Chapter 6 Collections / Generics

6.1 Given a design scenario, determine which collection classes and/or interfaces should be used to properly implement that design, including the use of the Comparable interface.

Summary of collections interfaces

- `Collection` - This is a basic set of methods for working with data structures. A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
- `List extends Collection` - An ordered collection. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
- `Map` (does NOT extend `Collection`) - An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. Stores key/value pairs, rapidly accessible by key.
- `SortedMap extends Map` - A map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys, or by a comparator provided at sorted map creation time. All keys inserted into a sorted map MUST implement the `Comparable` interface (or be accepted by the specified comparator).
- `Set extends Collection` - A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one `null` element.
- `SortedSet extends Set` - A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see `Comparable`), or by a `Comparator` provided at sorted set creation time.

Summary of general-purpose implementations

- `HashSet implements Set` - Hash table implementation of the `Set` interface. The best all-around implementation of the `Set` interface. This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.

  This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets.

  Note that this implementation is not synchronized.

  Class without overriden `hashCode()` and `equals(...)` methods:

```
/**
 * The WeirdString class uses the derived from 'Object'
 * implemenations of 'equals' and 'hashCode' methods
 */
public class WeirdString {
  private String str;

  WeirdString(String str ){
         this.str = str;
  }

  public String toString() {
         return "WeirdString : '" + str + "'";
  }
}

import java.util.HashSet;

public class CollClient {

  public static void main(String ... sss) {
          HashSet myMap = new HashSet();
          String s1 = new String("abcdef");
          String s2 = new String("abcdef");
          WeirdString s3 = new WeirdString("abcdef");
          WeirdString s4 = new WeirdString("abcdef");

          myMap.add(s1);
          myMap.add(s2);
          myMap.add(s3);
          myMap.add(s4);

          System.out.println(myMap);
  }
}
```

The output (3 objects in the set):

```
[abcdef, WeirdString : 'abcdef', WeirdString : 'abcdef']
```

The next example shows class with correctly overriden `hashCode()` and `equals(...)` methods:

```
/**
 * The WeirdStringFixed class correctly
 * implements 'equals' and 'hashCode' methods
 */
public class WeirdStringFixed {
  private String str;

  WeirdStringFixed(String str ){
          this.str = str;
  }

  public String getStr() {
          return str;
  }

  public boolean equals(Object o){
          if (!(o instanceof WeirdStringFixed)) {
                  return false;
          }

          return ((WeirdStringFixed) o).getStr().equals(str);
  }

  public int hashCode() {
          return 12345; // pretty legal
  }

  public String toString() {
          return "WeirdStringFixed : '" + str + "'";
  }
}
```

```
import java.util.HashSet;

public class CollClientFixed {

  public static void main(String ... sss) {
         HashSet myMap = new HashSet();
         String s1 = new String("abcdef");
         String s2 = new String("abcdef");
         WeirdStringFixed s3 = new WeirdStringFixed("abcdef");
         WeirdStringFixed s4 = new WeirdStringFixed("abcdef");

         myMap.add(s1);
         myMap.add(s2);
         myMap.add(s3);
         myMap.add(s4);

         System.out.println(myMap);
  }
}
```

The output (2 objects left in the set - duplicates were removed):

```
[abcdef, WeirdStringFixed : 'abcdef']
```

- `TreeSet implements SortedSet` - Red-black tree implementation of the `SortedSet` interface. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements (see `Comparable`), or by the comparator provided at set creation time, depending on which constructor is used.

  Note that this implementation is not synchronized.

  The `add(...)` method may throw `ClassCastException` if the specified object cannot be compared with the elements currently in the set:

```
/** Non-Comparable class */
public class WeirdString {
  private String str;

  WeirdString(String str ){
         this.str = str;
  }
}

import java.util.TreeSet;

public class CollClient {

  public static void main(String ... sss) {
         TreeSet myMap = new TreeSet();
         String s1 = new String("abcdef");
         String s2 = new String("abcdef");
         WeirdString s3 = new WeirdString("abcdef");
         WeirdString s4 = new WeirdString("abcdef");

         myMap.add(s1);
         myMap.add(s2);
         myMap.add(s3); // ClassCastException at runtime !!!
         myMap.add(s4);

         System.out.println(myMap);
  }
}
```

- `LinkedHashSet extends HashSet implements Set` - Hash table and linked list implementation of the `Set` interface. An insertion-ordered `Set` implementation that runs nearly as fast as `HashSet`. Hash table and linked list implementation of the `Set` interface, with predictable iteration order. This implementation differs from `HashSet` in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order).

  Note that this implementation is not synchronized.

- `ArrayList implements List` - Resizable-array implementation of the `List` interface. (Essentially an unsynchronized `Vector`.) The best all-around implementation of the `List` interface.

  Note that this implementation is not synchronized.

- `LinkedList implements List, Queue` - Doubly-linked list implementation of the `List` interface. May provide better performance than the `ArrayList` implementation if elements are frequently inserted or deleted within the list. Can be used as a double-ended queue (deque). Also implements the `Queue` interface. When accessed via the `Queue` interface, `LinkedList` behaves as a FIFO queue.

  Linked list implementation of the `List` interface. Implements all optional list operations, and permits all elements (including `null`). In addition to implementing the `List` interface, the `LinkedList` class provides uniformly named methods to `get`, `remove` and `insert` an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque).

  Note that this implementation is not synchronized.

- `HashMap implements Map` - Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

  Note that this implementation is not synchronized.

- `TreeMap implements SortedMap` - Red-black tree based implementation of the `SortedMap` interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class (see `Comparable`), or by the comparator provided at creation time, depending on which constructor is used.

  Note that this implementation is not synchronized.

- `LinkedHashMap extends HashMap implements Map` - Hash table and linked list implementation of the `Map` interface, with predictable iteration order. This implementation differs from `HashMap` in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

  Note that this implementation is not synchronized.

