Object Oriented Programming: Objects are instances of a particular class or subclass with the class's. The classes can have methods or procedures and data variables. An object is what actually runs in the computer.

Polymorphism: Polymorphism is when multiple subclasses of the same type (Animal) can call the same method such as make sound. The make sound method is inherited from the super class as part of the instance of that specific object. Polymorphism in this example is using the same method for different types of objects (sub-classes/different animals).

Inheritance is where a subclass (or object) inherits a method from a super class. Objects inherit methods, procedures, and data variables from classes.

Encapsulation: In Animal, all the data is encapsulated (hidden/protected/shelled) in one class such as sound, type, height, color, etc...

Abstraction: When an operation (ie: sleeping) in super class Animal is not needed by a subclass, it is an abstraction. When it is needed, it is **concrete**.

Class: In Object Oriented Programming a class is a program-code-template for creating/instantiating objects, providing initial values for state member variables and implementations of functions and methods.

String Functions:

C#: Constructor -

```
public ActionResult Name (string firstName, string lastName)
{
string fullName = ( "My full name is: " +(firstName + lastName));
return PartialView (fullName);
}
```

C#: View-

```
@Model
---------------------
int x = 11;

string y = "How old are you?  ";

static void main()
{
return (x + y);
}
}
----------------------
```

Java and C++ Programming Tutorials.
Java Tutorial 5: Inheritance and Polymorphism

Java Tutorial

Inheritance and polymorphism: two big words to strike fear into the heart of any new Java programmer. However, the concepts that they refer to are not that complex.

Inheritance in Java

Let's take a look first at inheritance. Inheritance allows you to create child classes of existing classes. Why would you want to do such a thing? Well let's say you've got a class called Robot. Robot contains nothing other than some basic methods needed by all Robots. For example:

```
public class Robot {
  public void start() {
    System.out.println("Robot started.");
 }

  public void work() {
     System.out.println("Robot working.");
 }

  public void stop() {
     System.out.println("Robot stopped.");
 }
}
```

As you can see, the Robot class merely defines start(), work() and stop() methods, and each method merely prints out what it's supposed to do.

We'll create a class with a main method to run Robot.

```
public class Application {

  public static void main(String[] args) {
     Robot robot = new Robot();

     robot.start();
    robot.work();
     robot.stop();   }}
```

Running this program produces the following output:

```
Robot started.
Robot working.
Robot stopped.
```

All very well. But what if we want to create particular types of Robots that "inherit" all the functionality of Robot, but also add new functionality?

For instance, let's create a WasteDisposalRobot:

```
public class WasteDisposalRobot extends Robot {
    public void findWaste() {
      System.out.println("Finding waste");
   }
}
```

Using the keyword **extends**, we've create a WasteDisposalRobot that can do everything that Robot does, and also adds a findWaste() method.

We can use it as follows:

```
WasteDisposalRobot robot = new WasteDisposalRobot();

robot.start();
robot.findWaste();
robot.work();
robot.stop();
```

```
Robot started.
Finding waste
Robot working.
Robot stopped.
```

What we have here is inheritance at work. The WasteDisposalRobot is a **subclass** or child class of the Robot class; the Robot class is its **superclass** or parent class.

WasteDispoalRobot can do everything that Robot can do, and we can add new methods (and data) to it also.

Method Overriding

What if we've created a new subclass of some superclass, but we don't like one or more of the methods in the superclass? We'd like to change it to do something else. We can do that simply by defining the method again in the child class.

In the following code, we've **overridden** the work() method in the Robot parent class so that it does something different in the WasteDisposalRobot child class.

```
public class WasteDisposalRobot extends Robot {
  public void findWaste() {
      System.out.println("Finding waste");
   }

   @Override
  public void work() {
       System.out.println("Disposing waste!");
    }
}
```

```
Robot started.
Finding waste
Disposing waste!
Robot stopped.
```

Note the **@Override** directive just before the overridden method. This is not obligatory, but you should always use it. It tells the Java compiler that you intend to override a method in the parent class. If you misspell the method name and try to override a method that does not exist, the compiler will warn you by throwing an error.

Instance Variable Inheritance

It might occur to you to wonder what happens with instance variables. If the Robot superclass has some instance variables, do the child classes also have access to these variables? This depends on whether you define the instance variables in the parent class using the **public**, **private** or **protected** access specifiers, or with none at all. We'll look at this in more detail later, but for now let's just say that as long as instance variables are not **private**, they can be accessed by subclasses.

