Assignment 6 Supplemental

Traversing a List with Iterators

When processing data in a List, often times this is done so in linear sequence. For example, we may want to output all of the values in a List. To do this, we would traverse the list from beginning to end, outputting each element as we go.

```
for (int index = 0; index < list.size(); index++)

      System.out.println(list.get(index));
```

This is particularly costly for data structures like Linked Lists, as each access requires a traversal from the beginning or end (in the case of Doubly Linked Lists) of the Linked List to the element being accessed.

Iterators are a mechanism used with Lists that we as programmers can use to traverse a list ourselves, and retain a position in the List, moving it forward as necessary. The List interface contains the iterator() method. This method will return a new Iterator over the List. An Iterator is typed with the type of data the List holds.

```
Iterator<String> itr = list.iterator();
```

Assuming list is an existing List of Strings, this will create a new Iterator for list. A new Iterator does not refer to any element in the list. When Iterator's next() method is first called, it will then refer to the 0^{th} element and return the data at the 0^{th} element. Each subsequent call to next() will move the iterator forward one element and return the value residing there. It can be determined if the Iterator is at the end of a List with its hasNext() method. hasNext() returns a boolean; true if there is an element following the Iterator's current position, false if there is not.

```
Iterator<String> itr = list.iterator();

while (itr.hasNext())

      System.out.println(itr.next());
```

This will do the same thing as the for loop, with one important difference. For a Linked List, this will not have to traverse to the current element for every access. The Iterator will hold its position until moved.

Iterators should be used with Linked List when multiple, sequential accesses will be made.

To see everything that Iterator can do, look at the full documentation.

If you need an Iterator over a List that can go forwards and backwards check out ListIterator.

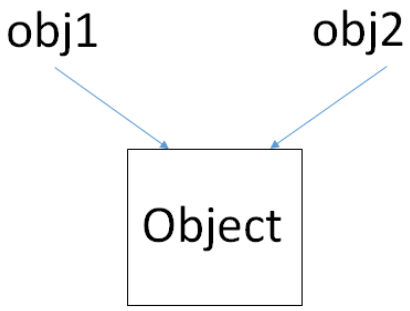
Shallow vs. Deep Copy and the Cloneable interface

Pertaining to reference types, what does it mean to make a copy? What exactly is copied? If we look at basic assignment of reference types:

```
Object obj1 = new Object();

Object obj2 = obj1;
```

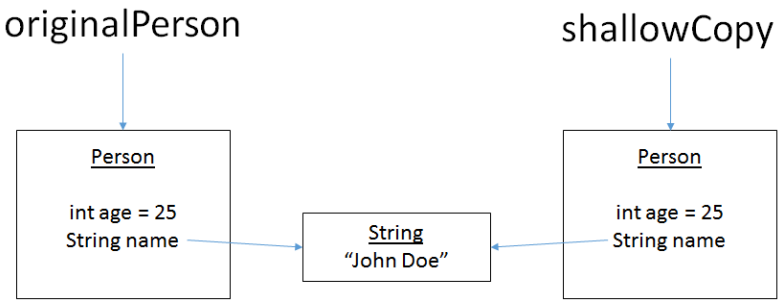
We know that the object itself isn't copied, only the reference is copied.



If we want an object to be copied, such that a new object is created, there is an interface called Cloneable that can be implemented. The Cloneable interface requires us to override the clone() method. The purpose of the clone() method is to generate and return a copy of the calling object. This is initially done by calling super.clone(). If the class does not have any subclasses that are Cloneable, this will call upon Object's clone method which creates what is referred to as a "Shallow Copy".

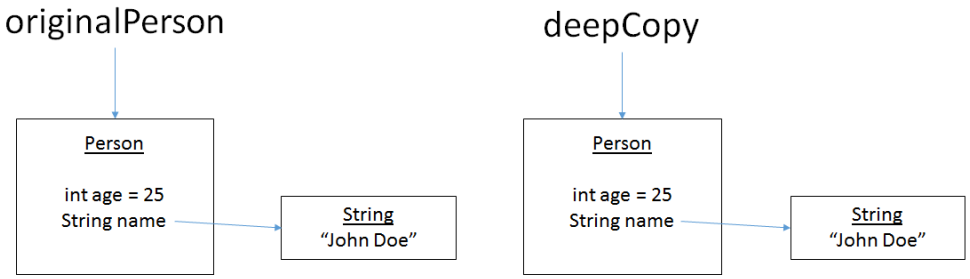
Shallow Copy

A Shallow Copy is a new object that makes copies of all data members through assignment. Assignment of primitive types makes a copy of the value, however, recalling from above, an assignment of a reference type only copies the reference, not the object that is being referred to. Notice here, the primitive int "age" value is copied, but the reference String "name" both refer to the same String object. If name is altered in originalPerson, it is also altered in shallowCopy since they refer to the exact same object.