Summary of legacy implementations

- `Vector implements List, RandomAccess` - Synchronized resizable-array implementation of the `List` interface with additional "legacy methods."

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Unlike the new collection implementations, Vector is synchronized

- Hashtable implements Map - Synchronized hash table implementation of the Map interface that DOES NOT allow null keys or values, with additional "legacy methods."

  To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

  Unlike the new collection implementations, Hashtable is synchronized

Ordering

- Comparable - This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

  Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

```
package java.lang;

public interface Comparable<T> {

  /**
   * Compares this object with the specified object for order.  Returns a
   * negative integer, zero, or a positive integer as this object is less
   * than, equal to, or greater than the specified object.
   */
  public int compareTo(T o);
}
```

- Comparator - Represents an order relation, which may be used to sort a list or maintain order in a sorted set or map. Can override a type's natural ordering, or order objects of a type that does not implement the Comparable interface.

  A comparison function, which imposes a total ordering on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as TreeSet or TreeMap).

```
package java.util;

public interface Comparator<T> {
  /**
   * Compares its two arguments for order.  Returns a negative integer,
   * zero, or a positive integer as the first argument is less than, equal
   * to, or greater than the second.
   */
  int compare(T o1, T o2);

  boolean equals(Object obj);
}
```

6.2 Distinguish between correct and incorrect overrides of corresponding hashCode and equals methods, and explain the difference between == and the equals method.

The difference between equals and ==

The equals method can be considered to perform a deep comparison of the value of an object, whereas the == operator performs a shallow comparison. The equals method compares what an object points to rather than the pointer itself (if we can admit that Java has pointers). This indirection may appear clear to C++ programmers but there is no direct comparison in Visual Basic.

Using the equals method with String

The equals method returns a boolean primitive. This means it can be used to drive an if, while or other looping statement. It can be used where you would use the == operator with a primitive. The operation of the equal method and == operator has some strange side effects when used to compare Strings. This is one occasion when the immutable nature of Strings, and the way they are handled by Java, can be confusing.

There are two ways of creating a String in Java. The one way does not use the new operator. Thus normally a String is created

```
String s = new String("Hello");
```

but a slightly shorter method can be used

```
String s= "GoodBye";
```

Generally there is little difference between these two ways of creating strings, but the Exam may well ask questions that require you to know the difference.

The creation of two strings with the same sequence of letters without the use of the new keyword will create pointers to the same String in the Java String pool. The String pool is a way Java conserves resources. To illustrate the effect of this

```
String s = "Hello";
String s2 = "Hello";
if (s==s2){
    System.out.println("Equal without new operator");
}
```

```
String t = new String("Hello");
string u = new String("Hello");
if (t==u){
    System.out.println("Equal with new operator");
}
```

From the previous objective you might expect that the first output "Equal without new operator" would never be seen as s and s2 are different objects, and the == operator tests what an object points to, not its value. However because of the way Java conserves resources by re-using identical strings that are created without the new operator s and s2 have the same "address" and the code does output the string

```
"Equal without new operator"
```

However with the second set of strings t and u, the new operator forces Java to create separate strings. Because the == operator only compares the address of the object, not the value, t and u have different addresses and thus the string "Equal with new operator" is never seen.

```
The equals method applied to a String, however that String was
created, performs a character by character comparison.
```

The business of the use of the String pool and the difference between the use of == and the equals method is not obvious, particularly if you have a Visual Basic background. The best way to understand it is to create some examples for yourself to see how it works. Try it with various permutations of identical strings created with and without the new operator.

Using the equals method with Boolean

The requirement to understand the use of the equals operator on java.lang.Boolean is a potential gotcha. Boolean is a wrapper object for the boolean primitive. It is an object and using equals on it will test

According to the JDK documentation the equals method of the Boolean wrapper class

"Returns true if and only if the argument is not null and is a Boolean object that contains the same boolean value as this object".

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
if(b1.equals(b2)){
    System.out.println("We are equal");
}
```

As a slight aside on the subject of boolean and Boolean, once you are familiar with the if operator in Java you will know you cannot perform the sort of implicit conversion to a boolean beloved of bearded C/C++ programmers.

```
int x =1;
if(x){
    //do something, but not in Java
}
```

This will not work in Java because the parameter for the if operator must be a boolean evaluation, and Java does not have the C/C++ concept whereby any non null value is considered to be true. However you may come across the following in Java

```
boolean b1=true;

if(b1){
    //do something in java
}
```

Although this is rather bad programming practice it is syntactically correct, as the parameter for the if operation is a boolean.

Using the equals method with Object

Due to the fundamental design of Java an instance of any class is also an instance of java.lang.Object. Testing with equals performs a test on the Object as a result of the return value of the toString() method. For an Object the toString method simply returns the memory address. Thus the result is the equivalent of performing a test using the == operator. As Java is not designed to manipulate memory addresses or pointers this is not a particularly useful test.

Take the following example

```
public class MyParm{
public static void main(String argv[]){
        Object m1 = new Object();
        Object m2 = new Object();
        System.out.println(m1);
        System.out.println(m2);
        if (m1.equals(m2)){
                System.out.println("Equals");
                }else{
                System.out.println("Not Equals");
                }
        }
}
```

If you attempt to compile and run this code you will get an output of

```
java.lang.Object@16c80b
java.lang.Object@16c80a
Not Equals
```

Those weird values are memory addresses, and probably not what you want at all.

Using the hashcode method

The hashcode method is inherited from the great grandparent of all classes Object, thus an instance of any object can make a call to hashcode. The signature of hashcode is

```
public int hashCode()
```

Thus you might get an exam question that offers you some bogus signatures for hashcode that return types other than int or take some parameters instead of none. However I suspect the questions tend to be slightly more theoretical than this.

The int value that is returned is of particular use with the hash based Collection classes, ie

HashTable, HashSet, HashSet

The nature of hash based collections is to store keys and values. The key is what you use to look up a value. Thus you could for instance use a HashMap to store employee id in the key value and the employee name in the value part.

Generally a hashcode value will be the memory address of the object. You can demonstrate this to yourself quite easily with some trivial code such as.

