**Chapter # 02: Object Oreinted Design with Java**

**2.1 Classes:**

Classes are a fundamental concept in object-oriented programming (OOP) and provide a way to encapsulate related data and behavior into a single unit. A class is essentially a blueprint or template for creating objects, which are instances of the class.

In Java, classes are used to define objects, which contain both data (known as instance variables or fields) and behavior (known as methods). These objects can be used to model real-world entities or concepts, such as a person, a bank account, or a car.

Java classes have the following primary advantages:

1. **Abstraction:** Classes provide you the ability to mask the complexity of the implementation that lies beneath and offer a streamlined user interface for interacting with objects. This makes using and understanding the code simpler.
2. **Encapsulation:** By enclosing an object's data and behavior in a class, you can shield it from outside disturbance and make it simpler to manage and update.
3. **Reusablity:** Reusable classes make it simpler to develop modular, maintainable code because they can be used in different sections of one program or in separate applications altogether.

**2.2 Class Creation:**

To create a class in Java, you use the **class** keyword followed by the name of the class, which should be in CamelCase format (e.g. Person, BankAccount). Here's an example of a simple class in Java:

|  |
| --- |
| Sample Person Class |
| public class Person {      // instance variables      private String name;      private int age;      // constructor      public Person(String name, int age) {          this.name = name;          this.age = age;      }      // methods      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;      }  } |

In this example above, the Person class has two instance variables (name and age) and three methods (getName(), setName(), and getAge(), setAge()). The constructor method is a special method that is called when a new instance of the class is created, and is used to initialize the instance variables.

**2.3 Attributes and Methods in class**

In Java, methods and attributes are the two main components of a class that encapsulate the behavior and state of objects, respectively.

Attributes (also known as instance variables or fields) represent the state of an object, and define the properties or data that are associated with the object. These can be any data type, such as primitive types (int, boolean, etc.) or complex types (other objects or arrays). Attributes are declared inside a class, and can be accessed using an object of the class. For example, here is a simple class with two attributes:

|  |
| --- |
| Attribute Sample in Java |
| public class Person {      private String name;      private int age;  } |

In this example, the Person class has two attributes: name, which is a String, and age, which is an int. These attributes are declared as private, which means they can only be accessed from within the class itself. We can also create public attributes which can accessed from outside the class. To Access the private members of the class we have to create Setter and Getters. Which are just methods that return or set the value of an attribute pass by the user through the arguements. We will disscuss setter and getters after methods.

Methods, on the other hand, represent the behavior of an object, and define the actions that can be performed on the object. Methods are declared inside a class, and can be called using an object of the class. Methods can also take parameters (input) and return values (output). For example, here is a simple class with two methods:

|  |
| --- |
| Sample Methods Class |
| public class Person {      private String name;      private int age;      public void sayHello() {          System.out.println("Hello, my name is " + name);      }      public void setAge(int newAge) {          age = newAge;      }  } |

In this example, the Person class has two methods: sayHello(), which prints a greeting to the console, and setAge(), which sets the value of the age attribute to a new value. The sayHello() method does not take any parameters and does not return any value, while the setAge() method takes an integer parameter and does not return any value.

Just like attributes, methods can also be private or public. Which means we can set which method we want to allow access outside the class.

**2.4 Setter and Getters:**

Getters and setters are unique Java methods that are used to read and change the values of private attributes (instance variables) of a class. These methods are sometimes referred to as accessor and mutator methods.

Getters are methods that let you extract an attribute's value from an object. The pattern get<AttributeName>, where AttributeName is the name of the attribute, is generally used to name them. Here is an illustration of a getter method for a Person class's name attribute:

|  |
| --- |
| Sample Getters Class |
| public class Person {      private String name;      public String getName() {          return name;      }  } |

Setters, on the other hand, are methods that allow you to modify the value of an attribute in an object. They are typically named using the pattern set<AttributeName>, where AttributeName is the name of the attribute. Here's an example of a setter method for the age attribute of a Person class:

|  |
| --- |
| Sample Setters Class |
| public class Person {      private int age;      public void setAge(int newAge) {          age = newAge;      }  } |

Using getters and setters is considered good practice in Java, because it provides a way to control access to the attributes of a class and enforce data encapsulation. By making the attributes private and providing getter and setter methods, you can ensure that the state of an object can only be modified in a controlled way, which can help to prevent bugs and ensure the correctness of your code.

**2.5 This Keyword:**

In Java, **this** is a special keyword that refers to the current object instance of the class in which it is used. The this keyword can be used in several ways:

1. **To refer to the current object's attributes:** For example, if a method has a parameter with the same name as an attribute of the class, you can use this to refer to the attribute instead of the parameter:

|  |
| --- |
| public class Person {      private String name;      public void setName(String name) {          this.name = name;      }  } |

1. **To call a constructor from another constructor:** For example, if a class has multiple constructors, you can use this to call one constructor from another:

|  |
| --- |
| public class Person {      private String name;      private int age;      public Person(String name, int age) {          this.name = name;          this.age = age;      }      public Person(String name) {          this(name, 0); // calls the other constructor with default age value      }  } |

1. **To return the current object from a method:** For example, if a method needs to return the current object, you can use this to return a reference to the current object:

|  |
| --- |
| public class Person {      private String name;      public Person(String name) {          this.name = name;      }      public Person self() {          return this;      }  } |

The **this** keyword in Java is a way to refer to the current object instance of a class, and can be used to disambiguate between variables with the same name, call a constructor from another constructor, or return a reference to the current object.

**2.6 Constructor Overloading:**

The ability to declare numerous constructors with the same name but different parameters is known as constructor overloading in Java. Programmers can create objects using various initialization values depending on the particular requirements of the program thanks to constructor overloading.

A Java class is considered to have overloaded constructors if it defines numerous constructors with the same name but distinct argument lists. Based on the quantity, arrangement, and types of the parameters that each of these constructors accepts, the Java compiler distinguishes between them.

Take the Java class with two constructors as an illustration:

|  |
| --- |
| Contructor Overloading Class |
| public class Person {      private String name;      private int age;      public Person() {          name = "";          age = 0;      }      public Person(String name, int age) {          this.name = name;          this.age = age;      }      // other methods...  } |

The Java compiler determines which constructor to use based on the number and types of arguments passed to the constructor. If no arguments are passed, the first constructor is called. If two arguments of the correct types are passed, the second constructor is called.

Constructor overloading is a powerful feature of Java that allows programmers to create classes with flexible and customizable initialization options. By providing multiple constructors with different parameter lists, programmers can make their classes more versatile and easier to use.