exception

```
throw new IOException ("sorry device error);
```

‘throws’ Keyword

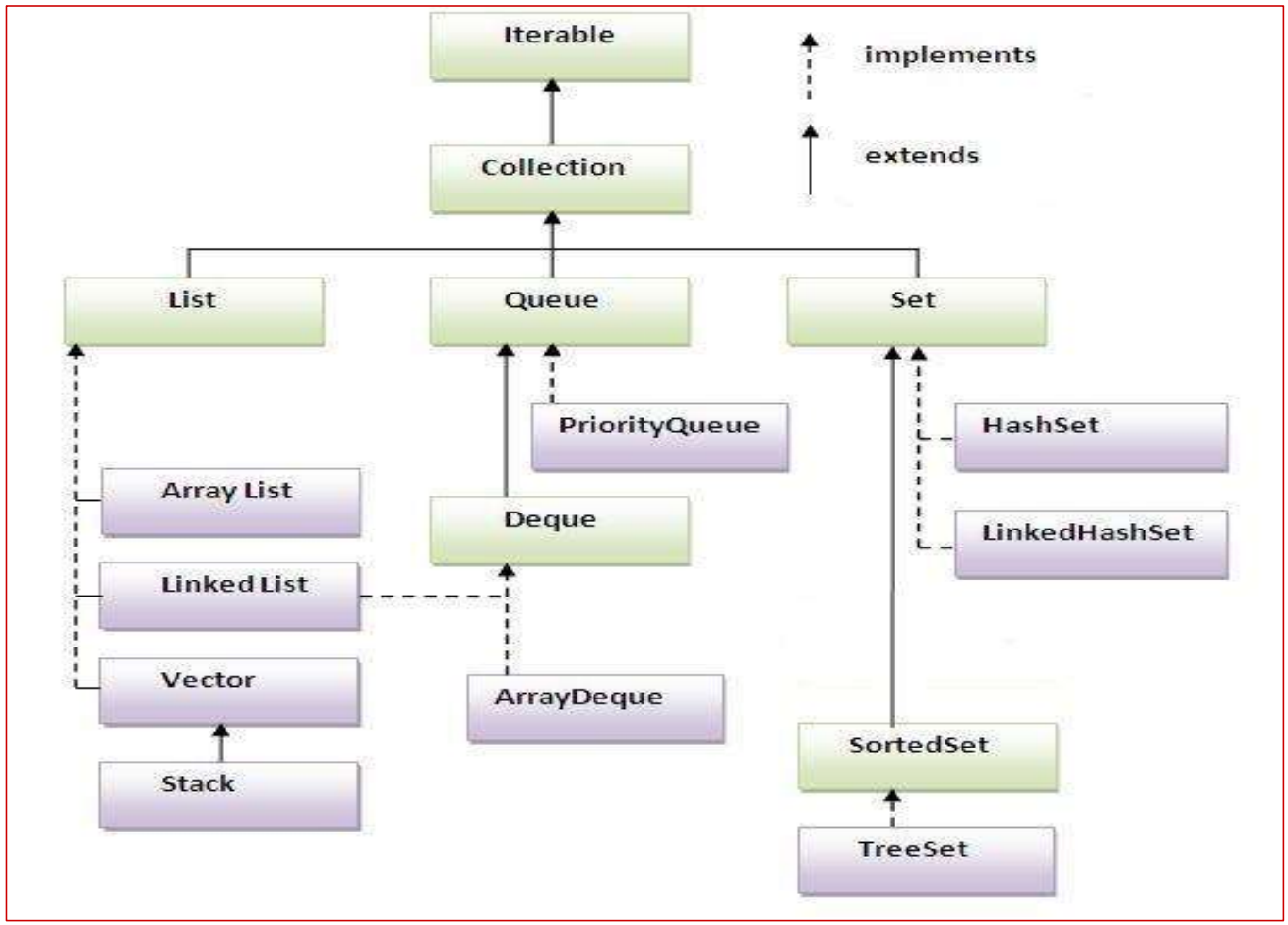
- Exception propagation – Exception is thrown from the top, drops down the call stack until it is caught.
- If a method that is implementing a particular business logic doesn’t want to handle the exception it throws it to the calling method.
- ‘***throws’*** is the keyword used to declare an exception.
- ***‘throws’*** is mandatory in case of checked exceptions otherwise compiler will give an error.
- In case of ‘**unchecked**’ exception, exceptions can be handled anywhere in the call stack.