```
public class ShowHash{
    public static void main(String argv[]){
        ShowHash sh = new ShowHash();
        System.out.println(sh.hashCode());
    }
}
```

When I compiled and ran this code I got an output of

```
7474923
```

Which is a representation of the memory address of that class on that run of the program. This illustrates one of the features of a hashcode is that it does not have to be the same value from one run of a program to another. If you think about the memory address of an object there is not guarantee at all what it will be from one run to another of a program.

Here is a quote from the JDK1.4 docs that covers this part of the requirements for a hashcode value.

"Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application."

Note how it says that the value returned from hashCode must be the same in the same program run provided no information used in the equals method comparisons on the object is modified. This brings us to the relationship between the equals and the hashCode method.

equals and hashCode

Every object has access to an equals method because it is inherited from the great grandparent class called Object. However this default object does not always do anything useful as by default it simply compares the memory address of the object. The downside of this can be seen dramatically when used with the String classes. If the String class did not implement its own version of the equals method comparing two Strings would compare the memory address rather than the character sequence. This is rarely what you would want, and for this reason the String class implements it's own version of the equals method that makes a character by character comparison.

Here is another of the points from the API documentation

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

This principle is illustrated with the following code,

```
public class CompStrings{
    public static void main(String argv[]){
    String s1 = new String("Hello");
    String s2 = new String("Hello");
    System.out.println(s1.hashCode());
    System.out.println(s2.hashCode());
    Integer i1 = new Integer(10);
    Integer i2 = new Integer(10);
    System.out.println(i1.hashCode());
    System.out.println(i2.hashCode());
    }

}
```

This code will print out the same hashCode value for s1 and s2 and i1 and i2 on each run of the program. In theory it could print out different values under different circumstances.

```
Objects that are equal according to the equals method must return
the same hashCode value
```

When two objects are not equal

It would be a plausible extrapolation from what I have covered so far to believe that two objects that are not equal according to the equals() method would have to return different hashCode values. This is not so, as stated in the API docs.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing

```
If two objects are not equal according to equals, they are not
required to return different hashCode values.
```

It is worthwhile looking up the original API docs to understand the requirements of the hashCode method.

6.3 Write code that uses the generic versions of the Collections API, in particular, the Set<E>, List<E>, Queue<E> and Map <K,V> interfaces and implementation classes. Recognize the limitations of the non-generic Collections API and how to refactor code to use the generic versions.

When you take an element out of a `Collection`, you must cast it to the type of element that is stored in the collection. Besides being inconvenient, this is unsafe. The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.

Generics provide a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and can insert the correct casts on values being taken out of the collection.

Without generics:

```
static void expurgate(Collection c) {
        for (Iterator i = c.iterator(); i.hasNext(); ) {
                if (((String) i.next()).length() == 4) {
                        i.remove();
                }
        }
}
```

Here is the same example modified to use generics:

```
static void expurgate(Collection<String> c) {
        for (Iterator<String> i = c.iterator(); i.hasNext(); ) {
                if (i.next().length() == 4) {
                        i.remove();
                }
        }
}
```

When you see the code `<Type>`, read it as "of Type"; the declaration above reads as "Collection of String c". The code using generics is clearer and safer. We have eliminated an unsafe cast and a number of extra parentheses. The compiler can verify at compile time that the type constraints are not violated at run time. Because the program compiles without warnings, we can state with certainty that it will not throw a `ClassCastException` at run time. The net effect of using generics, especially in large programs, is improved readability and robustness.

public interface Queue<E> extends Collection<E>

A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

The `offer` method inserts an element if possible, otherwise returning `false`. This differs from the `Collection.add` method, which can fail to add an element only by throwing an unchecked exception. The `offer` method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The `remove()` and `poll()` methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The `remove()` and `poll()` methods differ only in their behavior when the queue is empty: the `remove()` method throws an exception, while the `poll()` method returns `null`.

The `element()` and `peek()` methods return, but do not remove, the head of the queue.

The `Queue` interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the `BlockingQueue` interface, which extends this interface.

`Queue` implementations generally do not allow insertion of `null` elements, although some implementations, such as `LinkedList`, do not prohibit insertion of `null`. Even in the implementations that permit it, `null` should not be inserted into a `Queue`, as `null` is also used as a special return value by the `poll` method to indicate that the queue contains no elements.

`Queue` implementations generally do not define element-based versions of methods `equals` and `hashCode` but instead inherit the identity based versions from class `Object`, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

```
public boolean offer(E element)

public E remove()       // removes !
public E poll()         // removes !

public E element()      // DOES NOT remove !
public E peek()         // DOES NOT remove !
```

PriorityQueue<E>

An unbounded priority queue based on a priority heap. This queue orders elements according to an order specified at construction time, which is specified either according to their natural order (see `Comparable`), or according to a `Comparator`, depending on which constructor is used. A priority queue DOES NOT permit `null` elements. A priority queue relying on natural ordering also DOES NOT permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

NOTE, `java.lang.String` implements `Comparable`.

The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements - ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

Example:

```
PriorityQueue queue = new PriorityQueue();
queue.offer("CCC-1");
queue.offer("BBB");
queue.offer("AAA");
queue.offer("CCC-2");

out.println("1. " + queue.poll()); // removes
out.println("2. " + queue.poll()); // removes
out.println("3. " + queue.peek());
out.println("4. " + queue.peek());
out.println("5. " + queue.remove()); // removes
out.println("6. " + queue.remove()); // removes
out.println("7. " + queue.peek());
out.println("8. " + queue.element()); // Throws NoSuchElementException !
```

The output will be:

```
1. AAA
2. BBB
3. CCC-1
4. CCC-1
5. CCC-1
6. CCC-2
7. null
Exception in thread "main" java.util.NoSuchElementException
        at java.util.AbstractQueue.element(Unknown Source)
        at regex.Replacement.main(Replacement.java:28)
```

Non-generic collections

Example of using raw (non-parameterized) collections with generics (parameterized) collections:

```
ArrayList list = new ArrayList(); // OK
ArrayList<String> listStr = list; // WARNING, but OK
ArrayList<StringBuffer> listBuf = list; // WARNING, but OK
listStr.add(0, "Hello"); // OK
StringBuffer buff = listBuf.get(0); // Runtime Exception !