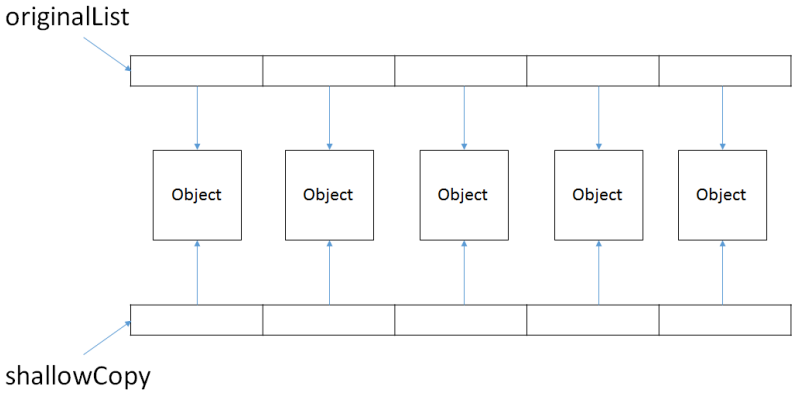


Deep Copy

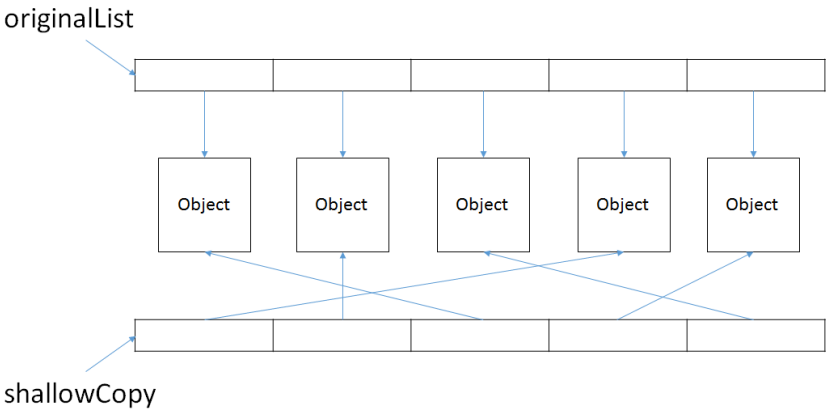
A Deep Copy is a copy that also generates copies of the objects themselves. Now if name is altered in originalPerson, it will have no effect on deepCopy's name.



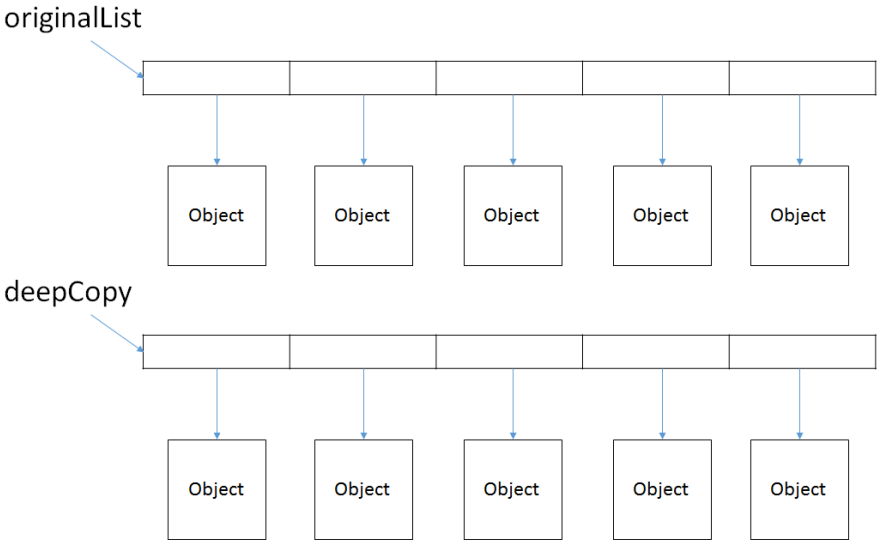
Another example using a collection, such as an Array List. Similar concepts apply to Linked Lists and other data structures.



The same ideas apply to collections, as they store references to objects, not the objects themselves. We can alter the order of the copy, but they are still referring to the same objects in memory, so if an object is altered it effects both the original and the Shallow Copy.



A Deep Copy of a collection would also make copies of all of the objects it refers to. The copies of the objects referred to should be Deep Copies as well.



Cloneable Interface

When implementing the clone method, super.clone() will create a Shallow copy. The copy generated will have copies of primitives, but only the references to reference types will be copied. To make a Deep Copy, update the Shallow Copy's reference type members to refer to new objects that contain the same data as the original objects.

Object's clone() method throws a CloneNotSupportedException which is a checked Exception. This means when super.clone() is called, this Exception must be handled. A common way to handle this is to catch it, and in turn ,throw a new RuntimeException.

The clone() method in Cloneable returns an Object reference. Whatever type is being cloned, it will be returned to as an Object and may have to be cast when using it.

Documentation for Cloneable can be found here.