1. **A and E are classes, B and D are interfaces, C is an abstract class.Which would be correct? (Multiple answers)**

a. `class F implements B,C { }`

b. `class F implements B { }`

c. `class F extends A,E { }`

d. `class F extends E { }`

e. `class F implements B,D { }`

(a) is wrong because "implements" is used with interfaces (C is a class) and (c) is wrong because you can only inherit in Java from ONE class.

2. Garbage collection guarantees that a program will not run out of memory.

a. True.

b. False.

First of all, the garbage collector can only free memory which can no longer be accessed (because for instance the reference was overwritten by another reference). Add a few million of objects to a collection, you'll have problems ... But what I have seen quite often is simply programs that were calling "new" very fast, and faster than the garbage collector could free memory. Crashing after a few hours.

3. Every try block must be followed by a catch block or a finally block.

a. True.

b. False

No comment.

4. Deletion is faster in LinkedList than ArrayList.

a. True.

b. False.

In an ArrayList Java will recompact the structure (no gaps). In a LinkedList you just need to remove one node and make the preceding one directly reference the next one.

5. When does a "finally" block get executed?

a. Always when the "try" block gets executed, no matter if an exception occurred or not.

b. Always when a method gets executed, no matter if an exception occurred or not.

c. When a "try" block gets executed, if no exception occurs.

d. When an exception occurs in the "try" block code.

Note that this is also true if you return from the method from inside the try block.

6. The last value in an array called ar can be found at index:

a. 0

b. 1

c. ar.length

d. ar.length - 1

Easy one, because of the 0 ... n-1 numbering of arrays.

7. Which of the following is not a subinterface of Collection?

a. List

b. Set

c. SortedSet

d. Map

Map is usually presented as part of the "Collection Framework", which is a convenient expression for talking about structures where you can store objects. "Collection" refers to a special interface with special methods, and shouldn't be confused with the "Collections" class and its static methods.

8. **For the difference between ArrayLists and LinkedList, which is correct? (Multiple answers)**

a. The ArrayList data structure is implemented by a dynamic array, the LinkedList data structure is based on nodes holding references.

b. For random access (get and set), an ArrayList performs better than a LinkedList because a LinkedList needs to follow references.

c. For add and remove operations, a LinkedList is preferred because the ArrayList needs to move data.

d. An ArrayList is ordered by construct, not a LinkedList.

Neither is. Trees are ordered by construct.

e. To store the same amount of data, an ArrayList will always use more storage than a LinkedList.

It's the opposite, LinkedLists also store a reference to the next element.

9. Which of the following is right?

```
import java.util.*;
public class TestListSet{
   public static void main(String args[]){
      List list = new ArrayList();
      list.add("Hello");
      list.add("Learn");
      list.add("Hello");
      list.add("Welcome");
      Set set = new HashSet();
      set.addAll(list);
          // Populate the HashSet with the list values
      System.out.println(set.size());
   }
}
```

a. Compilation fails

b. Run-time exception

c. Output is 3

HashSet, being a set, has no duplicates and stores only one "Hello"

d. Output is 4

10. What is the output of this program (tricky)?

```
import java.util.Collection;
import java.util.HashSet;
import java.util.ArrayList;
import java.util.HashMap;

abstract static class Super {

    public static String
          getType(Collection<?> collection) {
      return "Super:collection";
    }

    public String getType(ArrayList<?> list) {
      return "Super:arrayList";
    }
```

10. What is the output of this program (tricky)?

```
    public String getType(HashSet<?> set) {
      return "Super:hashSet";
    }
  }

  static class Sub extends Super {
    public static String
          getType(Collection<?> collection) {
       return "Sub:collection";
    }
  }
}
```

10. What is the output of this program (tricky)?

```
public class Demo {

  public static void main(String[] args) {
    // <?> is a wild card -it stands for "any type"
    Collection<?>[] collections =
              {new HashSet<String>(),
               new ArrayList<String>(),
               new HashMap<String, String>().values()};
    Super subToSuper = new Sub();
    for(Collection<?> collection: collections) {
      System.out.print(subToSuper.getType(collection)
                       + " ");
    }
    System.out.println("");
  }
}
```

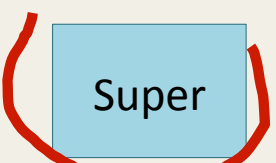
```
for(Collection<?> collection: collections) {
```