Exception in thread "main" java.lang.ClassCastException: java.lang.String
        at Client.main(Client1.java:28)
```

6.4 Develop code that makes proper use of type parameters in class/interface declarations, instance variables, method arguments, and return types; and write generic methods or methods that make use of wildcard types and understand the similarities and differences between these two approaches.

Parameters and arguments

In Java 5.0 code, generics will manifest itself in two forms, as *type parameters* and as *type arguments*. You will be familiar with the distinction between parameters and arguments in methods:

```
void foo(int aaa, int bbb) {
        ...
}

void bar(int ccc) {
        foo(ccc, 142);
}
```

Above, `aaa` and `bbb` are the parameters to `foo()`. When `foo()` is called, `ccc` and `142` are passed as arguments. Parameters are the "generic" part, and arguments are the "specific" part.

Also note that `ccc` is both a parameter of `bar(...)` and an argument to `foo(...)` (this will also happen in generics with type parameters and arguments).

Reading generics is a matter of working out where a type parameter is being declared, and where a type argument is being passed.

Type parameters

Type parameters can appear in two locations, class (or interface) declarations, and method declarations. When type parameters are used, we are saying that this class/interface/method body is parameterized over those types. We can use those type parameters in the body as if they were a real classname we had imported, but we don't care what actual class it is.

A generic class/interface is declared by putting type parameters **after the name of the class/interface**. Type parameters begin and end with angle brackets and are separated by commas. You can specify more than one type parameter, and each type parameter can have a bound (constraint):

```
public class Foo <TypeParam1, TypeParam2> {
        ...
        // type parameters used here
        ...
}
```

or

```
public interface I<T> {
        public T getData();
        public void releaseData(T data);
}
```

A type bound places a constraint on the type arguments that can be passed to the type parameter.

No bound, any type argument can be used:

```
<T>
```

`T` can be any subclass of `InputStream`:

```
<T extends InputStream>
```

`T` must implement the `Serializable` interface (NOTE, that `extends` is used here, even when talking about implementing interfaces):

```
<T extends Serializable>
```

`T` must be a subclass of `InputStream` and implement `Serializable`:

```
<T extends InputStream & Serializable>
```

`T` must implement three interfaces:

```
<T extends Serializable & Runnable & Cloneable>
```

Two type parameters (such as the type of the keys, and type of the values, in a `Map`):

```
<K, V>
```

Two type parameters; the second bound is defined in terms of the first. Type bounds can be mutually- or even self-recursive:

```
<T, C extends Comparator<T>>
```

Lets define our own immutable "pair" class. Notice how we have declared fields and methods in terms of the type parameters. `getFirst()` is a method in a generic class and uses one of the generic type parameters, but that is different to a generic method:

```
public class Pair <X, Y> {
        private final X a;
        private final Y b;

        public Pair(X a, Y b) {
                this.a = a;
                this.b = b;
        }

        public X getFirst() {
                return a;
        }
        public Y getSecond() {
                return b;
        }
}
```

Methods can have their own type parameters, independent of the type parameters of the enclosing class (or even if the enclosing class is not generic). **The type parameters go just before the return type of the method declaration**:

```
class PairUtil {

        public static <A extends Number, B extends Number> double
```

```
                                                   add(Pair<A, B> p) {
                return p.getFirst().doubleValue() + p.getSecond().doubleValue();
        }

        public static <A, B> Pair<B, A> swap(Pair<A, B> p) {
                A first = p.getFirst();
                B second = p.getSecond();
                return new Pair<B, A>(second, first);
        }
}
```

We have done a few things in the `add(...)` method:

- The method is generic over two type parameters `A` and `B`.
- We don't care what `A` and `B` are, so long as they are subclasses of `Number`.
- The argument to the method is a `Pair`; the type arguments to that `Pair` happen to include the type parameters to `add(...)`.

The second method, `swap(...)`, is even more interesting:

- The type parameters are used to define the return type, as well as the argument.
- Local variables in the method are declared in terms of the type parameters.
- The type parameters `A` and `B` are used as type arguments in the constructor call.

Type arguments

Type parameters are for defining generic classes; type arguments are for using generic classes. And you will be using generic classes far more often than you write them.

Wherever you use a generic classname, you need to supply the appropriate type arguments. These arguments go **straight after the classname**, surrounded by angle brackets. The arguments you supply must satisfy any type bounds on the type parameters. There are 5 main contexts where you can use type arguments, shown here.

1. Variable declaration.

   When using a generic class for a variable (local, field or method parameter), you need to supply the type arguments:

   ```
   Pair<String, Integer> p1;
   Pair<Integer, String> p2;
   Pair<String, Integer> p3;
   ```

2. Constructor call.

   When calling the constructor of a generic class, the type arguments **must also follow the classname** (that is, the constructor name).

   ```
   String s = "foo";
   Integer i = new Integer(3);
   p1 = new Pair<String, Integer>(s, i);
   ```

3. Inferred generic-method call.

   When calling a generic method, the method's arguments may contain enough information for the compiler to infer the type arguments to the method call.

