1. What are the 4 main pillars of Object Oriented Programing and give a description of each and how they are applied.

- Abstraction:

  This pillar hide out the working style of an object showing only the required information of the object in understandable manner and it solves the problem in design level, for use it we call is “implement” not “extend”.

Example:

```
public Interface Animals
{
  Void speak();
}


In this next two class we are using abstraction
Public class Cat extend Animals
{
        @Override
        Public void speak()
        {
                System.out.println(“Miauuu”);
        }
}

Public class Dog extend Animals
{
        @Override
        Public void speak()
        {
                System.out.println(“Wuau Wuau”);
        }
}
```

- <u>Encapsulation</u>:

  It's the pillar in OOP who allow using a complicated class without thinking about the logic inside, encapsulation use "private" for all the information or data that it wants to hide from the class who are going to call it, and it'll use "public" and maybe "protected" in the function than the user will need to use. One case of encapsulation in the real life can be when you wake up and going to the coffee machine to make your coffee, you just need to know where you need to put the water and the coffee, you don't need to be worried about all the process inside the coffee machine.

Example:

```
Public class Vehicle
{
        Private String color;
        Private String brand;
        Private int wheel;

        Public Vehicle(String color, String brand, int wheel)
        {
                this.color   = color;
                this.brand = brand;
                This.wheel = wheel;
        }

        Public void showColor()
        {
                System.out.println("The color of this vehicle is "+color);
        }
}
```

```
public static void main(String[] args)
{
        Vehicle car = new Vehicle(“Red”,”Toyota”,4);
        car.showColor();
}
```

And because we used Encapsulation with just call car.showColor(); it’ll show to the user the color of the car.

- Polymorphism:

  It’s the ability to redefine methods for derived classes, a subclass can have their own behavior and share some behavior from it’s parent class not vice versa. A parent class cannot have the behavior of it’s subclass. Polymorphism can be Overloading and Override, and example in real life can be Twin brothers looks alike but they hold different characters.

Example:

```
Public void printNumber()
{
        System.out.println(“I don’t have anything”);
}

Public void printNumber(int num)
{
        System.out.println(“I have the number “+num);
}
```

They have the same name but they hold different text to show

- Inheritance:

  It’s the ability of creating a new class from an existing class that mean an object acquires the properties of another object. It helps to reuse, customize and enhance the existing code. So it helps to write a code accurately and reduce the development time.

Example:

```
Public class Teacher
{
        String designation = "Teacher";
        String collegeName = "Beginnersbook";
        Public void does()
        {
                System.out.println("Teaching");
        }
}
```

```
public class PhysicsTeacher extends Teacher
{
        String mainSubject = "Physics";
        public static void main(String args[])
        {
                PhysicsTeacher obj = new PhysicsTeacher();
                System.out.println(obj.collegeName);
                System.out.println(obj.designation);
                System.out.println(obj.mainSubject);
                obj.does();
        }
}
```

2. What are SOLID programming principles and what does each section detail?

It’s an acronym for the first five object-oriented design, when we combined together make it easy for a programmer to develop software that are easy to maintain and extend and are also a part of the agile or adaptive software development.

- **S** - Single-responsiblity principle
- **O** - Open-closed principle
- **L** - Liskov substitution principle
- **I** - Interface segregation principle
- **D** - Dependency Inversion Principle

- ### **S** - Single-responsiblity principle(S.R.P):

  A class should have one and only one reason to change, meaning that a class should have only one job.

- ### **O** - Open-closed principle (O.C.P):

  Objects or entities should be open for extension, but closed for modification.

- ### **L** - Liskov substitution principle:

  Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

- ### **I** - Interface segregation principle (I.S.P):

  A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

- **D** - Dependency Inversion Principle :

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

3. What are the differences of the Following:

- HashMap vs HashTable

It is non synchronized	It is synchronized
It is not-thread safe and can't be shared between many threads without proper synchronization	It is thread-safe and can be shared with many threads.
It is generally preferred over HashTable if thread synchronization is not needed	It isn't generally preferred over HashMap if thread synchronization is not needed
It allows one null key and multiple null values	It doesn't allow any null key or value

- ArrayList vs List

It's a class which extends Abstract List and it implements the List interface	It's an interface which extends Collection. As it extends Collection it declares it's behavior and stores a sequence of elements
(Syntax) Public class ArrayList<E> extends AbstractList<E> implements List<E>,RandomAccess,Cloneable,Serializable	(Syntax) Public interface List<E> extends Collection<E>
(Working)	(Working)

ArrayList<Type> str = new ArrayList<Type>();	List a = new ArrayList()

- Array vs ArrayList

array is simple fixed sized array	ArrayList is dynamic sized array
Array can contain both primitives and objects	ArrayList can contain only object elements
You can't use generics along with array	ArrayList allows us to use generics to ensure type safety.
You can use *length* variable to calculate the length of an array	*You need to use size()* method to calculate size of ArrayList.
Array use assignment operator to store elements	ArrayList use *add()* to insert elements.

- HashSet vs HashMap

HashSet class implements the Set interface	HashMap class implements the Map interface
In HashSet, we store objects(elements or values) e.g. If we have a HashSet of string elements then it could depict a set of HashSet elements: {“Hello”, “Hi”, “Bye”, “Run”}	HashMap is used for storing key & value pairs. In short, it maintains the mapping of key & value (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This is how you could represent HashMap elements if it has integer key and value of String type: e.g. {1->”Hello”, 2->”Hi”, 3->”Bye”, 4->”Run”}
HashSet does not allow duplicate elements that mean you can not store duplicate values in HashSet.	HashMap does not allow duplicate keys however it allows having duplicate values.
HashSet permits to have a single null value.	HashMap permits single null key and any number of null values.

Insertion method is Add()	Insertion method is Put()

- StringBuilder vs StringBuffer

StringBuilder is not thread safe and synchronized	StringBuffer is thread safe and synchronized
StringBuilder is more faster than StringBuffer.	StringBuffer is more slower than StringBuilder

4. Why is it important to override the equals and hashCode methods for Java objects?

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable. By defining equals() and hashCode() consistently, you can improve the usability of your classes as keys in hash-based collections. As the API doc for hashCode explains: "This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable."

5. What is the difference in an Abstract Class and Interface?

Methods of a Java abstract class can have instance methods that implements a default behavior	Methods of a Java interface are implicitly abstract and cannot have implementations
Variables declared in a Java abstract class may contain non-final variables	Variables declared in a Java interface is by default final

A Java abstract class can have the usual flavors of class members like private, protected, etc..	Members of a Java interface are public by default
Java abstract class should be extended using keyword “extends”	Java interface should be implemented using keyword “implements”
Abstract class can extend another Java class and implement multiple Java interfaces	Interface can extend another Java interface only
Java class can extend only one abstract class	Java class can implement multiple interfaces
Java abstract class cannot be instantiated, but can be invoked if a main() exists	Interface is absolutely abstract and cannot be instantiated
	Java interfaces are slow as it requires extra indirection