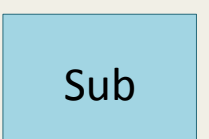
What is passed to getType is a Collection, so it can only be (a) or(c).

a. Sub:collection Sub:collection Sub:collection

b. Super:hashSet Super:arrayList Sub:collection

c. Super:collection Super:collection Super:collection






```
static String getType(Collection)
       String getType(ArrayList)
       String getType(HashSet)
```

```
static String getType(Collection)
```

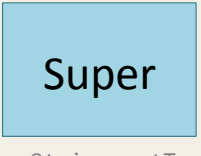
subToSuper is a Super reference. BECAUSE METHODS ARE STATIC it's the one of Super that is called.

If methods were NOT static, as we call the creator of Sub(), the method associated with the OBJECT would be overriden.

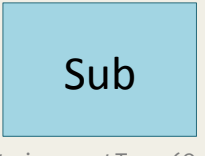
a. Sub:collection Sub:collection Sub:collection

b. Super:hashSet Super:arrayList Sub:collection

c. Super:collection Super:collection Super:collection




```
String getType(Collection)
String getType(ArrayList)
String getType(HashSet)
```




```
String getType(Collection)
```

Assignment between **class** and **subclass**

```
class GeoPoint { ... }

class City extends GeoPoint { ... }

public class InheritanceAssignment {

   public static void main(String[] args) {
       City     c = new City("Singapore",
                             1.28333, 103.8333);
       GeoPoint g;

       c.setPopulation(5610000);
       g = c;
       System.out.println("GeoPoint is " + g.toString());
   }
}
```

Similar problem with this example. Here toString() is NOT static and overriden in City (in GeoPoint it shows name, latitude and longitude, in City it adds the population).

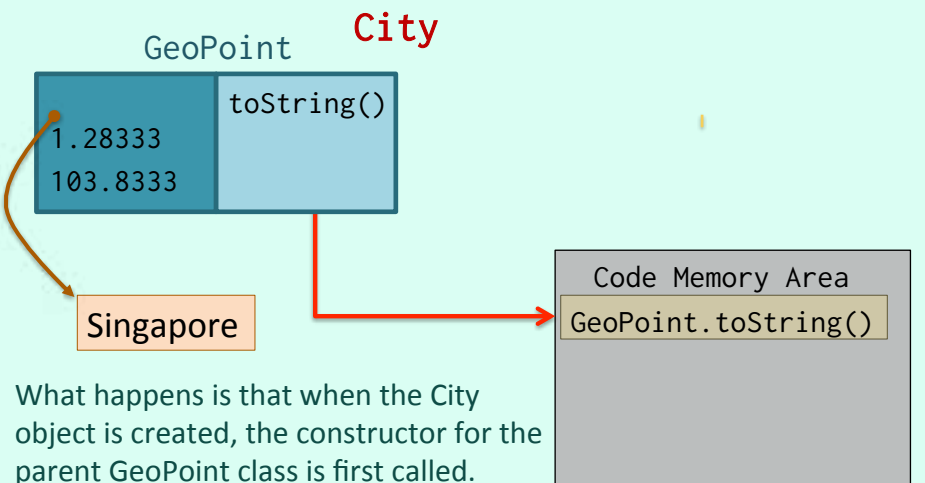
```
$ java InheritanceAssignment
GeoPoint is Singapore (1.28333,103.8333) - pop. 5610000
$
```

City version of method

Although we have assigned the reference to a GeoPoint reference, what is called is the method of the City because, as it's not static, it just replaced the original one (note that a language such as C++ behaves differently and would call the GeoPoint toString() method in this case)