```
...
public static <A, B> Pair<B, A> swap(Pair<A, B> p) {
...

p2 = PairUtil.swap(p1);
```

4. Explicit generic-method call.

   When calling a generic method where you do need to supply the type arguments, the **arguments go after the dot of the method call**. This applies to static or non-static method calls.

```
...
public static <A, B> Pair<B, A> swap(Pair<A, B> p) {
...

p2 = PairUtil.<String,Integer>swap(p1);
```

5. Casting.

   You can cast to a generic class, and you can supply the type arguments. But you get a compiler warning for that:

```
Object o = p1;
p3 = (Pair<String, Integer>) o; // WARNING !!!
```

Declaring instance variables

```
public class Basket<E> {
        ...
}

class Fruit {
}

class Apple extends Fruit {
}

class Orange extends Fruit {
}

Basket b = new Basket(); // OK !
Basket b1 = new Basket<Fruit>();  // OK !
Basket<Fruit> b2 = new Basket<Fruit>(); // OK !

// Type mismatch: cannot convert from Basket<Fruit> to Basket<Apple>
Basket<Apple> b3 = new Basket<Fruit>(); // WRONG !!!

// Type mismatch: cannot convert from Basket<Apple> to Basket<Fruit>
Basket<Fruit> b4 = new Basket<Apple>();  // WRONG !!!

Basket<?> b5 = new Basket<Apple>(); // OK !

// 1. Cannot instantiate the type Basket<?>
// 2. Type mismatch: cannot convert from Basket<?> to Basket<Apple>
Basket<Apple> b6 = new Basket<?>(); // WRONG !!!
```

Implementing generic types

In addition to using generic types, you can implement your own. A generic type has one or more type parameters. Here is an example with only one type parameter called E. A parameterized type must be a reference type, and therefore primitive types are NOT allowed to be parameterized types.

```
interface List<E> {
        void add(E x);
        Iterator<E> iterator();
}

interface Iterator<E> {
        E next();
        boolean hasNext();
}

class LinkedList<E> implements List<E> {
        // implementation
}
```

Here, E represents the type of elements contained in the collection. Think of E as a placeholder that will be replaced by a concrete type. For example, if you write LinkedList<String> then E will be replaced by String.

In some of your code you may need to invoke methods of the element type, such as Object's hashCode() and equals(). Here is an example that takes two type parameters:

```
class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V> {
        ...
        public V get(Object k) {
                ...
                int hash = k.hashCode();
                ...
        }
}
```

The important thing to note is that you are required to replace the type variables K and V by concrete types that are subtypes of Object.

Generic methods

Genericity is not limited to classes and interfaces, you can define generic methods. Static methods, nonstatic methods, and constructors can all be parameterized in almost the same way as for classes and interfaces, but the syntax is a bit different. Generic methods are also invoked in the same way as non-generic methods.

Before we see an example of a generics method, consider the following segment of code that prints out all the elements in a collection:

```
public void printCollection(Collection c) {
        Iterator i = c.iterator();
        for(int k = 0; k < c.size() ; k++) {
                out.printn(i.next());
        }
}
```

Using generics, this can be re-written as follows. Note that the Collection<?> is the collection of an unknown type:

```
void printCollection(Collection<?> c) {
        for(Object o:c) {
                out.println(e);
        }
}
```

This example uses a feature of generics known as *wildcards*.

Some more generics examples:

```
public class Basket<E> {
        private E element;

        public void setElement(E x) {
                element = x;
        }

        public E getElement() {
                return element;
        }
}

class Fruit {
}

class Apple extends Fruit {
}

class Orange extends Fruit {
}
```

A client code (compilation problem):

```
Basket<Fruit> basket = new Basket<Fruit>();
basket.setElement(new Apple());       // OK, can assign Apple reference to
                                      // Fruit variable

// Type mismatch: cannot convert from Fruit to Apple !!!

Apple apple = basket.getElement();    // WRONG ! Compilation error !

Apple apple = (Apple) basket.getElement(); // OK ! Compiles and runs fine.
```

A client code (runtime exception):

```
Basket<Fruit> basket = new Basket<Fruit>();
basket.setElement(new Apple());
Orange orange = (Orange) basket.getElement(); // Runtime exception !!!


Exception in thread "main" java.lang.ClassCastException: Apple
        at Client.main(Client1.java:8)
```

Wildcards

There are three types of wildcards:

1. "`? extends Type`": Denotes a family of subtypes of type `Type`. This is the most useful wildcard.
2. "`? super Type`": Denotes a family of supertypes of type `Type`.

3. "?": Denotes the set of all types or ANY.

As an example of using wildcards, consider a `draw()` method that should be capable of drawing any shape such as circle, rectangle, and triangle. The implementation may look something like this. Here `Shape` is an abstract class with three subclasses: `Circle`, `Rectangle`, and `Triangle`:

```
public void draw(List<Shape> shape) {
        for(Shape s: shape) {
                s.draw(this);
        }
}
```

It is worth noting that the `draw(...)` method can only be called on lists of `Shape` and cannot be called on a list of `Circle`, `Rectangle`, and `Triangle` for example. In order to have the method accept any kind of shape, it should be written as follows:

```
public void draw(List<? extends Shape> shape) {
        for(Shape s: shape) {
                s.draw(this);
        }
}

public static <T extends Comparable<? super T>> void sort(List<T> list) {
        Object a[] = list.toArray();
        Arrays.sort(a);
        ListIterator<T> i = list.listIterator();
        for (int j = 0; j < a.length; j++) {
                ...
        }
}
```

Another example of unbounded wildcards:

```
Basket<?> basket = new Basket<Apple>();
basket.setElement(new Apple());       // WRONG !!!
Apple apple = (Apple) basket.getElement();
```

The compiler does not know the type of the element stored in `basket`. That is why it cannot guarantee an apple can be inserted into the basket `basket`. So the statement `basket.setElement(new Apple())` is not allowed. The methode `basket.setElement(...)` cannot be used at all (read only collection).

If we have some subclasses of the `Apple` class:

