Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like
'int','char' or 'double'.

```
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T>
{
   // An object of type T is declared
   T obj;
   Test(T obj) {  this.obj = obj;  }  // constructor
   public T getObject()  { return this.obj; }
}

// Driver class to test above
class Main
{
   public static void main (String[] args)
   {
      // instance of Integer type
      Test <Integer> iObj = new Test<Integer>(15);
      System.out.println(iObj.getObject());

      // instance of String type
      Test <String> sObj =
                  new Test<String>("Hello world");
      System.out.println(sObj.getObject());
   }
}
```

Output:

```
15
Hello world
```

We can also pass multiple Type parameters in Generic classes.

```
// A Simple Java program to show multiple
// type parameters in Java Generics
```

```
// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("Welcome", 15);

        obj.print();
    }
}
```

Output:

```
Welcome
15
```

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
```

```
        System.out.println(element.getClass().getName() +
                           " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
         // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("Java class");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

Output :

```
java.lang.Integer = 11
java.lang.String = Java class
java.lang.Double = 1.0
```

Advantages of Generics:

Programs that uses Generics has got many benefits over non-generic code.

Code Reuse: We can write a method/class/interface once and use for any type we want.

Type Safety : Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

```
// A Simple Java program to demonstrate that NOT using
// generics can cause run time exceptions
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creatinga an ArrayList without any type specified
        ArrayList al = new ArrayList();

        al.add("Sachin");
        al.add("Rahul");
        al.add(10); // Compiler allows this
```

```
        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        // Causes Runtime Exception
        String s3 = (String)al.get(2);
    }
}
```

Output :

```
Exception in thread "main" java.lang.ClassCastException:
  java.lang.Integer cannot be cast to java.lang.String
      at Test.main(Test.java:19)
```

How generics solve this problem?
At the time of defining ArrayList, we can specify that this list can take only String objects.

```
// Using generics converts run time exceptions into
// compile time exception.
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

.

Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

```
// We don't need to typecast individual members of ArrayList
import java.util.*;

class Test
{
   public static void main(String[] args)
   {
      // Creating a an ArrayList with String specified
      ArrayList <String> al = new ArrayList<String> ();

      al.add("Sachin");
      al.add("Rahul");

      // Typecasting is not needed
      String s1 = al.get(0);
      String s2 = al.get(1);
   }
}
```

.

Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

Need for Collection Framework :

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

```
// Java program to show why collection framework was needed
import java.io.*;
import java.util.*;

class Test
{
        public static void main (String[] args)
        {
                // Creating instances of array, vector and hashtable
                int arr[] = new int[] {1, 2, 3, 4};
                Vector<Integer> v = new Vector();
```

```
                Hashtable<Integer, String> h = new Hashtable();
                v.addElement(1);
                v.addElement(2);
                h.put(1,"Hello");
                h.put(2,"World");

                // Array instance creation requires [], while Vector
                // and hastable require ()
                // Vector element insertion requires addElement(), but
                // hashtable element insertion requires put()

                // Accessing first element of array, vector and hashtable
                System.out.println(arr[0]);
                System.out.println(v.elementAt(0));
                System.out.println(h.get(1));

                // Array elements are accessed using [], vector elements
                // using elementAt() and hashtable elements using get()
        }
}

op:1
1
Hello
```

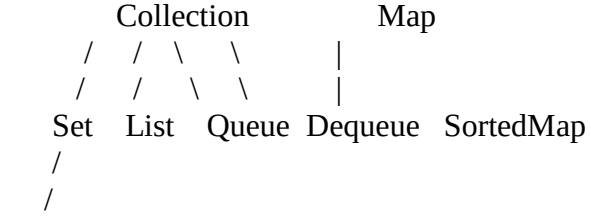
As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection.
Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2.

Both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

Advantages of Collection Framework:

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. Reduces programming effort: A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

Hierarchy of Collection Framework



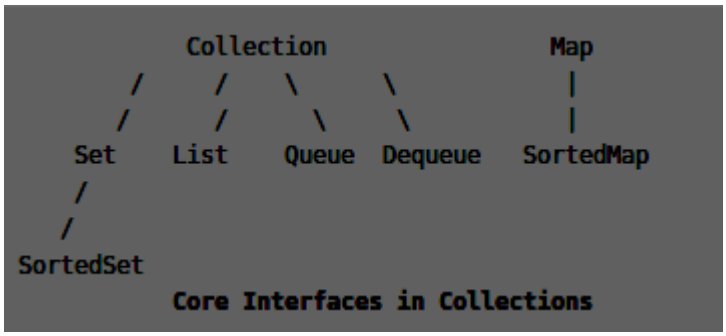
SortedSet

Core Interfaces in Collections

Note that this diagram only shows core interf

List Interface in Java with Examples

The Java.util.List is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.



Creating List Objects:

List is an interface, and the instances of List can be created in the following ways:

```
List a = new ArrayList();
List b = new LinkedList();
List c = new Vector();
List d = new Stack();
```

Generic List Object:

After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. The type-safe List can be defined in the following way:

```
// Obj is the type of object to be stored in List.
List<Obj> list = new ArrayList<Obj> ();
```

Operations on List:

List Interface extends Collection, hence it supports all the operations of Collection Interface, along with following additional operations:

1. **Positional Access:**
   List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:
   - **void add(int index,Object O):** This method adds given element at specified index.
   - **boolean addAll(int index, Collection c):** This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.

- **Object remove(int index):** This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
- **Object get(int index):** This method returns element at the specified index.
- **Object set(int index, Object new):** This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

```
// Java program to demonstrate positional access
// operations on List interface
import java.util.*;

public class ListDemo
{
	public static void main (String[] args)
	{
		// Creating a list
		List<Integer> l1 = new ArrayList<Integer>();
		l1.add(0, 1); // adds 1 at 0 index
		l1.add(1, 2); // adds 2 at 1 index
		System.out.println(l1); // [1, 2]

		// Creating another list
		List<Integer> l2 = new ArrayList<Integer>();
		l2.add(1);
		l2.add(2);
		l2.add(3);

		// Will add list l2 from 1 index
		l1.addAll(1, l2);
		System.out.println(l1);

		// Removes element from index 1
		l1.remove(1);
		System.out.println(l1); // [1, 2, 3, 2]

		// Prints element at index 3
		System.out.println(l1.get(3));

		// Replace 0th element with 5
		l1.set(0, 5);
		System.out.println(l1);
	}
}
```

Output:

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
```

Search:
List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:

- **int indexOf(Object o):** This method returns first occurrence of given element or -1 if element is not present in list.
- **int lastIndexOf(Object o):** This method returns the last occurrence of given element or -1 if element is not present in list.

```
// Java program to demonstrate search
// operations on List interface
import java.util.*;

public class ListDemo
{
	public static void main(String[] args)
	{
		// Type safe array list, stores only string
		List<String> l = new ArrayList<String>(5);
		l.add("welcome:");
		l.add("class");
		l.add("Java ");
		l.add(“class”);

		// Using indexOf() and lastIndexOf()
		System.out.println("first index of class:" +
								l.indexOf("class"));
		System.out.println("last index of class:" +
							l.lastIndexOf("class"));
		System.out.println("Index of element"+
			" not present : " + l.indexOf("Hello"));
	}
}
```