Object Oriented Programming
CS F213

J. Jennifer Ranjani
email: jennifer.ranjani@pilani.bits-pilani.ac.in
Chamber: 6121 B, NAB
Consultation: Appointment by e-mail

BITS Pilani
Pilani Campus

Queries asked during the previous class

Difference between String Tokenizer & split method in String class

```
String str = "http://www.java.com/strings";

StringTokenizer st = new StringTokenizer(str, "://");
for (int i = 0; st.hasMoreTokens(); i++) System.out.println("#" + i + ": " +
    st.nextToken());

String[] split1 = str.split("://");
for (int i = 0; i < split1.length; i++) System.out.println("#" + i + ": " +
    split1[i]);

String[] split2 = str.split(":|//|/");
for (int i = 0; i < split2.length; i++) System.out.println("#" + i + ": " +
    split2[i]);
```

Output

```
#0: http
#1: www.java.com
#2: strings
#0: http
#1: www.java.com/strings
#0: http
#1:
#2: www.java.com
#3: strings
```

Difference between String Tokenizer & split method in String class

- Split considers the delimiter as a regular expression.
- Two consecutive delimiter or delimiters in the beginning cause the split method to return blank strings
- String Tokenizer is fast compared to split
- String Tokenizer can not group the delimiters

Capacity of the String Buffer

- If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2

```
StringBuffer sb = new StringBuffer("Java");
System.out.println(sb.capacity());

sb.append(" Programming");
System.out.println(sb.capacity());

sb.append("is ea");
System.out.println(sb.capacity());
```

Output:

```
20
20
42
```

ensureCapacity of the StringBuffer

```
StringBuffer sb = new StringBuffer("Java is my favourite language");
System.out.println(sb.capacity());

sb.ensureCapacity(10);
System.out.println(sb.capacity());

sb.ensureCapacity(50);
System.out.println(sb.capacity());
```

Output:

```
45
45
92
```

Generics (J2SE 5)

Generic Class

- Allows type to be a parameter to methods
- <> is used to specify the parameter types
- To create objects use the following syntax

  BaseType <Type> obj = new BaseType <Type>()

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

Generic Class - Example

```
class Identity<T>{
    T obj;
    Identity(T obj) {  this.obj = obj;  }
    public T getObject()  { return this.obj; }
}

class Test {
    public static void main (String[] args)     {

        Identity <Long> number = new Identity<Long>(9999955555L);
        System.out.println(number.getObject());

        Identity <String> name = new Identity<String>("Ankit");
        System.out.println(name.getObject());
    }
}
```

Multiple Type Parameters

```
class Identity<T,U> {
    T obj1; U obj2;
    Identity(T obj1,U obj2 ) {  this.obj1 = obj1;this.obj2 = obj2;  }
    public void printObject()  {
     System.out.print(this.obj1+"\t");System.out.println(this.obj2); }
}

class Test {
    public static void main (String[] args)     {
    Identity <String, Integer> I1 = new Identity<String,
     Integer>("Ankit",20171007);
    Identity <Integer,String> I2 = new
     Identity<Integer,String>(20171007,"Ankit");
    I1.printObject();
    I2.printObject();
    }}
```

Generic Functions

```
class Identity {
    public <T> void printObject(T obj)  {System.out.println(obj); }
}

class Test {
    public static void main (String[] args)     {
    Identity I1, I2;
    I1 = new Identity();
    I2 = new Identity();
    I1.printObject(20071007);
    I2.printObject("Ankit");
    }
}
```

Generic Functions with generic return type

```
class Identity {
    public <T> T printObject(T obj)  {return obj; }
}

class Test {
    public static void main (String[] args)     {
    Identity I1, I2;
    I1 = new Identity();
    I2 = new Identity();
    System.out.println(I1.printObject(20071007));
    System.out.println(I2.printObject("Ankit"));
    }
}
```

Bound Type with Generics

- Used to restrict the types that can be used as arguments in a parameterized type.
  - Eg: Method operating on numbers should accept the instances of the Number class or its subclasses.

- Declare a bounded type parameter
  - List the type parameter’s name.
  - Along by the extends keyword
  - And by its upper bound.

Bound Type - Example

```
class Identity<T extends Number> {
    T obj;
    Identity(T obj) {  this.obj = obj;  }
    public T getObject()  { return this.obj; }}
class Test {
    public static void main (String[] args)     {
        Identity <Integer> iObj = new Identity<Integer>(20071007);
        System.out.println(iObj.getObject());
        Identity <Double> dObj = new Identity<Double>(2007.00);
        System.out.println(dObj.getObject());
        Identity <String> sObj = new Identity<String>("Ankit");
        System.out.println(sObj.getObject());
    }}
```

Note: Bound Mismatch: type argument String is not within bounds of type-variable T

Collections

What are Collections

- Group of Objects treated as a single Object.