```
class GoldenDelicious extends Apple {}
class Jonagold extends Apple {}
```

And want to create a parameterized method which will accept basket of any apples, then we can create the following method with wildcards:

```
public static boolean isRipeInBasket(Basket<? extends Apple> basket) {
        Apple apple = basket.getElement();
        ...
}
```

or parameterized method:

```
public static <A extends Apple> boolean isRipeInBasket(Basket<A> basket) {
        Apple apple = basket.getElement();
        ...
}
```

If we want to add some sort of apples to basket, the following generic method is required:

```
public static <A extends Apple> void insert(A apple, Basket<? super A> basket) {
        basket.setElement(apple);
}
```

or

```
public static <A extends Apple> void insert(Apple apple, Basket<? super A>
basket) {
        basket.setElement(apple);
}
```

Wild-cards in type arguments

Java 5.0 allows the use of a wild-card in a type argument, in order to simplify the use of generics (easier to type, and to read).

For example, you may want to use a generic class, but you don't particularly care what the type argument is. What you could do is specify a "dummy" type parameter `T`, as in the generic method `count_0()` below:

```
public static <T> int count_0(List<T> list) {
    int count = 0;
    for (T n : list) {
        count++;
    }
    return count;
}
```

Alternatively, you can avoid creating a generic method, and just use a `?` wild-card as in `count_1()`. You should read `List<?>` as "a list of whatever":

```
public static int count_1(List<?> list) {
    int count = 0;
    for (Object n : list) {
        count++;
    }
    return count;
}
```

Wild-cards with upper and lower bounds

The next two methods use a bounded wild-card to express the required subclass/superclass relationship. You should read `List<? extends T>` as "A list of T, or any subclass of T". You can read `List<? super T>` as "A list of T, or any T's super-classes":

```
List<Number> listOfNumbers = new ArrayList<Number>();
listOfNumbers.add(new Integer(3));
listOfNumbers.add(new Double(4.0));

List<Integer> listOfIntegers = new ArrayList<Integer>();
```

```
listOfIntegers.add(new Integer(3));
listOfIntegers.add(new Integer(4));

addAll_1(listOfIntegers, listOfNumbers);
addAll_2(listOfIntegers, listOfNumbers);
...

/**
 * Append src to dest, so long as "whatever the types of things in src are",
 * they extend "the types of things in dest".
 */
public static <T> void addAll_1(List<? extends T> src, List<T> dest) {
        for (T o : src) {
                dest.add(o);
        }
}

/**
 * Append src to dest, so long as "whatever the types of things dest can hold",
 * they are a superclass of "the types of things in src".
 */
public static <T> void addAll_2(List<T> src, List<? super T> dest) {
        for (T o : src) {
                dest.add(o);
        }
}
```

Erasure

Generics are implemented by the Java compiler as a front-end conversion called erasure, which is the process of translating or rewriting code that uses generics into non-generic code (that is, maps the new syntax to the current JVM specification). In other words, this conversion erases all generic type information; **all information between angle brackets is erased**. For example, `LinkedList<Integer>` will become `LinkedList`. Uses of other type variables are replaced by the upper bound of the type variable (for example, `Object`), and when the resulting code is not type correct, a cast to the appropriate type is inserted.

Let's take a look at the following code:

```
public class Basket<E> {
        private E element;

        public void setElement(E x) {
                element = x;
        }

        public E getElement() {
                return element;
        }
}

class Fruit {
}

class Apple extends Fruit {
}

class Orange extends Fruit {
}
```

The following code will compile and run correctly:

```
Basket basket = new Basket<Orange>();
basket.setElement(new Apple());
Apple apple = (Apple) basket.getElement();
```

Beause we use a generic class without specifying the type of its element, the compiled code is not type safe and the compiler will issue a warning. During the runtime the JVM has no information about the types of the elements of the used `Basket` and so it cannot tell a difference between a `Basket<Orange>` and a `Basket<Apple>`. So we are indeed allowed to insert an apple where only oranges should be allowed (we have been warned). Since there is an `Apple` in the `basket`, no exception will be thrown in `Apple apple = (Apple)basket.getElement()`.

The following example gives compile time error:

```
public <A extends Fruit> void erasureTest(A a) {}
public <B extends Fruit> void erasureTest(B b) {} // WRONG !!!
```

Both of them will look like this at runtime:

```
public void erasureTest(Fruit a) {}
public void erasureTest(Fruit b) {} // WRONG !!!
```

The following methods also will cause a compile time error:

```
public static void erasureTest2(Basket<? extends Apple> basket) {}
public static void erasureTest2(Basket<? extends Orange> basket) {} // WRONG !!!
```

At the runtime signatures of these 2 methods will be:

```
public static void erasureTest2(Basket basket) {}
public static void erasureTest2(Basket basket) {} // WRONG !!!
```

The following code is a legal example of overloaded methods:

```
public static <A extends Apple> void erasureTest2(A a,
                                          Basket<? super A> b){} // OK
public static <G extends Orange> void erasureTest2(G g,
                                          Basket<? super G> b){} // OK
```

The compiler converts the signatures of the insertRipe methods to the following ones:

```
public static void erasureTest2(Apple a,  Basket b)
public static void erasureTest2(Orange g, Basket b)
```

Those signatures are different and so the method name can be overloaded.

The following code will compile (with warnings):

```
Basket<Orange> bO = new Basket<Orange>();
Basket b = bO;
Basket<Apple> bA = (Basket<Apple>) b;         // WARNING
```

Because at runtime the last line will be:

```
Basket bA = (Basket) b;
```

The using of `instanceof` is ILLEGAL with parameterized types:

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { // WRONG !!! Compilation error !
        ...
}
```

