Constructors:

A constructor is a special method that gets called when an object is created from a class. It's used to initialize the object's state or perform any necessary setup. If you don't explicitly define a constructor, Java provides a default constructor with no arguments.

```java
public class Car {
    String make;
    String model;
    int year;

    // Constructor with parameters
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // Method to display car information
    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}
```

In the example above, we've added a constructor to the `Car` class that accepts parameters for initializing the object's fields.

Access Modifiers:

Java provides access modifiers to control the visibility and accessibility of class members (fields, methods, and nested classes). The main access modifiers are `public`, `private`, `protected`, and default (package-private).

```java
public class Car {
    public String make;     // Accessible from anywhere
    private String model;   // Accessible only within the class
    protected int year;     // Accessible within the class and subclasses
    String color;           // Default access (package-private)
```

```
}
```

Getter and Setter Methods:

Getter and setter methods are used to access and modify private fields of a class. They follow naming conventions: `get` for getters and `set` for setters, followed by the field name with the first letter capitalized.

```java
public class Car {
    private String make;
    private String model;

    // Getter for make
    public String getMake() {
        return make;
    }

    // Setter for make
    public void setMake(String make) {
        this.make = make;
    }
}
```

Static Members:

Static members (fields and methods) belong to the class itself rather than to individual objects. They can be accessed using the class name, even without creating an object.

```java
public class Car {
    static int totalCars;  // Static field

    public Car() {
        totalCars++;
    }

    public static void displayTotalCars() {
        System.out.println("Total cars: " + totalCars);
    }
}
```

Inheritance:

Inheritance allows you to create a new class that's based on an existing class. The new class inherits fields and methods from the existing class and can also have its own additional members.

```java
public class ElectricCar extends Car {
    int batteryCapacity;

    public ElectricCar(String make, String model, int year, int batteryCapacity) {
        super(make, model, year);
        this.batteryCapacity = batteryCapacity;
    }

    public void displayBatteryCapacity() {
        System.out.println("Battery Capacity: " + batteryCapacity + " kWh");
    }
}
```

In this example, `ElectricCar` inherits fields and methods from the `Car` class and adds its own `batteryCapacity` field and `displayBatteryCapacity()` method.

Polymorphism:

Polymorphism allows you to use objects of different classes through a common interface. It includes method overriding (subclass providing a specific implementation of a method) and method overloading (multiple methods with the same name but different parameters).

```java
public class Main {
    public static void main(String[] args) {
        Car car = new ElectricCar("Tesla", "Model S", 2023, 100);
        car.displayInfo();              // Calls displayInfo from ElectricCar
        ((ElectricCar) car).displayBatteryCapacity();  // Calls displayBatteryCapacity from ElectricCar
    }
}
```

In this example, we're using polymorphism to create an `ElectricCar` object and then calling methods from both the parent `Car` class and the child `ElectricCar` class.

These advanced concepts build on the foundation of classes and objects in Java, enabling you to create more sophisticated and flexible programs.