```java
import java.util.*;
public class Bucky {
    public static void main(String[] args) {

        Integer[] iray = {1,2,3,4,5};
        Character[] cray = {'b','u','c','k','y'};

        printMe(iray);
        printMe(cray);
    }

    //generic emthod
    public static <T> void printMe(T[] x){
        for(T b : x)
            System.out.printf("%s ", b);
        System.out.println();
    }
}
```

How to return generic data

```java
public class Bucky {
    public static void main(String[] args) {

        System.out.println(max(23,42,1));
        System.out.println(max("apples","tots","chicken"));

    }

public static <T extends Comparable<T>> T max(T a, T b, T c){
    T m = a;

    if(b.compareTo(a) > 0)
        m = b;

    if(c.compareTo(m) > 0)
        m = c;

    return m;
}
```

```
<T extends Comparable<T>> :
        it means only object which extends comparable class
can be used in this method.

compareTo() method returns int for any comparison so we can
use it for generic data taype comparison
```

```java
public class Bucky {
    public static void main(String[] args) {

        String[] things = {"apple", "bob", "ham", "bob", "bacon"};
        List<String> list = Arrays.asList(things);

        System.out.printf("%s ", list);
        System.out.println();

        Set<String> set = new HashSet<String>(list);
        System.out.printf("%s ", set);

    }
```

```java
PriorityQueue<String> q = new PriorityQueue<String>();

q.offer("first");
q.offer("secoind");
q.offer("third");

System.out.printf("%s ", q);
System.out.println();

System.out.printf("%s ", q.peek());
System.out.println();

q.poll();
System.out.printf("%s ", q);
```

```java
for(String x : list2)
    System.out.printf("%s ", x);

Collections.addAll(list2, stuff);

System.out.println();
for(String x : list2)
    System.out.printf("%s ", x);
System.out.println();

System.out.println(Collections.frequency(list2, "digg"));
```

```
youtube google digg
youtube google digg apples beef corn ham
1
```

```java
boolean tof = Collections.disjoint(list1, list2);
```

returns true if thsy don't have anything in common..

```
youtube google digg
youtube google digg apples beef corn ham
1
false
```

```
  //create an array and convert to lsit
  Character[] ray ={'p','w','n'};
  List<Character> l = Arrays.asList(ray);
  System.out.println("List is : ");
  output(l);

//reverse and print out the lsit
Collections.reverse(l);
System.out.println("After reverse : ");
output(l);

//createa new array and a new lsit
Character[] newRay = new Character[3];
List<Character> listCopy = Arrays.asList(newRay);

//copy contents of list into lsitcopy
Collections.copy(listCopy, l);

  //fill collection with crap
  Collection.fill(l, 'X');
  System.out.println("After filling the lsit : ");
  output(l);
```

```
List is :
p w n
After reverse :
n w p
Copy of list :
n w p
After filling the lsit :
X X X
```

```
String[] crap = {"apples", "lemons", "geese", "bacon", "youtube"};
List<String> l1 = Arrays.asList(crap);

Collections.sort(l1);
System.out.printf("%s\n", l1);

Collections.sort(l1, Collections.reverseOrder());
```

```
[apples, bacon, geese, lemons, youtube]
[youtube, lemons, geese, bacon, apples]
```

```
String[] stuff = {"babies", "watermelong", "melons", "fudge"};
LinkedList<String> thelist = new LinkedList<String>(Arrays.asList(

thelist.add("pumpikinf");
thelist.addFirst("firstthing");

//cponmvert back to an array
stuff = thelist.toArray(new String[thelist.size()]);

for(String x : stuff)
    System.out.printf("%s ", x);
```

```
ing babies watermelong melons fudge pumpikinf
```

```
String[] things ={"eggs", "lasers", "hats", "pie"};
List<String> list1 = new ArrayList<String>();

//add array items to list
for(String x: things)
    list1.add(x);

String[] morethings = {"lasers", "hats"};
List<String> list2 = new ArrayList<String>();

//add array items to list
for(String y: morethings)
    list2.add(y);

for(int i =0;i<list2.size();i++){
    System.out.printf("%s ", list2.get(i));
}
```

