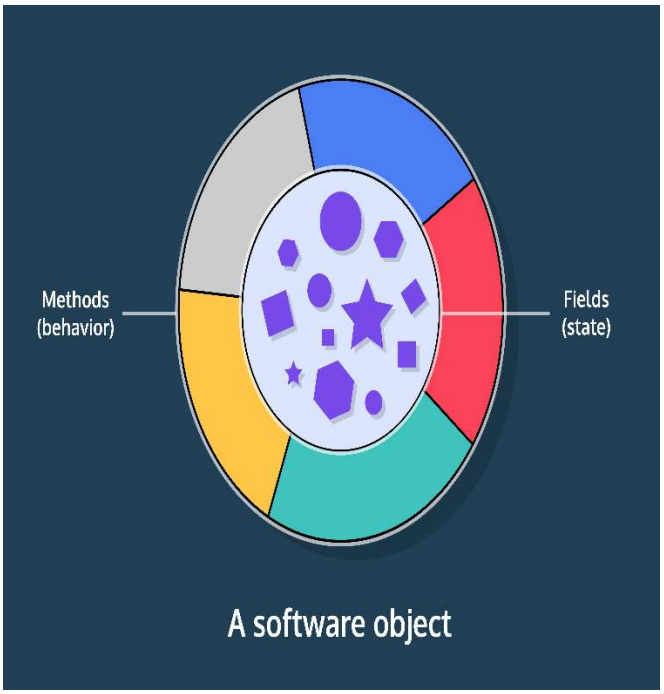
JAVA: INTRODUCTION TO CLASSES

Introduction to Classes

In every Java program, classes serve as representations of the real world.

A **class** is a template for creating objects in Java. Think of it as a blueprint for the representation of a real-world object. A class outlines the necessary components and how they interact with each other. For example, consider a program designed to monitor student test scores. This program can include classes such as Student and Grade to represent real-world entities of students and their grades. The Student class, representing a student, will have fields to store the student ID, all courses the student can enroll in, and several other fields that capture relevant information.



We represent each student as an instance, or object, of the Student class.

This is object-oriented programming: programs are built using objects.

Let's consider another example: a savings account at a bank.

What are the relevant details of a savings account in a bank? How about these fields:

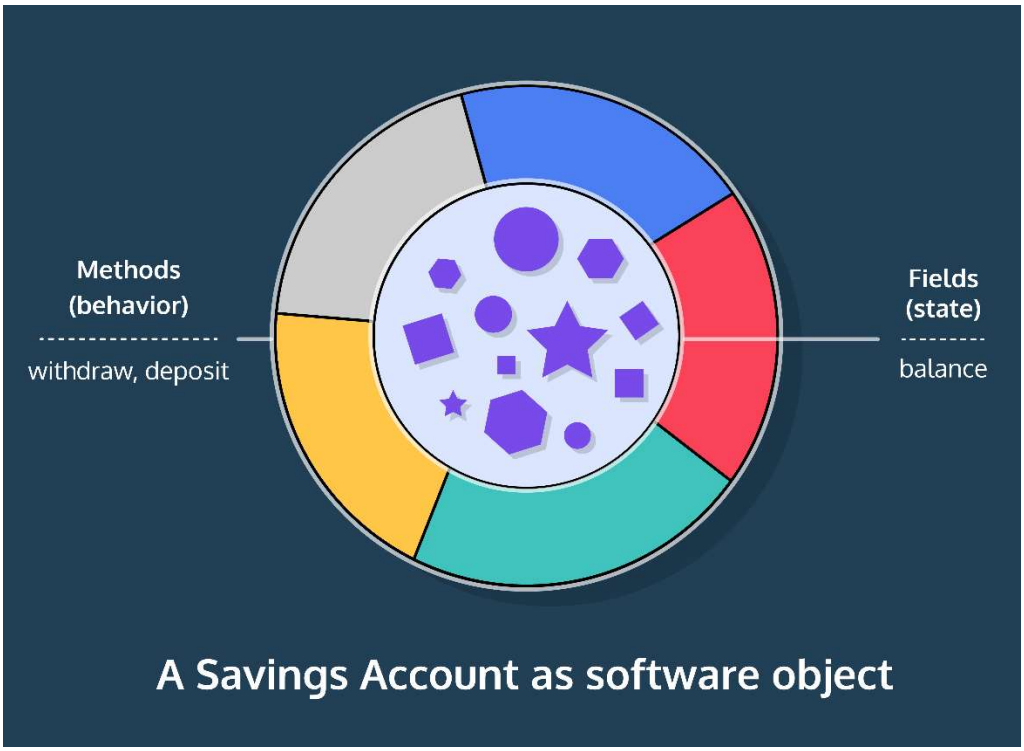
- Name of the owner
- Bank account number
- Amount of money in the account

What should a savings account do? Let's go with these functions:

- Deposit money.
- Withdraw money.

We can represent this data in a class called SavingsAccount.