Arrays and generics

The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type. You can declare array types whose element type is a type variable or a parameterized type, but not array objects.

```
// Cannot create a generic array of Basket<Apple>
Basket<Apple>[] b  = new Basket<Apple>[10];   // WRONG !!!!

// Cannot create a generic array of Basket<Apple>
Basket<?>[] b1 = new Basket<Apple>[10];       // WRONG !!!

Basket<?>[] b2 = new Basket<?>[10];           // OK !

public <T> T[] test() {                       // OK !
        return null;
}

// Cannot create a generic array of T
public <T> T[] test1() {                      // WRONG !!!
        return new T[10];
}
```

6.5 Use capabilities in the java.util package to write code to manipulate a list by sorting, performing a binary search, or converting the list to an array. Use capabilities in the java.util package to write code to manipulate an array by sorting, performing a binary search, or converting the array to a list. Use the java.util.Comparator and java.lang.Comparable interfaces to affect the sorting of lists and arrays. Furthermore, recognize the effect of the "natural ordering" of primitive wrapper classes and java.lang.String on sorting.

Interface Collection<E>

The following method from `Collection` interface returns an array containing all of the elements in this collection. If the collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The returned array will be "safe" in that no references to it are maintained by this collection. (In other words, this method must allocate a new array even if this collection is backed by an array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs:

```
Object[] toArray()
```

The following method from `Collection` interface returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection:

```
<T> T[] toArray(T[] a)
```

If this collection fits in the specified array with room to spare (i.e., the array has more elements than this collection), the element in the array immediately following the end of the collection is set to `null`. This is useful in determining the length of this collection only if the caller knows that this collection does not contain any null elements.)

If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

Like the `toArray` method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose `l` is a `List` known to contain only strings. The following code can be used to dump the list into a newly allocated array of `String`:

```
String[] x = (String[]) v.toArray(new String[0]);
```

Note that `toArray(new Object[0])` is identical in function to `toArray()`.

java.util.Arrays.sort()

The `sort` methods for the primitive types look like:

```
void sort (type[] array)
void sort (type[] array, int fromIndex, int toIndex)
```

There is a `sort` method for each primitive type except `boolean`. The elements in the array are sorted in ascending order.

The second version sorts those elements between the given indices.

For sorting `Object` types, there are two approaches. For the first approach there are two methods available:

```
void sort (Object[] array)
void sort (Object[] array, int fromIndex, int toIndex)
```

Here the objects are sorted into ascending order using the "natural ordering" of the elements. The array elements MUST implement the `java.lang.Comparable` interface and provide the method:

```
int compareTo (Object)
```

NOTE, `String` class implements `Comparable` interface.

This method defines the "natural ordering" such that comparing object `x` to `y` results in a negative number if `x` is lower than `y` (on a scale that makes sense for the class), equal to `0` if the objects are identical, and a positive number if `x` is greater than `y`.

Your `compareTo()` method must be written to return a negative, zero, or positive integer according to the ordering rules that make sense for the nature of the objects that the class describes.

For the ALTERNATIVE approach to comparing `Object` arrays, you create an auxiliary class that implements the `java.util.Comparator` interface and that knows how to compare and order two objects of interest. This technique is particularly useful if you need to sort classes that you CANNOT RE-WRITE to implement the `Comparable` interface. The `Comparator` class has two methods:

```
int compare (Object obj1, Object obj2)
boolean equals (Object obj1, Object obj2)
```

The `compare()` method should return a negative, zero, or positive integer, according to the ordering rules that you implement, if `obj1` is less than, equal to, or greater than `obj2`. Similarly, the `equals()` method compares the objects for equality according to the rules you implement. When using the `Comparator` technique, the two object array sorting methods are:

```
void sort (Object[] array, Comparator comp)
void sort (Object[] array, int fromIndex, int toIndex, Comparator comp)
```

java.util.Arrays.binarySearch()

Another useful set of overloaded methods in the `Arrays` class is the set of binary searching methods:

```
int binarySearch (type[] array, type key)
int binarySearch (Object[] array, Object key, Comparator comp)
```

There is an overloaded `binarySearch()` method for each primitive type except `boolean` and one more for `Object`. These methods search an array for the given `key` value. Returns index of the search key, if it is contained in the list. The array to be searched *MUST* be sorted first in ascending order, perhaps by using one of the `sort()` methods. If it is not sorted, the results are UNDEFINED. If the array contains multiple elements with the specified value, there is NO guarantee which one will be found.

The methods return the index of the search `key`, if it is contained in the list; otherwise, `(-(insertion_point) - 1)`. The `insertion_point` is defined as the point at which the `key` would be inserted into the array: the index of the first element greater than the `key`, or `array.length`, if all elements in the array are less than the specified `key`.

NOTE, this guarantees that the return value will be `>= 0` if and only if the `key` is found.

For the case of `Object` arrays and the first type of `binarySearch()` method above, the elements must implement the `Comparable` interface. If you cannot change the classes to implement this interface, then you can provide a `Comparator` and use the second type of `binarySearch()` method shown above.

The class `Arrays`, provides useful algorithms that operate on arrays. It also provides the static `asList()` method, which can be used to create `List` views of arrays. Changes to the `List` view affects the array and vice versa. The `List` size is the array size and CANNOT be modified. The `asList()` method in the `Arrays` class and the `toArray()` method in the `Collection` interface provide the bridge between arrays and collections:

```
Set mySet = new HashSet(Arrays.asList(myArray));

String[] strArray = (String[]) mySet.toArray();
```

The `asList(...)` method of `java.util.Arrays` class returns a fixed-size list backed by the specified array. Changes to the returned list "write through" to the array. This method acts as bridge between array-based and collection-based APIs, in combination with `Collection.toArray`. The returned list is serializable and implements `RandomAccess`.

