Constructors:

1. Constructor allow programmers to control how the attributes of an object are defined.
2. A constructor is usually defined by a public accessor type and whose name is identical to the class name.
3. It does not have a return type.
4. Constructors are not found in procedural languages such as C, COBOL etc.

Scanner object:

New is a keyword in java to create a new instance of that class.

```
import java.util.Scanner;

public class ChapterOneExamples {

    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        System.out.println("How old are you?");
        int age = keyboard.nextInt();
        System.out.println("You are " + age + " year(s) old.");
        keyboard.close();
    }
```

Types of Constructors:

1. **No argument constructors:**
   - **a.** No parameter passing
   - **b.** It is called when the object is instantiated.
   - **c.** It does not have a return type
   - **d.** The name is identical to the name of the class.

```
public class Car {

    private String manufacturer;
    private String model;
    private String year;

    //No-argument constructor
    public Car(){
        //Build the object
    }

}
```

   - **e.** In most cases, a class always has a constructor; if not, if not required, a default constructor is required that invokes the constructor of its super class.

```
public class Car {

    private String manufacturer;
    private String model;
    private String year;

    //Provided default constructor
    public Car(){
        super();
    }

}
```

2. **Multiple Constructors:**
    a. Created to handle missing or incomplete data.
    b. For unknown data
        i. getter(accessor) and setter(mutator) methods

```
public class Car {

    private int mileage;
    private String manufacturer;
    private String model;
    private String year;

    public Car(int mileage, String manufacturer, String model, String year) {
        this.mileage = mileage;
        this.manufacturer = manufacturer;
        this.model = model;
        this.year = year;
    }

    public Car(String manufacturer, String model, String year) {
        this.mileage = 0;
        this.manufacturer = manufacturer;
        this.model = model;
        this.year = year;
    }

}
```

```
public class Main {

    public static void main(String[] args) {

        Car car = new Car("Chevrolet", "Beretta", "1988");
    }

}
```

Object state and behavior:

State: It is a representation of data stored within the object.

Behavior: It describes operations on data or interaction with other objects.

```
public class Car {

    private int mileage;
    private String manufacturer;
    private String model;
    private String year;

    public Car(int mileage, String manufacturer, String model, String year) {
        this.mileage = mileage;
        this.manufacturer = manufacturer;
        this.model = model;
        this.year = year;
    }

    public Car(String manufacturer, String model, String year) {
        this.mileage = 0;
        this.manufacturer = manufacturer;
        this.model = model;
        this.year = year;
    }

    public int getMileage(){
        return this.mileage;
    }

    public void setMileage(int mileage){
        this.mileage = mileage;
    }

    public String getManufacturer(){
        return this.manufacturer;
    }

    public void setManufacturer(String manufacturer){
        this.manufacturer = manufacturer;
    }
```

Access Levels:

3 types of access levels

1. **Public:** The class itself and its instances or any other objects can access the state or behavior.
2. **Private:** Access to its object's state and behavior can only be done by itself.
3. **Protected:** The class itself, classes in the same package, and any subclass that inherits from it can access it states and behavior.

```
public class Car {

    private int mileage;
    private String manufacturer;
    private String model;
    private String year;

}
```

Scope:

Three types of attributes:

1. **Local attributes:** Scope is within the method only.
2. **Object (instance) attributes:** The attributes are shared among the different methods for that instance or object.

3. **Class attributes:** Attributes are shared across different instances/objects. These attributes share single 04-08-2023location in memory. In Java, it is achieved by making attribute as Static.

```
public class ProcessArray {

    private int[] array = {1,2,3,4,5};

    public int sumArray() {
        int sum = 0;
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }

    public int getAverage() {
        int sum = 0;
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum / array.length;
    }
```

Encapsulation and Information Hiding:

1. Modularity helps to break a program into small, individual, loosely coupled pieces of programs.
2. As the inner workings of methods are complex it is unnecessary to know how it exactly works. **Example how a car start?**
3. The process of hiding a system's inner working is known as information hiding.
4. The process of compartmentalizing the element of an abstraction that constitutes its structure and behavior.
5. The goal is to separate the object's internal composition and behavior from its external appearance and purpose. It is an integrity to the system.

Inheritance:

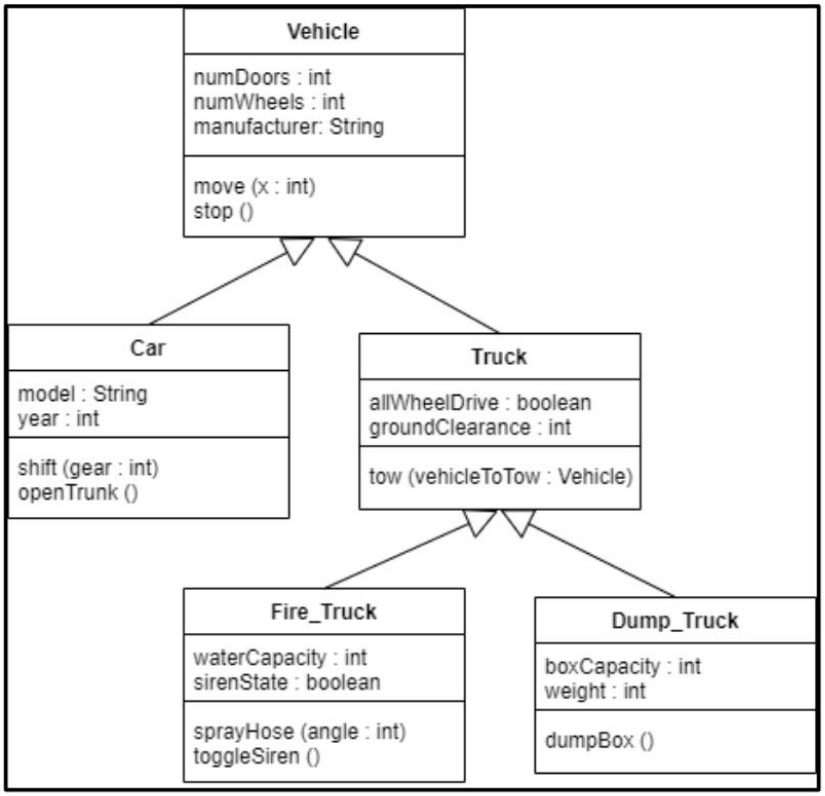
1. It provides an intuitive way of organizing class hierarchy that helps in code reuse and improving efficiency.
2. Inheritance is a relationship between different classes in which one class shares attributes of one or more different classes.
3. Inheritance is a relationship between a superclass and a subclass.
4. **Two types of inheritance**
    a. **Single inheritance:** It is a relationship in which a class has only on ancestor from which it directly inherits its attributes.
    b. **Multiple inheritance:** It occurs when a class calls on more than one superclass for properties.

```
public class Main {

    public static void main(String[] args) {
        //Create a dump truck
        Dump_Truck dt = new Dump_Truck();
        dt.dumpBox();
    }

}
```

```
public class Vehicle {

    int numDoors;
    int numWheels;
    String manufacturer;

    public void move(int x){
        //Code to move
    }

    public void stop() {
        //Code to stop
    }

}
```

```
public class Truck extends Vehicle {

    boolean allWheelDrive;
    int groundClearance;

    public void tow(Vehicle vehicleToTow) {
        //Code to tow
    }
```

```
public class Dump_Truck extends Truck {

    int box Capacity;
    int weight;

    public void dumpBox() {
        //Code to dump
    }
```



Overloading:

- It is one of the important concepts in object-oriented paradigm, wherein, different methods of a class can share same name.
- This is important when you don't know what parameters a method may need.
- Some operations on data can still be performed, hence the multiple functions with same name.
- Overloading follows three principles
    - The method being overloaded has the same name.
    - It is in the same class.
    - It has different parameters type.

```
public class Main {

    public static void main(String[] args) {

        System.out.println(add(1,1));
        System.out.println(add(1,1,1));

    }

    public static int add(int a, int b) {
        return a + b;
    }

    public static double add(double a, int b) {
        return a + b;
    }

    public static int add(int a, int b, int c) {
        return a + b + c;
    }

    public static double add(double a, double b) {
        return a + b;
    }

}
```

- When implementing constructors for a class, we are actually overloading different variations of constructors which allows the JVM to determine which one to call at runtime depending on the arguments passed. This is known as static polymorphism.

Overriding:

- The way of replacing a method inherited from a superclass with a newly defined method in the subclass.
- It is different from overloading.
- In overriding, the method has same name, same return type, and parameters, as the method that it is replacing in the superclass.

```
public class Main {

    public static void main(String[] args){
        Car car = new Car();
        car.move();
    }

}
```

```
public class Vehicle {

    public void move() {
        System.out.println("Vehicle is moving");
    }

}
public class Car extends Vehicle {

    public void move() {
        System.out.println("Car is moving");
    }

}
```

Output: Car is moving

Polymorphism:

- It is an engineering concept concerning the ability of separate classes and their derived instances to be accessed in the same way assuming that they are derived from the same superclass.
- It is achieved by encapsulating the inner working while allowing access to a familiar way.
- Two types of polymorphism
    - Static or compile time
    - Dynamic or run-time: - The type of object is determined during run-time. (i.e., the method to be get called.)

```
public class Main {

    public static void main(String[] args) {
        Animal a = new Cat();
        a.makeSound();
    }

}
```