Imagine two people each have a bank account. Each of their accounts will be represented by an instance of the SavingsAccount class.



Classes: Syntax

Let's explore how to define a class in Java. We will use a class called Car that represents a real-world car.

The Car class in Java is defined like so:

```
public class Car {

  // Empty Java Class

}
```

In this example, a class named Car is defined with the public keyword. We'll discuss what the keyword public means in a later exercise. Inside the body of the class, we can add fields to represent relevant information and functions to

represent how this class can interact with other objects. We will fill out this class in later exercises.

Let's practice creating classes by creating a class to represent a store.

1. In the code editor, create a public class called Store.
2. Inside the Store class, create a method called public void buyItems(). Keep the method body empty for now.

```
public class Store {
  public void buyItems() {
  }
}
```

Classes: Constructors

In Java, the constructor is a special type of method defined within the class, used to initialize fields when an instance of the class is created. The name of the constructor method must be the same as the class itself. Generally, the constructor is defined as public. Again, don't worry about the meaning of the public keyword for now. We will discuss this in a later exercise.

Let's look at an updated Car class, which includes a constructor.

```
public class Car {

  // Constructor
  public Car() {

    // instructions for creating a Car instance
  }
}
```

In the following example, the Car instance is assigned to the variable ferrari:

```
Car ferrari = new Car();
```

In this example, we have created an instance of a Car class called ferrari.

After the assignment operator, (=), we call the constructor method, Car(), using the keyword new to indicate that we're creating a new instance of the Car class. Omitting the new keyword causes an error.

Keep Reading: AP Computer Science A Students

If we print the value of the variable ferrari we would see its memory address: Car@76ed5528 In the above example, our variable ferrari is declared as a reference data type rather than with a primitive data type like int or boolean. This means that the variable holds a reference to the memory address of an instance. During its declaration, we specify the class name as the variable's type, which in this case is Car.

If we use a special value, null, we can initialize a reference-type variable without giving it a reference. If we were to assign null to an object, it would have a void reference because null has no value.

For example, consider the following code snippet where we create an instance of a Car, assign it a reference, and then set its value to null:

```
public class Car {
  public static void main(String[] args) {

    Car thunderBird = new Car();

    System.out.println(thunderBird); // Prints: Car@76ed5528

    thunderBird = null; // change value to null

    System.out.println(thunderBird); // Prints: null
  }
}
```

It's important to understand that using a null reference to call a method or access an instance variable will result in a NullPointerException error.

Classes: Instance Fields

A real car has characteristics such as brand, model, year, color, etc. In a Java object representing a car, these characteristics are represented by *instance fields* or *instance variables*.

In the last exercise, we created an object, but our object has no characteristics! We'll add characteristics to our class by including instance fields or instance variables Let's revisit the Car class example.

We want our Car objects to have different colors, so we declare an instance field called color. Instance variables are often characterized by their "has-a" relationship with the object. For example, a Car "has-a" colour, "has-a" make, "has-a" model name, and "has-a" model year.

Think about what qualities other than color a car might have.

```
public class Car {
   /* declare fields inside the class
   by specifying the type and name */

   public String color;
   public int year;
   public String modelName;
   public String make;

   public Car() {
    /*instance fields available in
    scope of the constructor method */
   }
  }
```

Instance variables are specific to each instance of the class which means that each object created from the class will have its own copy of these variables. These fields can be set in the following three ways:

- If they are `public`, they can be set like this `instanceName.fieldName = someValue;`
- They can be set by class methods
- They can be set by the constructor method (shown in the next exercise).

Classes: Constructor Parameters

In Java, parameters are placeholders that we can use to pass information to a method.

Since the constructor is a method, we can include parameters to assign values to instance fields.

Here the `Car` constructor has a parameter: `String carColor`:

```
public class Car {
  public String color;

  // constructor method with a parameter
  public Car(String carColor) {
    // parameter value assigned to the field
    color = carColor;
  }
}
```

Now, when we create a new instance of the `Car` class and pass in a string value to the constructor, it will be stored in the parameter `carColor`. Inside the constructor, we can use this passed value however we want. In our example, we assign the value stored in `carColor` to the instance field `color`.

A method can be characterized by its **signature**, which is the name, number of, and parameters of the method. In the above example, the signature is `Car(String carColor)`.

Later, we'll learn how to pass values into a method!

There are two types of parameters: formal and actual. The parameter we defined in the above example, `String carColor`, is a formal parameter. We can think of them as variables that will store the data that is passed into a method. It specifies the type and name of the data.

We'll learn about actual parameters in the next exercise.

For now, let's practice working with constructor parameters.

Keep Reading: AP Computer Science A Students

A class can have multiple constructors We can differentiate them based on their parameters. The signature helps the compiler to differentiate between different methods

For example, here we have defined two constructors:

```
public class Car {
  public String color;
  public int mpg;
  public boolean isElectric;

  // constructor 1
  public Car(String carColor, int milesPerGallon) {
    color = carColor;
    mpg = milesPerGallon;
  }
  // constructor 2
  public Car(boolean electricCar, int milesPerGallon) {
    isElectric = electricCar;
    mpg = milesPerGallon;
  }
}
```

The first constructor has two parameters: `String carColor` and `int milesPerGallon`.

