What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while **object-oriented programming** is about creating objects that contain both data and methods.

Everything in Java is associated with **classes** and **objects**, along with its attributes and methods.

A class is a **user defined prototype** from which objects are created. It represents the set of **properties** or **methods** **that are common to all objects of one type**.

Object-oriented programming has several advantages over procedural programming:

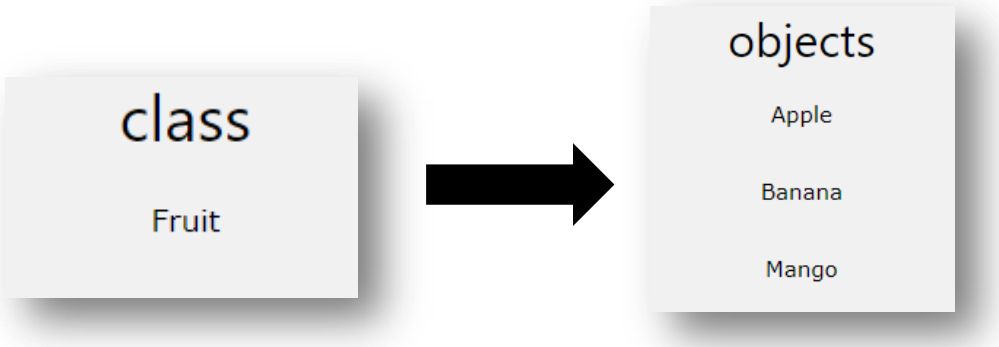
- OOP is **faster** and **easier** to execute
- OOP provides a **clear structure** for the programs
- OOP helps to keep the Java code **DRY "Don't Repeat Yourself",** and makes **the code easier to maintain, modify and debug**
- OOP makes it possible to create **full reusable applications** with less code and **shorter development time.**

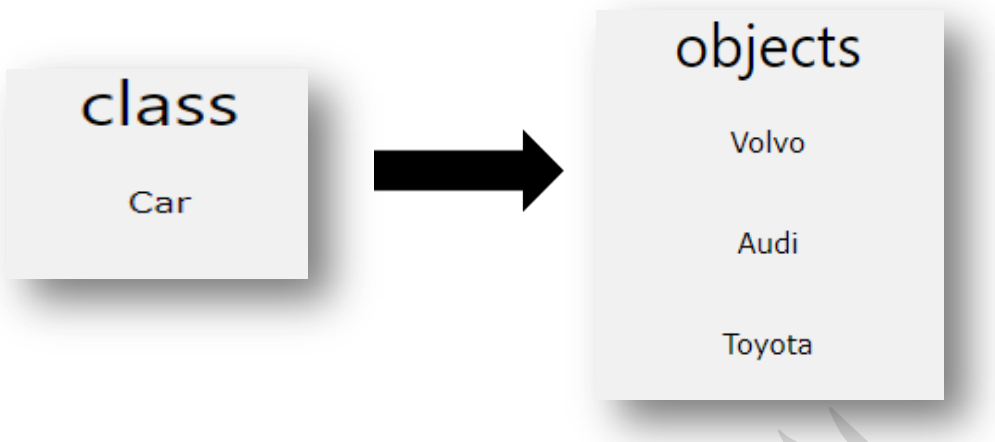
Note: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. **You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it**.

What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:





So, a class is a **template** for objects, and an object is an **instance of a class**.

When the individual objects are created, they inherit all the variables and methods from the class.

Create a Class

To create a class, use the keyword **class**:

Main.java

Create a class named "**Main**" with a variable x:

```
public class Main {
  int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, **specify the class name**, followed by the object name, and use the keyword **new**:

Example

Create an object called "**myObj**" and print the value of x:

```
public class Main {
  int x = 5;
```

```
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Output :

5

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of Main:

```
public class Main {
  int x = 5;
  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

Output :
5
5

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used **for better organization of classes** (**one class has all the attributes and methods, while the other class holds the main() method (code to be executed)**).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

Main.java

Second.java

Main.java

```
public class Main {
  int x = 15;
}
```

Output :
15

Second.java

```
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Object

It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Java Class Attributes

It is actually an attribute of the class. Or you could say that class attributes are **variables within a class:**

Example

Create a class called "Main" with two attributes: **x and y**:

```
public class Main {
  int x = 5;
  int y = 3;
}
```

Accessing Attributes

You can access attributes by creating an object of the class, and by using **the dot syntax (.):**

The following example will create an object of the Main class, with the name **myObj**. We use the x attribute on the object to print its value:

Example

Create an object called "**myObj**" and print the value of x:

```
public class Main {
  int x = 5;
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Modify Attributes

You can also modify attribute values:

Example

Set the value of x to 40:

```
public class Main {
```

```
  int x;
  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Example

Change the value of x to 25:

```
public class Main {
  int x = 10;
  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

If you don't want the ability to **override** existing values, declare the attribute as **final**.

Example

```
public class Main {
  final int x = 10;
```

```
  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

Example

Inside main, call myMethod():

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }
  public static void main(String[] args) {
    myMethod();
  }
}
```

Static vs. Non-Static method

You will often see Java programs that have either **static** or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

Example

An example to demonstrate the differences between static and public methods:

```
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }
  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error
    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Access Methods with an Object

Example

Create a Car object named myCar. Call the fullThrottle() and speed() methods on the myCar object, and run the program:

```
// Create a Main class
public class Main {
   // Create a fullThrottle() method
```

```
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }
  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();   // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}
```

Output is :

// The car is going as fast as it can!

// Max speed is: 200

Using Multiple Classes

It is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

Main.java

Second.java

Main.java

```
public class Main {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

Second.java

```
class Second {
  public static void main(String[] args) {
    Main myCar = new Main();     // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
```

Java Constructors

A constructor in Java is a special method that is used to **initialize objects**. **The constructor is called when an object of a class is created**. It can be used to set initial values for object attributes:

Example

Create a constructor:

// Create a Main class

```
public class Main {
  int x;  // Create a class attribute
  // Create a class constructor for the Main class
  public Main() {
    x = 5;  // Set the initial value for the class attribute x
  }
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x); // Print the value of x
  }
}
```

Note: **that the constructor name must match the class name**, **and it cannot have a return type (like void).**

Also note that the constructor is called when the object is created.

All classes have constructors by default: **if you do not create a class constructor yourself, Java creates one for you**. However, then you are not able to set initial values for object attributes.

Constructor Parameters

Constructors can also take **parameters**, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example

```
public class Main {
```

```
  int x;
  public Main(int y) {
    x = y;
  }
  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println(myObj.x);
  }
}
```

You can have as many parameters as you want:

Example

```
public class Main {
  int modelYear;
  String modelName;
  public Main(int year, String name) {
    modelYear = year;
    modelName = name;
  }
  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}
```