Assignment between **class** and **subclass**

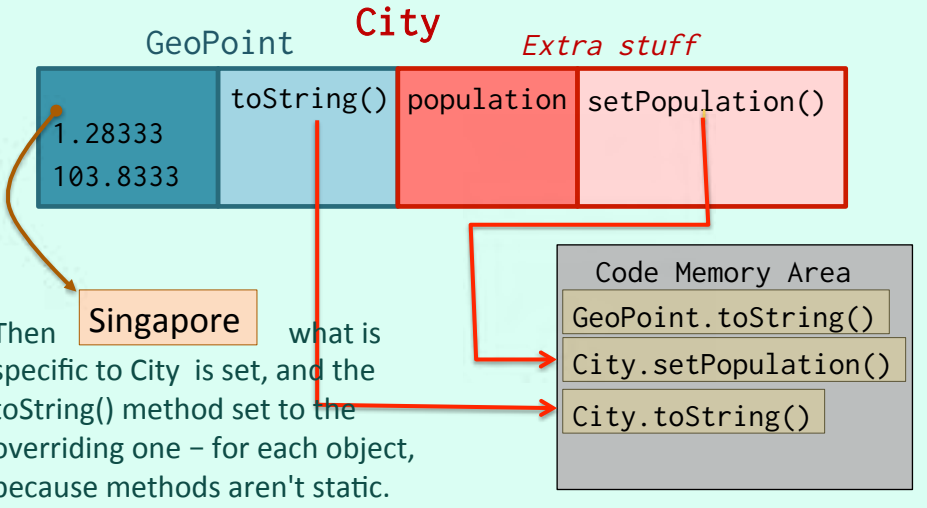
```
City  c = new City("Singapore", 1.28333, 103.8333);
```



What happens is that when the City object is created, the constructor for the parent GeoPoint class is first called.

Assignment between **class** and **subclass**

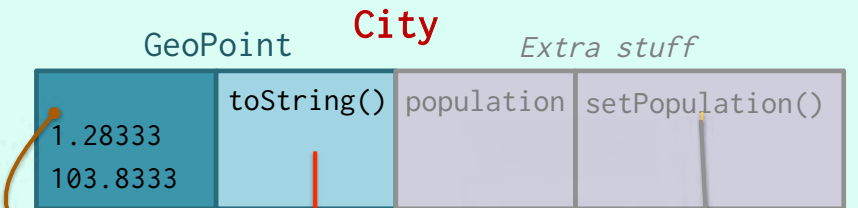
```
City  c = new City("Singapore", 1.28333, 103.8333);
```



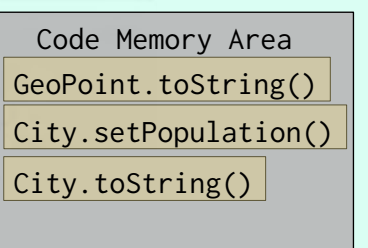
Then Singapore what is specific to City is set, and the toString() method set to the overriding one – for each object, because methods aren't static.

Assignment between **class** and **subclass**

```
City  c = new City("Singapore", 1.28333, 103.8333);
GeoPoint g = c;
```



When you are assigning the reference to a Singapore GeoPoint reference, you are basically saying to Java "look at this as if it were a GeoPoint object". Except that the method pointer, being in the object, was changed.



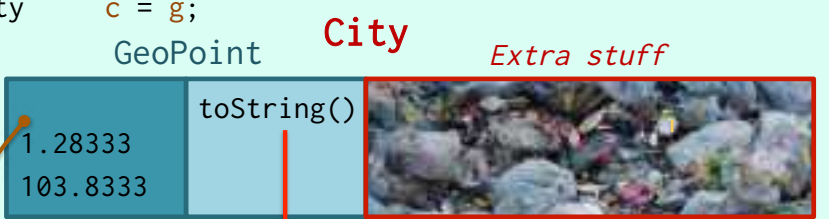
What about the opposite?

```
GeoPoint g = new GeoPoint("Singapore", 1.28333, 103.8333);
City     c;

c = g;
```

What about the opposite?

```
GeoPoint g = new GeoPoint("Singapore", 1.28333, 103.8333);
City     c = g;
```



It fails because you would say to Java "Now look at this larger bit of memory (remember that City "extends" GeoPoint – and nothing was set up there, so it's mostly garbage.

Singapore

Code Memory Area

GeoPoint.toString()

Every City is a GeoPoint

```
g = c;
```

Every GeoPoint is NOT a City

```
c = g;
```

Which of course makes sense.

Playing with HashMaps

Some simple HashMaps examples.

```
static void mystery1(HashMap<String, String> map) {
   HashMap<String, String> result
               = new HashMap<String, String>();
   for (String k : map.keySet()) {
      String v = map.get(k);
      if (k.charAt(0) <= v.charAt(0)) {
        result.put(k, v);
      } else {
        result.put(v, k);
      }
   }
   System.out.println(result);
}
```

This one switches key and value if the first letter of the value is smaller than the first value of the key.

```
public static void main(String[] args) {
        HashMap<String,String> map =
                    new HashMap<String,String>();
        map.put("dark", "bright");
        map.put("low", "high");
        map.put("below", "above");
        map.put("before", "after");
        map.put("slow", "fast");
        map.put("difficult", "easy");
        mystery1(map);
}
```