This method also provides a convenient way to create a fixed-size list initialized to contain several elements:

```
List stooges = Arrays.asList("Larry", "Moe", "Curly");

public static <T> List<T> asList(T... a)
```

`java.util.Comparator` interface:

```
package java.util;

public interface Comparator<T> {
        int compare(T o1, T o2);
        boolean equals(Object obj);
}
```

`java.lang.Comparable` interface:

```
package java.lang;

public interface Comparable<T> {
        public int compareTo(T o);
}
```

java.util.Collections

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The following method sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list MUST implement the `Comparable` interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

The specified list must be modifiable, but need not be resizable.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

The following method sorts the specified list according to the order induced by the specified comparator. All elements in the list MUST implement the `Comparable` interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

The specified list must be modifiable, but need not be resizable.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array:

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

With the help of the method `sort(list, comparator)` you can for example sort the list in reverse order:

```
/** Non-Comparable class */
public class WeirdString {
        private String str;

        WeirdString(String str ){
                this.str = str;
        }
}

import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class ReverseClient {

        public static void main(String ... sss) {
                List myList = new LinkedList();
                String s1 = new String("2345");
                String s2 = new String("12345");
                String s3 = new String("45");
                String s4 = new String("345");
                WeirdString s5 = new WeirdString("345");

                myList.add(s1);
                myList.add(s2);
                myList.add(s3);
                myList.add(s4);
                // myList.add(s5); // WRONG ! See *)

                System.out.println("Original list : " + myList);
                Comparator cmp = Collections.reverseOrder();
                // *) The 'sort(list, comp)' method may throw ClassCastException
                // if the list contains elements that are not *mutually*
                // comparable using the specified comparator
                Collections.sort(myList, cmp);
                System.out.println("Sorted list (descending order) : " + myList);
        }
}
```

The output of the `ReverseClient` class will be:

```
Original list : [2345, 12345, 45, 345]
Sorted list (descending order) : [45, 345, 2345, 12345]
```

The following method searches the specified list for the specified object using the binary search algorithm. The list MUST be sorted into ascending order according to the natural ordering of its elements (as by the `sort(List)` method) prior to making this call. If it is not sorted, the results are UNDEFINED. If the list contains multiple elements equal to the specified object, there is NO guarantee which one will be found.

Returns index of the search `key`, if it is contained in the list; otherwise, (`-(insertion_point) - 1`). The `insertion_point` is defined as the point at which the `key` would be inserted into the list: the index of the first element greater than the `key`, or `list.size()`, if all elements in the list are less than the specified key. NOTE, that this guarantees that the return value will be `>= 0` if and only if the `key` is found:

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list,
                                                                      T key)
```

This code sample demonstrates `sort(...)` and `binarySearch(...)` methods:

```
/** Non-Comparable class */
public class WeirdString {
        private String str;

        WeirdString(String str ){
                this.str = str;
        }
}

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class CollClient {

        public static void main(String ... sss) {
                List myList = new LinkedList();
                String s1 = new String("2345");
                String s2 = new String("12345");
                String s3 = new String("45");
                String s4 = new String("345");
                WeirdString s5 = new WeirdString("345");

                myList.add(s1);
                myList.add(s2);
                myList.add(s3);
                myList.add(s4);
                // myList.add(s5); // WRONG ! See *)
                System.out.println("Original list : " + myList);
                // Result may be invalid if collection unsorted in *ascending*
                // order!!!
                // *) May throw ClassCastException if the list contains
                // elements that are not *mutually* comparable !!!
                System.out.println("45 : " +
                                       Collections.binarySearch(myList, "45"));
                System.out.println("99 : " +
                                       Collections.binarySearch(myList, "99"));
                // *) May throw ClassCastException if the list contains
                //    elements that are not *mutually* comparable !!!
                Collections.sort(myList); // sort in ascending order
                System.out.println("Sorted list (ascending order): " + myList);
                System.out.println("45 : " +
                                       Collections.binarySearch(myList, "45"));
                System.out.println("99 : " +
                                       Collections.binarySearch(myList, "99"));
        }
}
```

The output:

```
Original list : [2345, 12345, 45, 345]
45 : 2
99 : -5
Sorted list (ascending order): [12345, 2345, 345, 45]
45 : 3
99 : -5
```

The following method searches the specified list for the specified object using the binary search algorithm. The list MUST be sorted into ascending order according to the specified comparator (as by the `sort(List, Comparator)` method,), prior to making this call. If it is not sorted, the results are UNDEFINED. If the list contains multiple elements equal to the specified object, there is NO guarantee which one will be found.

`c` - the comparator by which the list is ordered. A `null` value indicates that the elements' natural ordering should be used.

The method returns index of the search `key`, if it is contained in the list; otherwise, `(-(insertion_point) - 1)`. The insertion point is defined as the point at which the `key` would be inserted into the list: the index of the first element greater than the `key`, or `list.size()`, if all elements in the list are less than the specified `key`. NOTE, this guarantees that the return value will be >= 0 if and only if the `key` is found.

```
public static <T> int binarySearch(List<? extends T> list, T key,
                                            Comparator<? super T> c)
```

The following code sample shows how to reverse the list of objects:

```
/** Non-Comparable class */
public class WeirdString {
        private String str;

        WeirdString(String str ){
                this.str = str;
        }
}

import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class ReverseListClient {

        public static void main(String ... sss) {
                List myList = new LinkedList();
                String s1 = new String("2345");
                String s2 = new String("12345");
                String s3 = new String("45");
                String s4 = new String("345");
                WeirdString s5 = new WeirdString("345");

                myList.add(s1);
                myList.add(s2);
                myList.add(s3);
                myList.add(s4);
                myList.add(s5);        // OK, no need to be mutually comparable

                System.out.println("Original list : " + myList);
                Collections.reverse(myList);   // OK
                System.out.println("Reversed list : " + myList);
        }
}
```

The code sample's output will be (reversed list):

```
Original list : [2345, 12345, 45, 345, WeirdString@1f6a7b9]
Reversed list : [WeirdString@1f6a7b9, 345, 45, 12345, 2345]
```