While the second one has these: `boolean electricCar` and `int milesPerGallon`.

The values will help the compiler to decide which constructor to use. For example, `Car myCar = new Car(true, 40)` will be created by the second constructor because the arguments match the type and order of the second constructor's signature.

When we don't define the constructor, the Java compiler creates a default constructor that assigns default values to an instance. Default values can be created by assigning values to the instance fields during their declaration:

```
public class Car {
  public String color = "red";
  public boolean isElectric = false;
  public int cupHolders = 4;

  public static void main(String[] args) {
    Car myCar = new Car();
    System.out.println(myCar.color); // Prints: red
  } }
```

Notice that the color instance field of the myCar object will have a red value because we've already defined the default value during the declaration.

Classes: Assigning Values to Instance Fields

Since the constructor now accepts a parameter, let's see how we can use this constructor to create an instance of an object with initial values for its fields.

To use the constructor, we call it just as we would an ordinary method and pass in values for the parameters. These values, known as **arguments**, will be used to initialize the instance fields of the created object.

Let's revisit our previous example of the Car class.

```
public class Car {
  public String color;

  public Car(String carColor) {
    // assign parameter value to instance field
    color = carColor;
  }
}
```

In this case, when creating a specific instance of Car called ferrari, we pass the string "red" as the value for the carColor parameter.

```
class Main{
    public static void main(String[] args){
    Car ferrari = new Car("red");
    }
  }
```

When passing in values to a constructor, just like an ordinary method, the type of the value must match the type of the parameter.

In the code, we pass the String value “red” to the constructor method call: new Car("red"). The parameter carColor of type String now refers to the value passed in during the method call, which is “red”.

The field color of the object ferrari now has a value of “red”.

Remember, that we can access the fields of an object by using the dot operator like so:

```
ferrari.color; // "red"
```

Keep Reading: AP Computer Science A Students

An argument refers to the actual values passed during the method call while a parameter refers to the variables declared in the method signature.

When we pass an argument, a copy of the argument value is passed to the parameter rather than the actual variables. This process of calling a method with an argument value is called a call-by-value.

For example, we passed the String value “red” as an argument, but a copy of this value is assigned to the parameter carColor.

For example
Create a new instance of Store called lemonadeStand in the main() method of **Main.java** and pass “lemonade” as the argument

```
public class Store {
  // instance fields
  String productType;
  int inventoryCount;
  double inventoryPrice;

  // constructor method
  public Store(String product, int count, double price) {
    productType = product;
    inventoryCount = count;
    inventoryPrice = price;
  }}
```

Inside the main() method, print the instance field productType from lemonadeStand

```
public class Main{
    public static void main(String[] args) {
   Store lemonadeStand = new Store("lemonade");

   System.out.println(lemonadeStand.productType);
     }
   }
```

Classes: Multiple Fields

Objects are not limited to a single instance field. We can declare as many fields as necessary for our program's requirements. To illustrate this, let's add two more instance fields to our Car instances.

We'll add a boolean isRunning, which represents whether the car engine is on or not, and an int velocity, which indicates the speed at which the car is traveling.

```
public class Car {
  String color;

  // new fields!
  boolean isRunning;
  int velocity;

  // new parameters that correspond to the new fields
  public Car(String carColor, boolean carRunning, int milesPerHour) {
    color = carColor;

    // assign new parameters to the new fields
    isRunning = carRunning;
    velocity = milesPerHour;
  }
}
```

```
Public class Main(){

  public static void main(String[] args) {
    // new values passed into the method call
    Car ferrari = new Car("red", true, 27);
    Car renault = new Car("blue", false, 70);

    System.out.println(renault.isRunning); // false
    System.out.println(ferrari.velocity); // 27
  }
}
```

Now, the constructor has two new parameters: `boolean carRunning` and int speed`.

Remember, it's important to pass the arguments in the same order as they are listed in the parameters.

```
// values match types, no error
Car honda = new Car("green", false, 0);

// values do not match types, error!
Car junker = new Car(true, 42, "brown");
```

Classes: Review

Object-oriented programming revolves around classes and objects. The `class` is a fundamental concept of OOP and programs in Java are built with multiple classes and their objects.
Let's review what we've learned throughout this lesson:

- A class is a blueprint to create instances. It defines the state and behavior of these instances.
- Every class has a special method called constructor which is invoked when a new object is created. Constructors initialize the state of newly created instances.
- Instance fields define the characteristics of an object. We can declare them within a class but outside of any method or constructor.
- We use the dot operator (.) to access the instance fields.
- A program can have multiple classes, instances, and instance fields as per our program's requirements.

Later, we will explore how a program can be made from multiple classes. For now, our programs have a single class.

```
public class Dog {
  // instance field
  public String breed;

  // constructor method
```