Here is the data – everything is swapped except "difficult" and "easy"

```
{high=low, fast=slow, above=below,
 bright=dark, difficult=easy, after=before}
```

```
static void mystery2(HashMap<Integer, String> map1,
                     HashMap<Integer, String> map2) {
    HashMap<String, String> result =
                new HashMap<String,String>();
    for (Integer n : map1.keySet()) {
       if (map2.containsKey(n)) {
         result.put(map1.get(n), map2.get(n));
       }
    }
    System.out.println(result);
}
```

Other methods. The two maps are matched on the key, and the values assigned together to the resulting map.

```
public static void main(String[] args) {
     HashMap<Integer,String> map1 =
               new HashMap<Integer,String>();
     HashMap<Integer,String> map2 =
               new HashMap<Integer,String>();
     map1.put(1, "Argentina");
     map1.put(2, "Turkey");
     map1.put(3, "Denmark");
     map1.put(4, "Finland");
     map2.put(1, "Buenos Aires");
     map2.put(3, "Copenhagen");
     map2.put(4, "Helsinki");
     map2.put(5, "Cairo");
     mystery2(map1, map2);
}
```

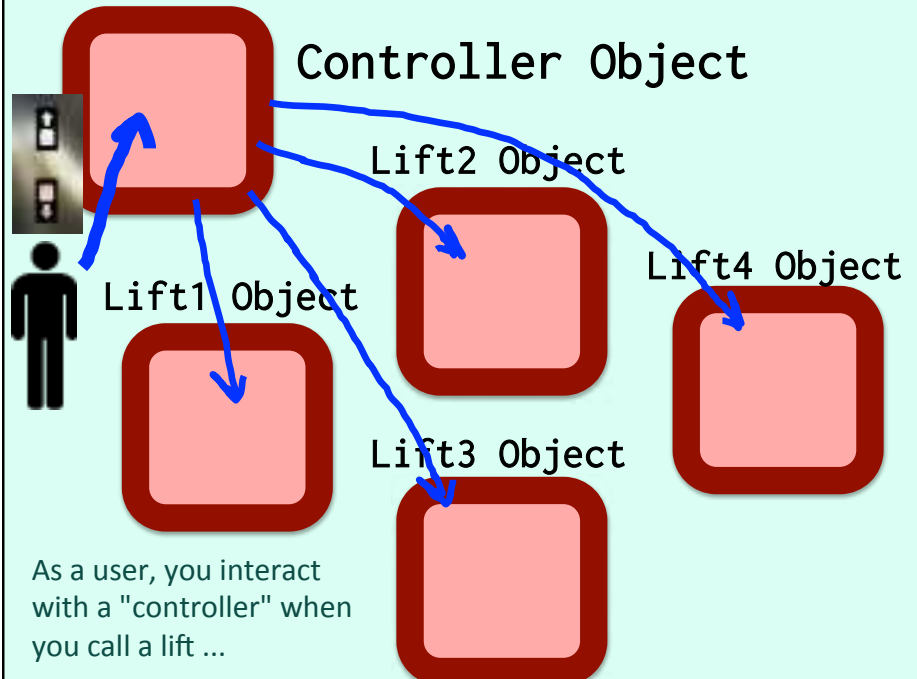
This is a technique sometimes used in databases (known as "hash join")

