Interview Theory Questions

1. **What is encapsulation in java?**

Encapsulation in Java is the mechanism of hiding the internal implementation details of an object from the outside world and restricting access to it through a well-defined interface. In Java, encapsulation is achieved through the use of access modifiers such as public, private, and protected.

Public methods and variables are accessible from outside the class, while private methods and variables are accessible only within the class. Protected methods and variables are accessible within the class and its subclasses.

By encapsulating data and methods, the internal details of an object can be protected from external interference and misuse. This helps to ensure that the object's state remains consistent and valid.

Encapsulation is an important concept in object-oriented programming as it helps to maintain the integrity of the object's internal state and ensures that changes to the object are made in a controlled and predictable manner.

2. **What is inheritance in java?**

Inheritance in Java is a mechanism that allows a class to inherit properties and methods from another class. In other words, it is the process by which one class acquires the properties and behaviors of another class.

The class that is being inherited from is called the superclass or parent class, and the class that is inheriting from the superclass is called the subclass or child class.

To inherit from a superclass, a subclass uses the keyword extends followed by the name of the superclass. The subclass can then use the properties and methods of the superclass as if they were its own, and it can also add its own properties and methods.

Inheritance is a powerful mechanism in Java as it allows for code reuse and helps to establish a hierarchical relationship between classes. It also allows for the creation of more specialized classes that are based on more general classes, which can help to simplify code and improve maintainability.

However, it's important to note that inheritance can also lead to complex and tightly-coupled code if not used appropriately, so it should be used judiciously and with care.

3. **What is polymorphism in java?**

Polymorphism in Java is the ability of an object to take on many forms. In other words, it allows objects of different classes to be treated as if they are objects of the same class.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

Compile-time polymorphism is achieved through method overloading, which is the process of defining multiple methods in a class with the same name but different parameter types. The compiler determines which method to call based on the number and types of arguments passed to it.
Runtime polymorphism, on the other hand, is achieved through method overriding, which is the process of providing a new implementation for an existing method in a subclass. When an object of the subclass is created, the method in the subclass is called instead of the method in the superclass, and this behavior is determined at runtime.
Polymorphism is an important concept in object-oriented programming as it allows for more flexible and extensible code. It allows objects to be used in a variety of different contexts, which can simplify code and make it more reusable.

4. **What is reference variable casting in java?**

In Java, reference variable casting is the process of converting a reference of one type to a reference of another type. It is often used in situations where a variable of one type needs to be treated as if it were a variable of another type.
There are two types of casting in Java: upcasting and downcasting.
Upcasting is the process of casting a subclass reference to a superclass reference. Since a subclass inherits all the properties and methods of its superclass, it can be treated as if it were an instance of its superclass. Upcasting is performed automatically by the compiler and does not require an explicit cast operator.
For example:

```
class Animal{}

class Dog extends Animal{}

Animal a= new Dog() //upcasting ();
```

Downcasting, on the other hand, is the process of casting a superclass reference to a subclass reference. Since a superclass does not inherit all the properties and methods of its subclass, downcasting requires an explicit cast operator and may result in a ClassCastException if the reference is not actually pointing to an instance of the subclass.

For example:

```
class Animal{}
class Dog extends Animal{}
```

```
Animal a=new Dog()//Upcasting
Dog d=  (Dog)a;//Downcasting
```

It's important to note that downcasting should only be used when the actual runtime type of the object is known to be a subclass. Otherwise, it may result in a runtime error.

5. **What is Method overriding in java?**

Method overriding in Java is the process of providing a new implementation for an existing method in a subclass that is already defined in its superclass. The method signature in the subclass must match the method signature in the superclass, including the method name, return type, and parameter types.
When an object of the subclass is created and a method is called on it, the JVM looks for the method in the subclass first. If the method is found in the subclass, it is called. If the method is not found in the subclass, the JVM looks for the method in the superclass. Overriding a method in a subclass allows the subclass to provide its own implementation of the method and customize the behavior of the method for the subclass. This can be useful for implementing specialized behavior that is specific to the subclass.

Here's an example:

```
class Animal {
	public void makeSound() {
		System.out.println("Animal is making a sound");
	}
}
class Cat extends Animal {
	@Override public void makeSound()
	{
		System.out.println("Meow");
	}
}
```

In this example, the Cat class overrides the makeSound method of the Animal class to provide its own implementation. When the makeSound method is called on a Cat object, the Cat implementation of the method will be called instead of the Animal implementation.
Note that the @Override annotation is used to indicate that the method is intended to override a method in the superclass. This annotation is optional but recommended as it

can help to catch errors at compile-time if the method signature does not match the signature of a method in the superclass.