Chapter 6

JAVA COLLECTION

Collections

- Why Collections?
  - Arrays are used to store a set of data, but their size is fixed.
  - insertion and deletion in the array is also costly in terms of performance.
  - In Actual development scenario, objects need to processed – added, deleted, sorted dynamically.
- Collections in java is a framework that provides an architecture to store and manipulate the group of objects.
- Operations such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections


Iterable
implements
extends
Collection
List
Queue
Set
PriorityQueue
HashSet
Array List
Deque
LinkedHashSet
Linked List
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Collection Interface

- Basic operations that are possible on any collection:
  - Add an object dynamically
  - Remove an object dynamically
  - List the objects by Iterating through the collection
  - Search specific object from collection
  - Retrieve an object from the collection
  - Sort the objects of the collection
- These common operations are defined in the form of interface.

List, List<E> interface

- Represents a collection of objects that can be accessed by an index.
- **Classes**: ArrayList, LinkedList, Vector.
- In List interface
  - Duplicates are allowed
  - Order is fixed
- **ArrayList** : Extends AbstractList class and implements List interface. Uses a dynamic array for storing the elements.
- **LinkedList**: Another Class that implements List interface, internally uses **doubly linked list** to store the elements.

Set, Set<E> interface

- Represents a collection that contains no duplicates.
- In Set interface order is not fixed.
- **Classes**: HashSet (unordered), LinkedHashSet (ordered), TreeSet(ascending order)
- **HashSet**: Extends Abstract Set class and implements Set interface
- Uses hashtable to store the elements.
- **TreeSet**: implements NavigableSet interface that extends the SortedSet interface. Maintains ascending order.

Map Interface

- Contains values based on the key i.e. key and value pair. (known as Entry)
- Map contains only unique elements.
- **Classes**: HashMap (unordered), LinkedHashMap (ordered), TreeMap (ascending order)

- **HashMap**: Extends AbstractMap class and implements Map interface
- Contains values based on the key.
- May have one null key and multiple null values.
- Maintains no order.

Iterator Interface

- Allows user to visit the elements in collection one by one
- Contains three methods:
  - Object next()
  - boolean hasNext()
  - void remove()


```
Iterator i = li.iterator();
while(i.hasNext())
{
        System.out.println(i.next());
}
```

Sorting of Collection Objects

- Sorting can be applied to:
  - String Objects
  - Wrapper class Objects
  - User-defined class Objects

- **Collections** class provides static methods for sorting the elements of collection.
- **public void sort (List list)** is the method used to sort List elements.
- If collection elements are of Set type, we can use TreeSet.

Comparable Interface

- Collections.sort() method provides sorting of List elements that are comparable.
- String class and Wrapper classes implement Comparable interface so sort () method can be directly used for these class objects.
- Comparable interface allows sorting of objects of user-defined class.
- Available in java.lang package.
- Provides single sorting sequence i.e. objects can be sorted based on only one data member.
- Provides **public int compareTo (Object obj)** method to compare objects of user-defined class.

Comparator Interface

- Used to order the objects of user-defined class.
- Available in java.util package.
- It provides multiple sorting sequence i.e. you can sort the elements based on any data member.
- Contains two methods to compare objects of user-defined class:
  - **public int compare(Object obj1,Object obj2):** compares the first object with second object.
  - **public void sort(List list,Comparator c):** is used to sort the elements of List by the given comparator.

Why Generics?

- All objects in non-generic collections are stored as Object type.
- Adding and retrieving values in such case requires essential type-cast.
- There is no guarantee of type safety with non-generic collections.
- Generics, forces the java programmer to store specific type of objects.

Advantages:

- Type-safety
- Type casting is not required
- Compile time checking

Generics Class

- A class that can refer to any type is known as generic class.
- **T** type parameter is used to create the generic class of specific type.

```
public class DemoGenerics <T> {
        T obj;
        public String getType(){
        return (obj.getClass().getName());
        }
public static void main(String[] args) {
        DemoGenerics<String> demo = new      DemoGenerics<>("Demo");
        System.out.println(demo.getType());  //returns     java.lang.String
}
```