Java Program to removed duplicates from ArrayList

```
// creating ArrayList with duplicate elements
List<Integer> primes = new ArrayList<Integer>();

primes.add(2);
primes.add(3);
primes.add(5);
primes.add(7);  //duplicate
primes.add(7);
primes.add(11);
```

```
// let's print arraylist with duplicate
System.out.println("list of prime numbers : " + primes);

// Now let's remove duplicate element without affecting order
// LinkedHashSet will guaranteed the order and since it's set
// it will not allow us to insert duplicates.
// repeated elements will automatically filtered.

Set<Integer> primesWithoutDuplicates = new LinkedHashSet<Integer>(primes);

// now let's clear the ArrayList so that we can copy all elements from LinkedHashSet
primes.clear();

// copying elements but without any duplicates
primes.addAll(primesWithoutDuplicates);

System.out.println("list of primes without duplicates : " + primes);
```

```
Output
list of prime numbers : [2, 3, 5, 7, 7, 11]
list of primes without duplicates : [2, 3, 5, 7, 11]
```

2 Ways to Remove Elements/Objects From ArrayList in Java

There are *two ways to remove objects from ArrayList in Java*, first, by using `remove()` method, and second by using `Iterator`. ArrayList provides overloaded `remove()` method, one accept index of object to be removed i.e. `remove(int index)`, and other accept object to be removed, i.e. `remove(Object obj)`. Rule of thumb is, If you know index of object, then use first method, otherwise use second method. By the way, you must remember to use ArrayList remove methods, only when you are not iterating over ArrayList, if you are iterating then use `Iterator.remove()` method, failing to do so may result in `ConcurrentModificationException` in Java. Another gotcha can be occurred due to autoboxing. If you look closely that two remove methods, `remove(int index)` and `remove(Object obj)` are indistinguishable if you are trying to remove from an ArrayList of `Integers`. Suppose you have three objects in ArrayList i.e. `[1,2,3]` and you want to remove second object, which is `2`. You may call `remove(2)`, which is actually a call to `remove(Object)` if consider autoboxing, but will be interpreted as a call to remove 3rd element, by interpreting as `remove(index)`. I have discussed this problem earlier on my article about best practices to follow while overloading methods in Java. Because of lesser known widening rule and

```
List<Integer> numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);
numbers.add(3);

System.out.println("ArrayList contains : " + numbers);

// Calling remove(index)
numbers.remove(1); //removing object at index 1 i.e. 2nd Object,

//Calling remove(object)
numbers.remove(3);
```

```
Output:
ArrayList contains : [1, 2, 3]
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 3
        at java.util.ArrayList.rangeCheck(ArrayList.java:635)
        at java.util.ArrayList.remove(ArrayList.java:474)
        at test.Test.main(Test.java:33)

Java Result: 1
```

You can see that second call is also treated as `remove(index)`. Best way to remove ambiguity is to take out autoboxing and provide actual object, as shown below.

```
System.out.println("ArrayList Before : " + numbers);

// Calling remove(index)
numbers.remove(1); //removing object at index 1 i.e. 2nd Object, which is 2

//Calling remove(object)
numbers.remove(new Integer(3));

System.out.println("ArrayList After : " + numbers);

Output :
ArrayList Before : [1, 2, 3]
ArrayList After : [1]
```

Remove Object From ArrayList using Iterator

This is actually a subtle detail of Java programming, not obvious for first timers, as compiler will not complain, even if you use `remove()` method from `java.util.ArrayList`, while using Iterator. You will only realize your mistake, when you see `ConcurrentModificationException`, which itself is misleading and you may spend countless hours finding another thread, which is modifying that `ArrayList`, because of Concurrent word. Let's see an example.

```
public static void main(String[] args) {

        List<Integer> numbers = new ArrayList<Integer>();
        numbers.add(101);
        numbers.add(200);
        numbers.add(301);
        numbers.add(400);

        System.out.println("ArrayList Before : " + numbers);

        Iterator<Integer> itr = numbers.iterator();

        // remove all even numbers
        while (itr.hasNext()) {
            Integer number = itr.next();

            if (number % 2 == 0) {
                numbers.remove(number);
            }
        }

        System.out.println("ArrayList After : " + numbers);
```

```
Output :

ArrayList Before : [101, 200, 301, 400]

Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:859)
        at java.util.ArrayList$Itr.next(ArrayList.java:831)
        at Testing.main(Testing.java:28)
```

You can `ConcurrentModificationException`, due to call to `remove()` method from `ArrayList`. This is easy in simple examples like this, but in real project, it can be really tough. Now, to fix this exception, just replace call of `numbers.remove()` to `itr.remove()`, this will remove current object you are Iterating, as shown below :

```
System.out.println("ArrayList Before : " + numbers);
Iterator<Integer> itr = numbers.iterator();

// remove all even numbers
while (itr.hasNext()) {
    Integer number = itr.next();

       if (number % 2 == 0) {
       itr.remove();
    }

}

System.out.println("ArrayList After : " + numbers);

Output
ArrayList Before : [101, 200, 301, 400]
ArrayList After : [101, 301]
```