Let's see an example. Here I've placed all relevant code in one file to make it easier to read.

```
class Fruit {
   String name;

   Fruit() {
      name = "Fruit";
    }

   public String getName() {
      return name;
   }
}

class Banana extends Fruit {
  Banana() {
     name = "Banana";
   }
}



public class Application {


    public static void main(String[] args) {
       Fruit fruit = new Fruit();
     Banana banana = new Banana();

       System.out.println(fruit.getName());
       System.out.println(banana.getName());
  }

}
```

The constructors of both classes set the *name* instance variable. The Banana class extends the Fruit class (i.e. inherits from it); its constructor also has access to the *name* instance variable, which it sets. Then when *name* is retrieved from either class using the getName() method, an appropriate name is returned and displayed.

```
Fruit
Banana
```

Constructor Inheritance

Constructors are inherited like other methods, and in fact when you construct a child object, the default constructor of its parent is called automatically first.

```
class Fruit {
   Fruit() {
      System.out.println("Fruit constructed");
   }

}

class Banana extends Fruit {
 Banana() {
     System.out.println("Banana constructed");
  }
}

public class Application {

   public static void main(String[] args) {
       Banana banana = new Banana();
  }
}
```

```
Fruit constructed
Banana constructed
```

If there is no default constructor in the parent class, you **must** define a constructor explicitly in the child class. If you want, you can then call the appropriate constructor in the parent class using the **super** keyword.

```
class Fruit {
    Fruit(String name) {
       System.out.println("Fruit constructed with name: " + name);
    }

}

class Banana extends Fruit {
 Banana() {
     super("Banana");
   }
}
```

```
public class Application {

   public static void main(String[] args) {
       Banana banana = new Banana();
  }
}
```

```
Fruit constructed with name: Banana
```

Polymorphism in Java

Polymorphism: big word, simple concept. Its literal meaning is "many shapes". But that tells you nothing. Polymorphism just means that, basically, once you've got a child class, you can use objects of that child class wherever you'd use objects of the parent class. Java will automatically invoke the right methods.

For instance, even if we have a variable with the type of a parent class, we can assign it to a child class and we can call overridden methods in the child class using that variable.

Let's see an example.

```
class Fruit {

   public void show() {
       System.out.println("Fruit");
   }
}

class Banana extends Fruit {

   @Override
  public void show() {
       System.out.println("Banana");
  }

   public void makeBananaTree() {
     System.out.println("Making a tree");
   }
}

public class Application {

   public static void main(String[] args) {
       Fruit banana = new Banana();
```

```
        banana.show();

        // The following WILL NOT work;
         // Variables of type Fruit know only
        // about Fruit methods.
         // banana.makeBananaTree();
    }
}
```

```
Banana
```

Of course, you can't assign a Banana to a Fruit variable and then use it to call methods that belong only to Banana and not to Fruit. Fruit only knows about Fruit methods. A variable of the type of a particular class knows only about methods defined in that particular class and its superclasses. It doesn't know about methods defined in subclasses, even though you can assign objects of subclass types to the variable (Banana objects to Fruit variables, as in this example) and the overridden methods will be correctly called.

More Stuff...

Next: Java Tutorial 6: Useful Standard Methods, Access Modifiers and Abstract Classes

Previous: Java Tutorial 4: Interfaces, and a Basic Swing App

Click here to see more in "Java Tutorial"

Java Collections Interview Questions and Answers

1. How to filter a Java collection?

The best way to filter a Java collection is to use Java 8. Java streams and lambdas can be used to filter a collection as below,

```
List<Person> passedStudents = students.stream()
    .filter(p -> p.getMark() > 50).collect(Collectors.toList());
```

If for some reason you are not in a position to use Java 8 or the interviewer insists on pre Java 8 solution, following is the best way to filter a Java collection.

2. How to Sort a Java Collection?

Use a Comparator to sort a Java Collection.

```
List<Animal> animals = new ArrayList<Animal>();
Comparator<Animal> comparator = new Comparator<Animal>() {
    public int compare(Animal c1, Animal c2) {
        //sort logic here
         return c2.getHeight() - c1.getHeight();
    }
};

Collections.sort(animals, comparator);
```

If Animal implements Comparable, then following is just enough.

```
Collections.sort(animals);
```

3. Best way to convert a List to a Set.

Instantiate Set using the HashSet.

```
Set<Animal> animalSet = new HashSet<Animal>(animalList);
```

4. When to use LinkedList over ArrayList?

Java's LinkedList implementation is a doubly linked list. ArrayList is a dynamically resizing array implementation. So to compare between LinkedList and ArrayList is almost similar to comparing a doubly linked list and a dynamically resizing array.

