**Abstraction:**

Abstraction in Java is the process of hiding the complexity of an object or system and presenting only the essential features to the user. Abstraction allows you to focus on the functionality of an object without worrying about its internal details.

In Java, abstraction can be achieved through abstract classes and interfaces. Here are examples of both:

1. Abstract Classes: An abstract class is a class that cannot be instantiated on its own, but can be extended by other classes. Abstract classes can contain both abstract and non-abstract methods. Abstract methods are methods that do not have an implementation in the abstract class and must be implemented by the subclass.

**Encapsulation:**

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) and refers to the mechanism of hiding the implementation details of a class from other classes. In Java, encapsulation is achieved by using access modifiers, such as private, public, and protected, to restrict the access of class members (fields, methods, and nested classes) from the outside world.

By encapsulating the implementation details, we can ensure that the internal state of an object is only modified through a set of well-defined methods (known as getters and setters) and that the object's behavior is consistent with its intended use. This helps to prevent unintended modifications to an object's state, which can lead to unexpected behavior and bugs in a program.

For example, consider a class called **Person** with private fields for the person's name, age, and address. We can use public getter and setter methods to allow other classes to access and modify these fields in a controlled manner, while keeping the implementation details of the Person class hidden:

public class Person {

private String name;

private int age;

private String address;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

**Difference between Encapsulation and Abstraction:**

Encapsulation and abstraction are two fundamental concepts in object-oriented programming. They are closely related but have different meanings.

1. Encapsulation:

Encapsulation is a mechanism that restricts direct access to an object's internal state or data from outside the object's scope. It means that the object's data and methods are kept hidden and can only be accessed through the object's public interface. In other words, encapsulation is about hiding the implementation details of an object and exposing only the essential information to the outside world. This helps to prevent unauthorized access to the object's data and ensures that the object's behavior is consistent.

For example, a bank account object might have private data fields such as account number, balance, and account holder name. The methods of the bank account object, such as deposit, withdrawal, and balance inquiry, can be used to access and modify this data. However, the data fields cannot be accessed directly from outside the object.

1. Abstraction:

Abstraction is a mechanism that focuses on the essential features of an object and hides the details that are not necessary for its intended use. It is the process of identifying and separating the essential characteristics of an object from the irrelevant details. Abstraction helps to simplify the complexity of an object by providing a high-level view of its behavior.

For example, a car object can be abstracted as a machine that has a steering wheel, brake, accelerator, and gears. The internal details of the engine, transmission, and suspension are not relevant for its use as a means of transportation.

In summary, encapsulation is about hiding the implementation details of an object and protecting its internal state, while abstraction is about focusing on the essential features of an object and hiding the irrelevant details.

**Polymorphism:**

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they were objects of a common superclass. In Java, polymorphism can be achieved through inheritance, interfaces, and method overloading.

1. Inheritance: In Java, a subclass can inherit methods and properties from its superclass. Polymorphism occurs when a subclass overrides a method in its superclass and provides its own implementation. This allows objects of the subclass to be treated as objects of the superclass, but with the added functionality of the subclass's implementation.
2. Interfaces: Interfaces in Java define a set of methods that a class must implement. Polymorphism can be achieved through interfaces when multiple classes implement the same interface. This allows objects of different classes to be treated as if they were objects of the same interface type, allowing for greater flexibility in program design.
3. Method Overloading: Method overloading occurs when a class has multiple methods with the same name but different parameters. Polymorphism occurs when a method is called on an object, and the appropriate method is chosen based on the parameters passed in. This allows for more flexible method calls and easier code reuse.

Polymorphism is an important concept in Java programming as it allows for greater flexibility and reusability of code. By using polymorphism, programmers can write code that is more easily extensible, maintainable, and adaptable to changes.

1. **Single Inheritance:** In single inheritance, a subclass inherits properties and behaviors from a single superclass. This is the most common type of inheritance used in Java and is implemented using the **extends** keyword.
2. **Multilevel Inheritance:** In multilevel inheritance, a subclass inherits properties and behaviors from a superclass, which in turn inherits properties and behaviors from another superclass. This creates a hierarchy of classes, with each subclass adding functionality to the previous one.
3. **Multiple Inheritance:** In multiple inheritance, a subclass inherits properties and behaviors from multiple superclasses. However, Java does not support multiple inheritance with classes. Instead, Java supports multiple inheritance through interfaces, where a class can implement multiple interfaces and inherit the properties and behaviors defined in each interface.