Autoboxing and unboxing in assignment:

This is the most common example of autoboxing in Java, earlier the code was bloated with explicit conversion which is now taken care by compiler.

```
//before autoboxing
Integer iObject = Integer.valueOf(3);
Int iPrimitive = iObject.intValue()

//after java5
Integer iObject = 3; //autobxing - primitive to wrapper conversion
int iPrimitive = iObject; //unboxing - object to primitive conversion
```

```
public void test(int num){
    System.out.println("method with primitive argument");

}

public void test(Integer num){
    System.out.println("method with wrapper argument");

}

//calling overloaded method
AutoboxingTest autoTest = new AutoboxingTest();
int value = 3;
autoTest.test(value); //no autoboxing
Integer iValue = value;
autoTest.test(iValue); //no autoboxing

Output:
method with primitive argument
method with wrapper argument
```

Difference between Method Overloading vs Overriding in Java

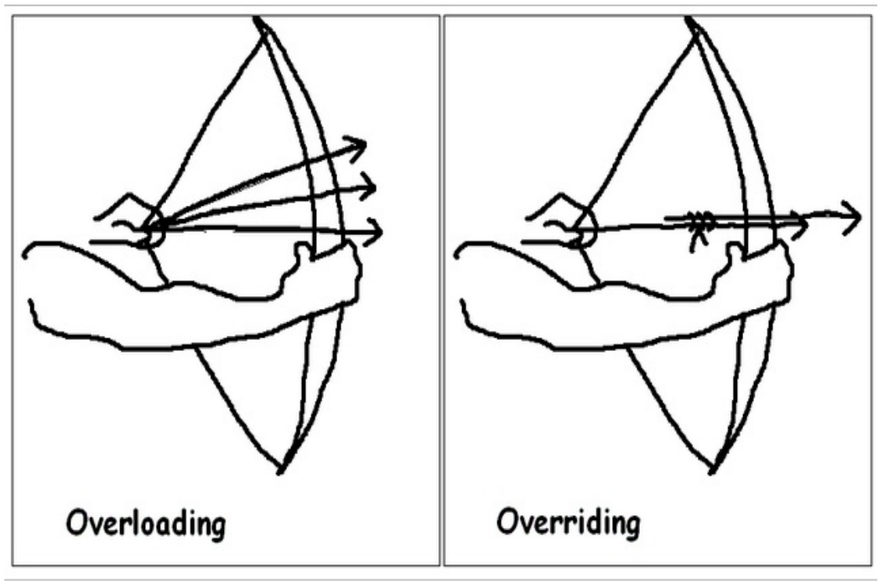
1) First and most important difference between method overloading and overriding is that, In case of method overloading in Java, signature of method changes while in case of method overriding it remain same.

2) Second major difference between method overloading vs overriding in Java is that You can overload method in one class but overriding can only be done on subclass.

3) You can not override `static`, `final` and `private` method in Java but you can overload static, final or private method in Java.

4) Overloaded method in Java is bonded by static binding and overridden methods are subject to dynamic binding.

5) Private and final method can also be not overridden in Java.



Java Tip to create and initialize List in one line

```
//Now here is simple Java tips to create and initialize List in one line
List<String> coolStringList = Arrays.asList("Java", "Scala", "Groovy");
System.err.println(" List created and initialized at same line : " + coolStringList);
```

```
List&lt;String&gt; lst = arrayList.subList(1,3);
```

Vector vs ArrayList in Java

In last section we saw some common properties between both of them and its time to see How much `ArrayList` and `Vector` are different to each other.

1) First and most common *difference between Vector vs ArrayList* is that Vector is synchronized and thread-safe while `ArrayList` is neither Synchronized nor thread-safe. Now, What does that mean? It means if multiple thread try to access `Vector` same time they can do that without compromising Vector's internal state. Same is not true in case of `ArrayList` as methods like `add()`, `remove()` or `get()` is not synchronized.

2) Second major difference on `Vector` vs `ArrayList` is Speed, which is directly related to previous difference. Since Vector is synchronized, its slow and ArrayList is not synchronized its faster than Vector.

