Class # 8 & 9

1. Encapsulation

- Encapsulation Concept
- Private Variables and Public Methods
- Getters and Setters

Code Example:

java code

```
public class Person {
private String name;
public String getName() {
 return name; }
public void setName(String n) {
 name = n; }
 }
```

Encapsulation:

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts, and it's a key principle in Java. It refers to the bundling of data (attributes or fields) and methods (functions) that operate on that data into a single unit known as a class. Encapsulation restricts direct access to some of an object's components and prevents unintended interference and misuse, ensuring that an object's internal state remains consistent and valid.

Here's how encapsulation works in Java:

1. **Private Fields**: The attributes (fields) of a class are often declared as private. This means they cannot be accessed or modified directly from outside the class. They are encapsulated within the class.

Code:

```
public class MyClass {
 private int myPrivateField;
}
```

2. **Public Methods**: To allow controlled access to the private fields, you provide public methods (getters and setters) within the class. These methods are the interface through which external code interacts with the class.

Java code

```
public class MyClass {
private int myPrivateField;
public int getMyPrivateField() {
return myPrivateField; }
public void setMyPrivateField(int value) {
myPrivateField = value;
}
}
```

3. **Getter and Setter Methods**: Getter methods retrieve the value of a private field, while setter methods modify it. They provide controlled access to the encapsulated data.
4. **Data Validation**: Within setter methods, you can include validation logic to ensure that the data remains within acceptable ranges. This helps maintain data integrity.
5. **Benefits of Encapsulation**:
    - **Control**: You can control what is allowed to be modified and how it is modified, reducing the risk of unexpected changes or errors.

- **Flexibility**: You can change the internal implementation details of a class (such as its fields) without affecting the external code that uses the class. This is known as information hiding.
- **Reusability**: Encapsulated classes are easier to reuse because their interface is well-defined and self-contained.

Here's an example illustrating encapsulation in Java:

Java code

```
public class BankAccount {
private double balance;
public BankAccount(double initialBalance) {
if (initialBalance >= 0) {
balance = initialBalance;
}
else
{ balance = 0; }
}
public double getBalance() {
return balance;
}
public void deposit(double amount) {
if (amount > 0) {
balance += amount;
}
}
public void withdraw(double amount) {
```

```
if (amount > 0 && amount <= balance)
{
 balance -= amount;
}
}
}
```

In this example, the **balance** field is private, and the **getBalance**, **deposit**, and **withdraw** methods provide controlled access to it. Encapsulation ensures that the **balance** remains valid and prevents unauthorized modifications.

************Access Modifiers**

Access modifiers, also known as access specifiers, are keywords in Java that define the visibility or accessibility of classes, variables, methods, and constructors. They determine which parts of your code can access or modify a particular element. Java provides four main access modifiers:

public: The element (class, variable, method, etc.) is accessible from any other class.

private: The element is only accessible within the same class. It cannot be accessed from outside the class.

protected: The element is accessible within the same class, within subclasses, and within the same package.

default (package-private): If no access modifier is specified, the element is accessible within the same class and within the same package (package-level access).

Here's an example demonstrating the use of access modifiers:

```
public class MyClass {
    public int publicVar;       // Accessible from anywhere
    private int privateVar;     // Accessible only within this class
    protected int protectedVar; // Accessible within this class and subclasses
    int defaultVar;             // default Access within this class and package
}
```

Access modifiers help control the level of encapsulation and visibility of class members, contributing to the principles of encapsulation and data hiding in Object-Oriented Programming.

2. Inheritance

- **Inheritance Concept:**

Inheritance is one of the fundamental concepts of object-oriented programming (OOP), and it plays a crucial role in Java. It allows you to create a new class (subclass or derived class) that inherits attributes and behaviors (fields and methods) from an existing class (superclass or base class). Inheritance forms an "is-a" relationship between classes, where a subclass is a specialized version of its superclass.

- **Superclass and Subclass**

Here are the key aspects of inheritance in Java:

Superclass and Subclass: Inheritance involves two classes, the superclass (parent class) and the subclass (child class). The superclass contains the common attributes and methods shared by multiple subclasses.

Extending a Class: To create a subclass, you use the extends keyword in the class declaration, followed by the name of the superclass. For example:

```
class Subclass extends Superclass {
    // Subclass-specific members
}
```

Inherited Members: The subclass inherits all non-private members (fields and methods) of the superclass. It can access these members directly as if they were defined in the subclass.

Method Overriding: In Java, you can override (redefine) methods inherited from the superclass in the subclass. This allows you to provide a specialized implementation of a method in the subclass. To override a method, you use the @Override annotation before the method definition in the subclass.

Access Modifiers: Inherited members' accessibility depends on their access modifiers. Public members are accessible from anywhere, protected members are accessible within the same package and subclasses, and package-private (default) members are accessible only within the same package.

Constructor Chaining: Constructors are not inherited, but a subclass constructor can call a constructor of its superclass using the super keyword. This is known as constructor chaining.

- **Method Overriding**

Code Example:

Here's a simple example to illustrate inheritance:

```
class Animal {
String name;
Animal(String name) {
this.name = name; }
void eat() {
System.out.println(name + " is eating.");
}
}
class Dog extends Animal {
Dog(String name) {
 super(name); // Call superclass constructor
}
void bark() {
System.out.println(name + " is barking.");
}
}

public class Main {
public static void main(String[] args) {
 Dog myDog = new Dog("Buddy");
myDog.eat(); // Inherited from Animal class
myDog.bark(); // Specific to Dog class
 }
}
```

In this example, the **Dog** class extends the **Animal** class, inheriting the **name** field and the **eat** method. It also defines its own method, **bark**. The **super** keyword is used to call the constructor of the superclass. When we create a **Dog** object, it can access both inherited and specific members.