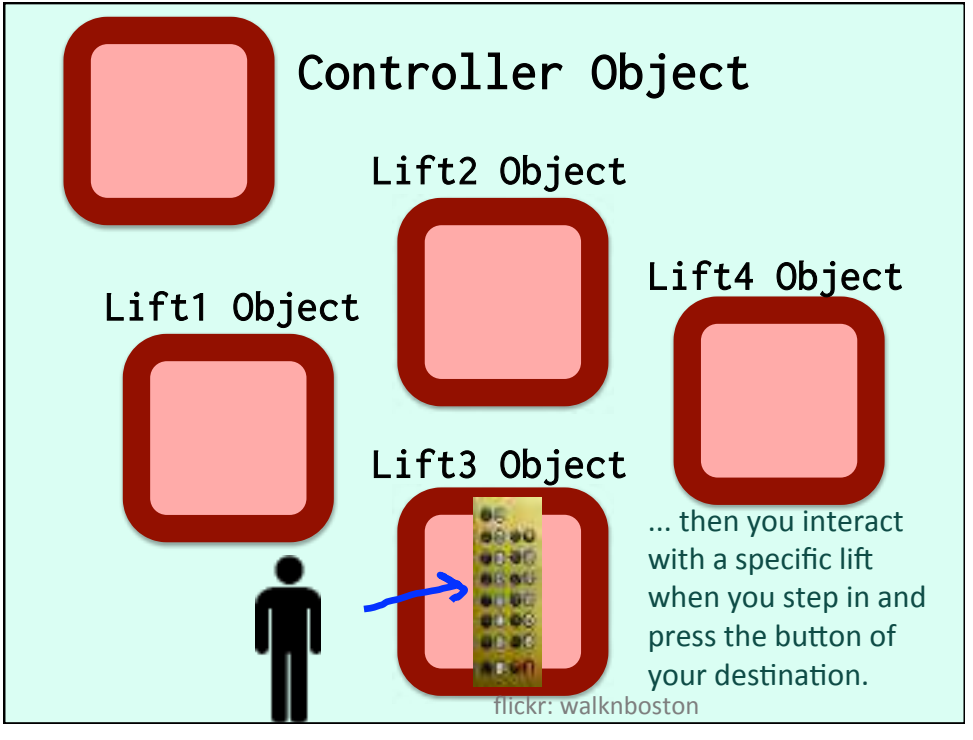
```
{Argentina=Buenos Aires, Finland=Helsinki,
 Denmark=Copenhagen}
```

Elevators (lifts) can be an interesting problem because requests cannot be answered on the spot, they have to be stored until they can be satisfied − which means finding the suitable collections.

Elevators 

Lifts 


Controller Object
Lift2 Object
Lift4 Object
Lift1 Object
Lift3 Object
As a user, you interact with a "controller" when you call a lift ...


Controller Object
Lift2 Object
Lift4 Object
Lift1 Object
Lift3 Object
... then you interact with a specific lift when you step in and press the button of your destination.
flickr: walknboston

Request

Requests received by a controller are made of a floor and a direction. They are not be going to be served in order (if the lift is three floors below and going up, someone going up on the floor below can call it after you and step into the lift before you)

from_floor direction

unique

However, if you press the "call" button several times, nothing will happen. The (floor, direction) combination is unique, which suggests using a set.

SET

Lift status

Aboard the lift, it's different. There is obviously a sequence of floors to serve. You have stop request from people inside, but also request from the controller to pick up someone.

current_floor

next_stop

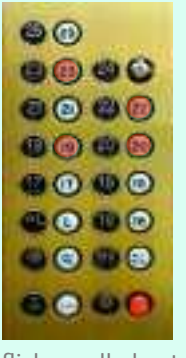
direction

Lift calls

LIST

Floors to serve might be store in a list. Note that a boolean array (stop here/don't stope here) could also do the job.

The problem of course if for the controller to assign a task to a lift so as to minimize waits for people and movement (meaning electricity consumption) for lifts.



flickr: walknboston

Interview Questions

If you are interviewing for a Java programming job, sometimes you can get this type of question.

What is a hash Code?

In java the Object class (that all objects extend) implements a hashCode() method, which means that you can call it for any object. Technically speaking, a hashcode is a small result (not necessarily numerical, just a small number of bytes) obtained by a mathematical computation ("hash function") applied to a "value" − which can perfectly be a bunch of bytes in memory. The qualities required of a hash function are to generate a large number of different values, and to "spread" as regularly as possible over a large interval the computations applied to "values", even when most of these values are clustered. Several values can hash into the same hash code, but the less often it happens, the better.

What is the super class for Exception and Error?

Throwable. The interesting methods of Exception and Error (getMessage(), printStackTrace(), and so forth) are all inherited from Throwable. Exception and Error only implement constructors.

What is the difference between HashTable and HashMap?

Pretty much the same thing, it's similar to "Vector" compared to "ArrayList". Vector and HashTable are old (the right word is "legacy") classes that have been around since before the "Collection Framework" was implemented. They are "synchronized" (more about this later) which means that you can have several separate threads accessing them without conflit at the same time. You can synchronize other collections, but they aren't by default, which means that they are faster when you don't need multithreading, and you should rather use ArrayList and HashMap. Other than that, you can have one null key and several null values in a HashMap, which I don't regard as really sound, not in a HashTable.