Friday, September 7, 2012

What is CopyOnWriteArrayList in Java - Example Tutorial

CopyOnWriteArrayList vs Array List in Java

`CopyOnWriteArrayList` is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin ConcurrentHashMap in Java. `CopyOnWriteArrayList` implements `List` interface like ArrayList, Vector and LinkedList but its a thread-safe collection and it achieves its thread-safety in a slightly different way than `Vector` or other thread-safe collection class. As name suggest `CopyOnWriteArrayList` creates copy of underlying `ArrayList` with every mutation operation e.g. `add` or `set`. Normally `CopyOnWriteArrayList` is very expensive because it involves **costly Array copy** with every write operation but its very efficient if you have a List where Iteration outnumber mutation e.g. you mostly need to iterate the ArrayList and don't modify it too often. Iterator of `CopyOnWriteArrayList` is fail-safe and doesn't throw `ConcurrentModificationException` even if underlying `CopyOnWriteArrayList` is modified once Iteration begins because Iterator is operating on separate copy of ArrayList. Consequently all the updates made on `CopyOnWriteArrayList` is not available to Iterator. In this Java Collection tutorial we will see *What is `CopyOnWriteArrayList` in Java*, *Difference between ArrayList and `CopyOnWriteArrayList` in Java* and One simple Java program example on How to use `CopyOnWriteArrayList` in Java.

1) First and foremost difference between `CopyOnWriteArrayList` and ArrayList in Java is that `CopyOnWriteArrayList` is a thread-safe collection while ArrayList is not thread-safe and can not be used in multi-threaded environment.

2) Second difference between ArrayList and `CopyOnWriteArrayList` is that Iterator of ArrayList is fail-fast and throw `ConcurrentModificationException` once detect any modification in List once iteration begins but Iterator of `CopyOnWriteArrayList` is fail-safe and doesn't throw `ConcurrentModificationException`.

3) Third difference between `CopyOnWriteArrayList` vs ArrayList is that Iterator of former doesn't support remove operation while Iterator of later supports `remove()` operation.

```
CopyOnWriteArrayList<String> threadSafeList = new CopyOnWriteArrayList<String>();
threadSafeList.add("Java");
threadSafeList.add("J2EE");
threadSafeList.add("Collection");

//add, remove operator is not supported by CopyOnWriteArrayList iterator
Iterator<String> failSafeIterator = threadSafeList.iterator();
while(failSafeIterator.hasNext()){
    System.out.printf("Read from CopyOnWriteArrayList : %s %n", failSafeIterator.next());
    failSafeIterator.remove(); //not supported in CopyOnWriteArrayList in Java
```

fail-fast Iterators in Java

As name suggest **fail-fast Iterators** fail as soon as they realized that *structure of Collection has been changed since iteration has begun.* Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection. fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws `ConcurrentModificationException`.

fail-safe Iterator in java

Contrary to fail-fast Iterator, **fail-safe iterator** doesn't throw any Exception if Collection is modified structurally while one thread is Iterating over it because they work on clone of Collection instead of original collection and that's why they are called as fail-safe iterator. Iterator of `CopyOnWriteArrayList` is an example of fail-safe Iterator also iterator written by `ConcurrentHashMap` keySet is also fail-safe iterator and never throw `ConcurrentModificationException` in Java.

How to create read only Collection in Java

Read only Collection in Java

You can create read only Collection by using `Collections.unmodifiableCollection()` utiltiy method. it returns unmodifiable or **readonly view of Collection** in which you can not perform any operation which will change the collection like `add()`, `remove()` and `set()` either directly or while iterating over iterator. It raise `UnsupportedOperationException` whenever you try to modify the List. One of the common misconception around read only `ArrayList` is that, you can create read only `arrayList` by using `Arrays.asList(String{[])`, which is apparently not true as this method only return a **fixed size list** on which `add()` and `remove()` are not allowed by `set()` method is still allowed which can change the contents of `ArrayList`. `Collections` class also provide different method to make `List` and `Set` read only. In this Java tutorial we will learn How to make any collection read only and How to create fixed size List as well.

```
//making existing ArrayList readonly in Java
ArrayList readableList = new ArrayList();
readableList.add("Jeffrey Archer");
readableList.add("Khalid Hussain");

List unmodifiableList = Collections.unmodifiableList(readableList);

//add will throw Exception because List is readonly
unmodifiableList.add("R.K. Narayan");
```

```
        //creating read only Collection in Java
        Collection readOnlyCollection = Collections.unmodifiableCollection(new
ArrayList<String>());
        readOnlyCollection.add("Sydney Sheldon"); //raises UnSupportedOperation exception
```