***1) What is static binding?***

Static **binding in Java** occurs during Compilion time while Dynamic **binding** occurs during Runtime. Private, final and static **methods** and variables uses static **binding.**

***2) What is the difference between static and dynamic binding?***

In the most general terms, **static binding** also called early binding, means that references are resolved at compile time. **Animal a = new Animal();** **a.Roar();** // The compiler can resolve this method call statically.

**Dynamic binding** means that references are resolved at run time. public void MakeSomeNoise(object a) { // Things happen...

**Dynamic binding** also called **dynamic** late binding, is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time. **Dynamic binding** is also known as late **binding** or run-time **binding**.

***3) What is constructor chaining in Java?***

Calling one **constructor** from other is called **Constructor chaining** in Java.**Constructors** can call each other automatically or explicitly using **this()** and **super()** keywords. **this()** denotes a no-argument **constructor** of the same class and **super()** denotes a no-argument or default **constructor** of parent class.

**4) Upcasting, downcasting**

When instance of Subclass type refers to the object or variable of (SuperClass)Parent class, it is known as **Upcasting**. If an instance of a Superclass refer to a variable or object of the Subclass, it is know as Downcasting, if it is directly performed a compiler gives Compilation error ClassCastException is thrown at runtime.

class Parent{ /\* ... \*/}

class ChildClass extends Parent { /\* ... \*/ }

public class Tester {

public static void main (String args[ ]) {

Parent p = new Parent ( );

/\*this is a downcast since the Parent class

object, "p" is being cast to a ChildClass type,

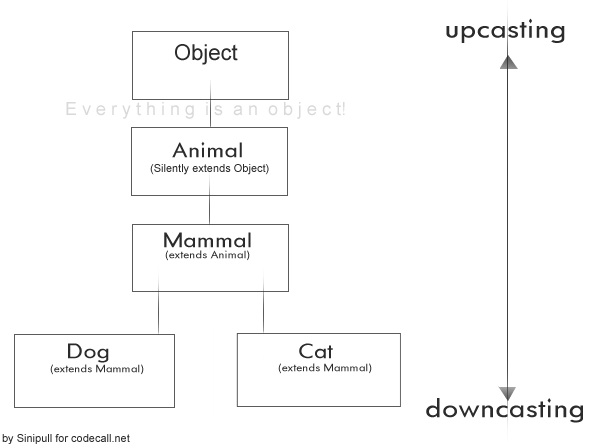
and ChildClass derives from the Parent class \*/

ChildClass c = (ChildClass) p;

}

}

**5) Upcasting, downcasting**

Upcasting and downcasting are important part of Java, which allow us to build complicated programs using simple syntax, and gives us great advantages, like Polymorphism or grouping different objects. Java permits an object of a subclass type to be treated as an object of any superclass type. This is called upcasting. Upcasting is done automatically, while downcasting must be manually done by the programmer, and i'm going to give my best to explain why is that so.  
  
Upcasting and downcasting are **NOT** like casting primitives from one to other, and i believe that's what causes a lot of confusion, when programmer starts to learn casting objects.  
  
Throughout this tutorial i'm going to use Animal hierarchy to explain how class hierarchy works.  
  
**Inheritance**  
  
  
  
What we have here, is a simplified version of an Animal Hierarchy. You can see, that Cat and Dog are both Mammals, which extends from Animal, which silently extends from Object. By silently, i mean, that Java automatically extends every class from Object class, which isn't extended from something else, so everything is an Object (except primitives).  
  
Now, if you ask - is Cat an Object - It doesn't extend Object, it extends Mammal?   
By inheritance Cat gets all the properties its ancestors have. Object is Cat's grandgrandparent, which means Cat is also an Object. Cat is also an Animal and a Mammal, which logically means - if Mammals possess mammary glands and Animals are living beings, then Cat also has mammary glands and is living being.  
  
What this means for a programmer, is that we don't need to write for every possible Animal, that it has health. We just need to write it once, and every Animal gets it through inheritance.  
Consider the following example:

Code:

class Animal {

int health = 100;

}

class Mammal extends Animal { }

class Cat extends Mammal { }

class Dog extends Mammal { }

public class Test {

public static void main(String[] args) {

Cat c = new Cat();

System.out.println(c.health);

Dog d = new Dog();

System.out.println(d.health);

}

}

When running the Test class, it will print "100" and "100" to the console, because both, Cat and Dog inherited the "health" from Animal class.  
  
**Upcasting and downcasting**  
  
First, you must understand, that by casting you are not actually changing the object itself, you are just labeling it differently.   
For example, if you create a Cat and upcast it to Animal, then the object doesn't stop from being a Cat. It's still a Cat, but it's just treated as any other Animal and it's Cat properties are hidden until it's downcasted to a Cat again.   
Let's look at object's code before and after upcasting:

Code:

Cat c = new Cat();

System.out.println(c);

Mammal m = c; // upcasting

System.out.println(m);

/\*

This printed:

Cat@a90653

Cat@a90653

\*/

As you can see, Cat is still exactly the same Cat after upcasting, it didn't change to a Mammal, it's just being labeled Mammal right now. This is allowed, because Cat is a Mammal.  
  
Note that, even though they are both Mammals, Cat cannot be cast to a Dog. Following picture might make it a bit more clear.