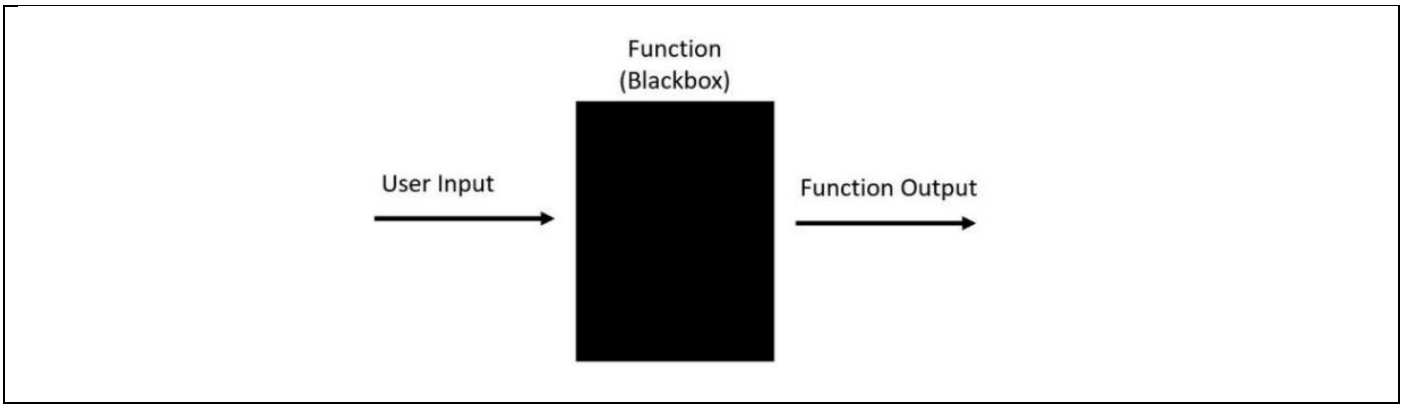
Output: Meow!

```
public class Animal {

    public void makeSound() {
        System.out.println("Animal makes sound.");
    }

}
public class Cat extends Animal {

    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }

}
```

Figure 2.7. Runtime Polymorphism.

Abstraction:

- Efficient management of complexity through the use of generalization.
- We are not interested in the inner workings of the methods, rather we focus on how we can utilize that method for our needs.
- Example Scanner class→ .nextLine(), .nextInt() methods
- Abstraction is a generalization of an object which includes only those details necessary to define it and to differentiate it from other objects.
- Abstraction is closely related to encapsulation and information hiding.



Abstract Classes:

- An abstract class is a class that is not supposed to be instantiated but rather serve to pass on its characteristics to more highly specified versions of itself.
- The class is not mean for instantiation, but rather pass its properties to its subclasses (child classes).

```
public class Main {

    public static void main(String[] args) {
        Six_Sided_Die six = new Six_Sided_Die();
        System.out.println(six.roll());
    }

}
```

```
public class Die {

    protected int sides;

    public int roll() {
        int i = (int) Math.ceil(Math.random() * this.sides);
        return i;
    }

}
```

```
public class Six_Sided_Die extends Die {

    public Six_Sided_Die() {
        super.sides = 6;
    }

}
```

```
public class Ten_Sided_Die extends Die {

    public Ten_Sided_Die() {
        super.sides = 10;
    }

}
```

- An abstract class is one way to enforce what methods should be implemented in the subclass.
- Keyword ***abstract*** is used to enforce the behavior.

```
public class Main {

    public static void main(String[] args) {
        // Instantiating Vehicle throws an error!
        // Vehicle v = new Vehicle();
        Car car = new Car();
        car.move();
    }

}
```

```
public abstract class Vehicle {

    abstract void move();

}
```

```
public class Car extends Vehicle {

    @Override
    void move() {
        System.out.println("Car is moving!");
    }

}
```

Generic components:

- Generics are the way to abstractly define class or methods that can be used in different way.
- Generics promote efficiency and reusability.

```
public class Main {

    public static void main(String[] args) {

        Character[] c = {'R', 'O', 'G', 'E', 'R'};
        Integer[] i = {1, 2, 3, 4, 5};

        print(c);
        print(i);

    }

    public static <T> void print(T[] x) {
        for(T i : x) {
            System.out.printf("%s", i);
        }
        System.out.println();
    }

}
```

```
public class Main {

    public static void main(String[] args) {

        PrintTest<String> stringObject = new PrintTest("Hello");
        System.out.println(stringObject.getObject());
        PrintTest<Double> doubleObject = new PrintTest(1.0);
        System.out.println(doubleObject.getObject());

    }

}

class PrintTest<T> {

    T object;

    public PrintTest(T object) {
        this.object = object;
    }

    public T getObject() {
        return this.object;
    }

}
```

Interface:

- An interface is a system that allows two entities to interact with each other.
- It is achieved by closing off an object's outward appearance from its inner working.
- Providing a set of methods for interaction.
- A class can implement multiple interfaces.
- Any method declared in the interface must be implemented within the class that is implementing that interface.

```
public class Car extends Vehicle implements MechanicInterface {

    @Override
    void move() {
        System.out.println("Car is moving!");
    }

    @Override
    public void displayBattery() {
        System.out.println("Battery is good!");
    }

}
```

```
public interface MechanicInterface {

    public void displayBattery();

}
```

Hierachies:

- We will discuss here, association, aggregation and composition.
- Object-oriented paradigm has two main types of hierarchy.
    - IS-A
    - HAS-A
- An IS-A relationship is formed through a hierarchical relationship of inheritance.
- The HAS-A relationship is a form of aggregation that specifies that an object can have another object.
- Aggregation specifies a PART-OF relationship.