LinkedList is convenient for back and forth traversal sequentially, but random access to an element is proportionally costlier to the size of the LinkedList. At the same time, ArrayList is best suited for random access using a position.

LinkedList is best for inserting and deleting an element at any place of the LinkedList. An ArrayList is not suited for inserting or deleting elements in the mid of the ArrayList. Since everytime a new element is inserted, all the elements should be shifted down and dynamic resizing should be done.

With respect to memory usage of LinkedList and ArrayList, LinkedList collection uses more memory as it needs to keep pointers to the adjacent elements. This overhead is not present for the ArrayList, just the memory required for the data is sufficient. Consider these factors and decide between LinkedList or ArrayList depending on the use case.

5. **Difference between HashMap and Hashtable.**
    - HashMap is not synchronized but Hashtable is synchronized.
    - HashMap allows null as key and value. Since the key is unique, only one null is allowed as key. Hashtable does not allows null in key or value.
    - LinkedHashMap extends HashMap and so can be converted. It helps to have a fixed iteration order. It is not possible with Hashtable.
    - In essense, there is almost no reason to use a Java Hashtable.
    -
6. **Explain Java hashCode() and equals() method.**

    equals() method is used to determine the equality of two Java objects. When we have a custom class we need to override the equals() method and provide an implementation so that it can be used to find the equality between two instance of it. By Java specification there is a contract between equals() and hashCode(). It says,

    "*if two objects are equal, that is obj1.equals(obj2) is true then, obj1.hashCode() and obj2.hashCode() must return same integer*"

    Whenever we choose to override equals(), then we must override the hashCode() method. hashCode() is used to calculate the position bucket and keys.

7. **What is Java Priority Queue?**

    Java PriorityQueue is a data structure that is part of Java collections framework. It is an implementation of a Queue wherein the order of elements will be decided based on priority of each elements. A comparator can be provided in the constructor when a PriorityQueue is instantiated. That comparator will decide the sort order of elements in the PriorityQueue collection instance.

8. **Difference between ArrayList and Vector.**

    We have beaten this enough in a old article difference between Vector and ArrayList in Java.

    - Vector is synchronized and ArrayList is not.
    - Vector doubles its internal size when its increased. But, ArrayList increases by half of its size when its increased.
    - ArrayList gives better performance over Vector as its not synchronized.
    - ArrayList's Iterators are fail-fast but Vector's Enumeration is not fail-fast.
    - ArrayList was introduced in Java 1.2 and Vector even before that. But initially Vector was not part of Java collections framework and later made part of collections framework.
    - As of now, there is no need to use Vector and it can be considered legacy and to be deprecated. If you need synchronized collection an ArrayList can be synchronized and used.
9. **What are Java Concurrent Collection Classes?**

Concurrent Collections were introduced in Java 5 along with annotations and generics. These classes are in java.util.concurrent package and they help solve common concurrency problems. They are efficient and helps us to reduce common boilerplate concurrency code. Important concurrent collection classes are BlockingQueue, ConcurrentMap, ConcurrentNavigableMap and ExecutorService.

10. Explain about Comparable and Comparator

A class can implement the Comparable interface to define the natural ordering of the objects. If you take a list of Strings, generally it is ordered by alphabetical comparisons. So when a String class is created, it can be made to implement Comparable interface and override the compareTo method to provide the comparison definition. We can use them as,

```
str1.compareTo(str2);
```

Now, what will you do if you want to compare two strings based on it length. We go for the Comparator. We create a class and let it implement the Comparator interface and override compare method. We can use them as,

```
Collections.sort(listOfStrings, comparatorObj);
```

The natural ordering is up to the person designing the classes. Comparator can be used in that scenario also and it can be used when we need multiple sorting options. Imagine a situation where a class is already available and we cannot modify it. In that case also, Comparator is the choice.

Java Collections Interview Questions and Answers

1. How Java `HashMap` works?

`HashMap` is a key-value pair data structure. Each key will have a corresponding value and the key is the identifier for that value.

Internally, these key-value pairs are stored in logical blocks (buckets). When a pair is put in a `HashMap`, its key is used to compute hash code and that hash code identifies a bucket. Imagine it as an array index or a logical address or a door number. The key-value pair is stored at the bucket where the hash code points to. A bucket can have more than one key-value pairs stored in it.

When a value is looked upon using its key, first the hashcode is computed and the pointing bucket is reached. Then if that bucket has multiple pairs, then each of the 'key's are compared using the `equals` method to identify the matching pair.

Refer this tutorial to know about what is hashcode and how it works? To know about buckets and hashing refer Java Hashtable tutorial.

