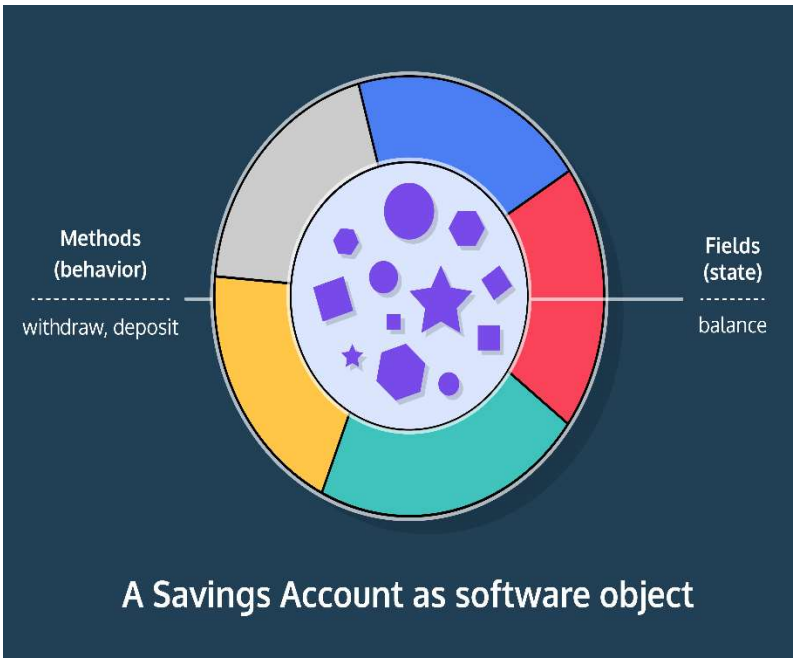
LEARN JAVA: METHODS

Introduction

In the last lesson, we learned that objects have state and behavior. We have seen how to give objects state through instance fields. Now, we're going to learn how to create object *behavior* using methods Remember our example of a Savings Account.



The state tells us what a savings account should know:

- The balance of money available

The behavior tells us what tasks a savings account should be able to perform:

- Depositing - increasing the amount available
- Withdrawing - decreasing the amount available
- Checking the balance - displaying the amount available.

Methods are repeatable, modular blocks of code used to accomplish specific tasks. We have the ability to define our own methods that will take input, do something with it, and return the kind of output we want.

Looking at the example above, recreating a savings account is no easy task. How can one program tackle such a large problem? This is where methods with their ability to accomplish smaller, specific tasks come in handy. Through *method decomposition*, we can use methods to break down a large problem into smaller, more manageable problems.

Methods are also reusable. Imagine we wrote a sandwich-making program that used 20 lines of code to make a single sandwich. Our program would become very long very quickly if we were making multiple sandwiches. By creating a `makeSandwich()` method, we can make a sandwich anytime simply by calling it.

makeSandwich( )

In this lesson, we'll learn how to create and call our very own methods inside of our programs.

Keep Reading: AP Computer Science A Students

If we were to share this sandwich-making program with another person, they wouldn't have to understand *how* `makeSandwich()` worked. If we wrote our program well, all they would need to know is that if they called `makeSandwich()`, they would receive a sandwich. This concept is known as *procedural abstraction*: knowing what a method does, but not how it accomplishes it.

Example
1. Look at the code editor in the file **Main.java**. We have created a `Main` class with no method except `main()` and the class constructor. Run the code and observe the output.

Notice that `main()` is very lengthy and contains many print statements. The

repetitive code can be tedious to write if we need to make more than five deposits.

In this lesson, we'll learn how to make methods to deposit, withdraw, and check account balance for the `SavingsAccount` class.

```
// savingsAccount.java file
public class savingsAccount {

    int balance;

    public SavingsAccount(int initialBalance){
      balance = initialBalance;
    }
  }
```

```
public class Main {
  public static void main(String[] args) {
    SavingsAccount savings = new SavingsAccount(2000);

    // Check balance:
    System.out.println("Hello!");
    System.out.println("Your balance is " + savings.balance);

    // Withdrawing:
    int afterWithdraw = savings.balance - 300;
    savings.balance = afterWithdraw;
    System.out.println("You just withdrew " + 300);

    // Check balance:
    System.out.println("Hello!");
    System.out.println("Your balance is " + savings.balance);
```

```
    // Deposit:
    int afterDeposit = savings.balance + 600;
    savings.balance = afterDeposit;
    System.out.println("You just deposited " + 600);

    // Check balance:
    System.out.println("Hello!");
    System.out.println("Your balance is " + savings.balance);

    // Deposit:
    int afterDeposit2 = savings.balance + 600;
    savings.balance = afterDeposit2;
    System.out.println("You just deposited " + 600);

    // Check balance:
    System.out.println("Hello!");
    System.out.println("Your balance is " + savings.balance);
  }
}
```

Defining Methods

If we were to define a `checkBalance()` method for the Savings Account example we talked about earlier, it would look like the following:

```
public void checkBalance(){
  System.out.println("Hello!");
  System.out.println("Your balance is " + balance);
}
```

The first line, `public void checkBalance()`, is the method declaration. It gives the program some information about the method:

- `public` means that other classes can access this method. We will learn more about that later in the course.
- The `void` keyword means that there is no specific output from the method. We will see methods that are not `void` later in this lesson, but for now, all of our methods will be `void`.
- `checkBalance()` is the name of the method.