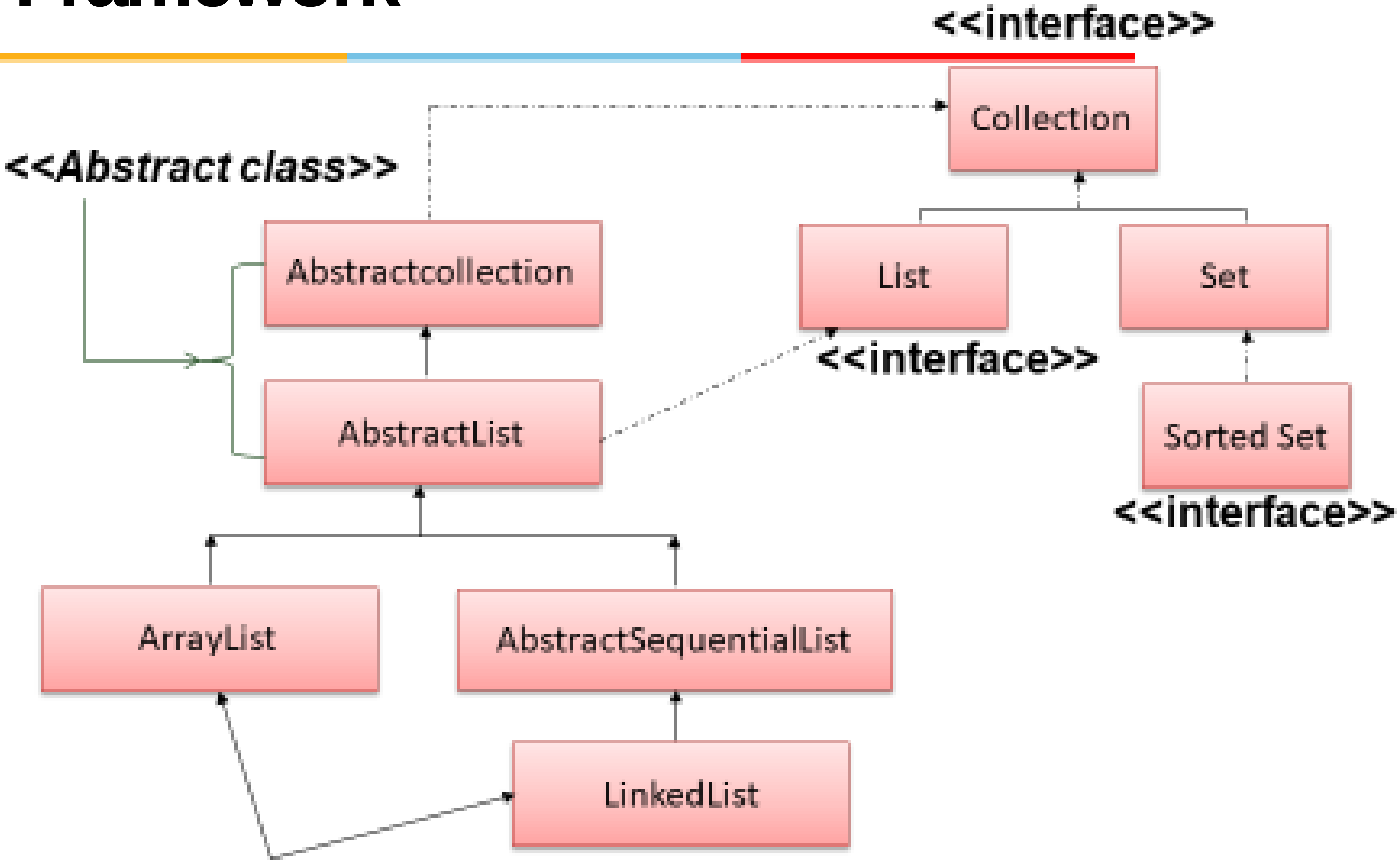
- Java provides supports for manipulating collections in the form of
  - Collection Interfaces
  - Collection Classes

- Collection interfaces provide basic functionalities whereas collection classes provides their concrete implementation

Collection Classes

- Collection classes are standard classes that implement collection interfaces
- Some Collection Classes are abstract and some classes are concrete and can be used as it is.
- Important Collection Classes:
  - ✓ AbstractCollection
  - ✓ AbstractList
  - ✓ AbstractSequentialList
  - ✓ LinkedList
  - ✓ ArrayList
  - ✓ AbstractSet
  - ✓ HasSet
  - ✓ LinkedHashSet
  - ✓ TreeSet

Partial View of Collection's Framework



Concrete Classes

ArrayList - Example

```
import java.util.*;
class Test{
public static void main(String args[]){
ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(20);
al1.add(9);

ArrayList<Integer> al2 = new ArrayList<Integer>();
al2.add(22);
al2.add(53);

al1.addAll(al2);
Collections.sort(al1);
System.out.println(al1);
System.out.println(al1.get(3));}
}
```

Output:

```
[9, 20, 22, 53]
53
```

List Iterator

- List Iterator is used to traverse forward and backward directions

Method	Description
boolean hasNext()	This method return true if the list iterator has more elements when traversing the list in the forward direction.
Object next()	This method return the next element in the list and advances the cursor position.
boolean hasPrevious()	This method return true if this list iterator has more elements when traversing the list in the reverse direction.
Object previous()	This method return the previous element in the list and moves the cursor position backwards.

List Iterator - Example

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(20);
al.add(9);
al.add(22);
al.add(53);

ListIterator<Integer> itr=al.listIterator();
System.out.println("Forward Traversal");
while(itr.hasNext()) {
System.out.println(itr.next());
}
System.out.println("Backward Traversal");
while(itr.hasPrevious()) {
System.out.println(itr.previous());
}
```

Output:

```
Forward Traversal
20
9
22
53
Backward Traversal
53
22
9
20
```

Question: What happens if the backward traversal happens before the forward?