2. What are fail-fast and fail-safe Iterators?

fail-fast Java iterators may throw `ConcurrentModifcationException` if the underlying collection is modified during an iteration is in progress. fail-safe iterators will not throw any exception as the iteration happens on a clone of the instance. fail fast and fail safe are paradigms that define how a system react when it encounters failure condition. Example for fail fast iterator is `ArrayList` and for fail safe iterator is `ConcurrentHashMap`.

3. What is `BlockingQueue` in Java?

Java BlockingQueue is a concurrent collection that is part of the util package. `BlockingQueue` is a type of queue which supports operations that wait for an element to become available when retrieving from it and similarly wait for a space to become available when storing elements in it. This collection is best used in a producer consumer scenario.

4. When do you use `ConcurrentHashMap`?

In the above interview question number 2 we saw `ConcurrentHashMap` as an example for fail safe iterator. It allows complete concurrency for retrievals and updates. When there is a scenario where a high number of concurrent updates are expected then ConcurrentHashMap can be used. This is very similar to a `Hashtable` but does not lock the entire table to provide concurrency and so it is better performance point of view. When there are high number of updates and less number of read concurrently, then `ConcurrentHashMap` should be used.

5. Which `List` implementation provides fastest insertion?

This is between `LinkedList` and `ArrayList`. These two are different variants of List implementations. LinkedList is a doubly linked list datastructure and ArrayList is dynamically resizing array. Performance of these two collections with respect to insert is `O(n)` for LinkedList and `O(n-index)` for ArrayList.

In LinkedList the cost is always a constant factor, it is about allocating a node and linking with adjacent elements. In `ArrayList` the cost varies based on whether the insertion is at beginning or at the end of the list. Other elements already existing in the ArrayList should be adjusted for positions according to the insertion.

If the insertion is at the end of the `List` then ArrayList is faster than LinkedList. If the insertion is at the beginning and also if the list is longer then `LinkedList` wins.

6. Difference between Iterator and ListIterator

`ListIterator` is used to traverse List type of collections exclusively where in `Iterator` can be used traverse any type of collections. ListIterator has got additional features over an Iterator and they are,

- ListIterator can traverse a List backwards where in Iterator cannot do.
- Using ListIterator an element can be added at any given point.
- Get the current index at any traversal moment.
- Replace an element at the traversal point.

7. What is CopyOnWriteArrayList, how it is different than ArrayList?

`CopyOnWriteArrayList` is a thread-safe counterpart of ArrayList. All mutable operations like add and set are implement by using a copy of the underlying array. Write operation is slower when compared the `ArrayList` as it takes a snapshot of the instance and does the write. This is useful when the traversal of the collection need not be synchronized during traversal with the original instance and the traversal is larger in count than the updates. This provides a fail safe iterator as it does not throw exception when the underlying collection is modified. At the same time the iterator will not reflect the modifications done to the collection, it just shows the snapshot of state taken at the moment when the iterator is created.

8. Difference between Iterator and Enumeration

If the interviewer asks this question in a Java interview any more, consider him to be legacy. Mainly `Iterator` is different from `Enumeration` in two ways,

- Iterator allows the removal of elements from the underlying collection.
- Method names are standardized in Iterator.

`Iterator` is brought in as a replacement for Enumeration in Java 1.2 release. Use `Iterator` everywhere instead of `Enumeration`

9. How a HashMap can be synchronized?

There are two options when we need a synchronized `HashMap`.

- Use `Collections.synchronizedMap(..)` to synchronize the `HashMap`.
- Use `ConcurrentHashMap`.

The preferred choice between these two options is to use the `ConcurrentHashMap`. That's because we need not lock the whole object and ConcurrentHashMap partitions the map and obtains lock as necessary. Read the interview question number 4 above. Need not reinvent the wheel unless you are going to provide a different and greater implementation than the ConcurrentHashMap.

10. Difference between IdentityHashMap and HashMap

`IdentityHashMap` is an implementation of `Map` interface. Unlike `HashMap`, this uses reference equality. That means,

- in HashMap, two elements are equal if `key1.equals(key2)`
- iin IdentityHashMap, two elements are equal if `key1 == key2`

Map imposes a contract to honor, the implementations should use object equality. If equals of method returns same value for two keys, then the hash code value should be same.

- IdentityHashMap intentionally violates the contract and does reference equality.
- For hashing, the IdentityHashMap uses `System.identityHashCode(object)` instead of `hashCode()` as done by HashMap.
- IdentityHashMap is relatively faster than HashMap for operations.
- Keys are mutable in `IdentityHashMap` but in HashMap keys are mutable.