Every method has its own unique *method signature* which is composed of the method's name and its parameter type. In this example, the method signature is `checkBalance()`.

The two print statements are inside the *body* of the method, which is defined by the curly braces: `{` and `}`.

Anything we can do in our `main()` method, we can do in other methods! All of the Java tools you know, like the math and comparison operators, can be used to make interesting and useful methods.

Keep Reading: AP Computer Science A Students

`checkBalance()` is considered a non-static method because its signature does not include the keyword `static` like the `main()` method does. We'll learn more about non-static methods later in this course.

Example

1. In the **Store** class, add a new method called **advertise()** that is accessible by other classes and does not return anything. Don't add anything to the body of the method.

Well done! Let's add some print statements inside our `advertise()` method. Print these two statements:

"Come spend some money!"

"Selling <productType>!"

The value stored in the variable `<productType>` should be printed instead of "productType" string.

Don't worry if you are not able to see the output. We'll learn how to use the `advertise()` method in later exercises.

```
public class Store {
  // instance fields
  String productType;

  // constructor method
  public Store(String product) {
    productType = product;
  }

// Add advertise method below
  public void advertise()
  {
    System.out.println("Come spend some money!" );
 System.out.println("Selling " + productType +" !");
  }
}
```

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

  }
}
```

Calling Methods

When we add a non-static method to a class, it becomes available to use on an object of that class. In order to have our methods get executed, we must *call* the method on the object we created.

Let's add a non-static `startEngine()` method to our `Car` class from the previous lesson. Inside the `main()` method, we'll call `startEngine()` on the `myFastCar` object:

```
class Car {

  String color;

  public Car(String carColor) {
    color = carColor;
  }

  public void startEngine() {
    System.out.println("Starting the car!");
    System.out.println("Vroom!");
  }

  public static void main(String[] args){
    Car myFastCar = new Car("red");
    // Call a method on an object
    myFastCar.startEngine();
    System.out.println("That was one fast car!");
  }
}
```

Let's take a closer look at the method call:

```
myFastCar.startEngine();
```

First, we reference our object `myFastCar`. Then, we use the *dot operator* (`.`) to call the method `startEngine()`. Note that we must include parentheses `()` after our method name in order to call it.

If we run the above program, we will get the following output

```
Starting the car!
Vroom!
That was one fast car!
```

Code generally runs in a top-down order where code execution starts at the top of a program and ends at the bottom of a program; however, methods are ignored by the compiler unless they are being called.

When a method is called, the compiler executes every statement contained within the method. Once all method instructions are executed, the top-down order of execution continues. This is why `Starting the car!` and `Vroom!` are outputted before `That was one fast car!`.

Now our `Store` class has a new method called `advertise()`, but we didn't see its output.

In the `main()` method of **Main.java**, call the `advertise()` method of the `lemonadeStand` instance and observe the displayed output.

Now, replace the argument "Lemonade" with "Coffee" in the instance definition and run the code.

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

    lemonadeStand.advertise();
  }
}
```

```
public class Store {
  // instance fields
  String productType;
  // constructor method
  public Store(String product) {
    productType = product;
  }
  // advertise method
  public void advertise() {
    System.out.println("Selling " + productType +
"!");
    System.out.println("Come spend some
money!");
  }
}
```

Scope;

A method is a task that an object of a class performs.

We mark the domain of this task using curly braces: `{`, and `}`. Everything inside the curly braces is part of the task. This domain is called the *scope* of a method.

We can't access variables that are declared inside a method in code that is outside the scope of that method.

Looking at the `Car` class again:

```
class Car {
  String color;
  int milesDriven;

  public Car(String carColor) {
    color = carColor;
    milesDriven = 0;
  }

  public void drive() {
    String message = "Miles driven: " + milesDriven;
```

```
    System.out.println(message);
  }

  public static void main(String[] args){
    Car myFastCar = new Car("red");
    myFastCar.drive();
  }
}
```

The variable `message`, which is declared and initialized inside of `drive`, cannot be used inside of `main()`! It only exists within the *scope* of the `drive()` method.

However, `milesDriven`, which is declared at the top of the class, can be used inside all methods in the class, since it is in the scope of the whole class.

Example

1.Let's see how the scope works!

Inside the **Store.java** file, check out the `advertise()` method. Notice how we are using `productType`.

Now, replace the `productType` variable with the `product` variable inside the `advertise()` method.

When you run the code, you will notice an error has occurred. This is due to the `product` variable not being in the `advertise()` method's scope. The variable `product` was declared in the constructor, so we are not able to access it.

2.To correct the error from the previous checkpoint, in the `advertise()` method, change the `product` variable back to `productType`.

Switch over to **Main.java** and run your code.

```
public class Store {
  // instance fields
  String productType;

  // constructor method
  public Store(String product) {
    productType = product;
  }

  // advertise method
  public void advertise() {
    String message = "Selling " + productType +
"!";
    System.out.println(message);
  }
}
```

```
// mian.java
public class Main {
  public static void main(String[] args) {
    String cookie = "Cookies";
    Store cookieShop = new Store(cookie);

    cookieShop.advertise();
  }
}
```

Adding Parameters

We saw how a method's scope prevents us from using variables declared in one method in another method. What if we had some information in one method that we needed to pass into another method?

Similar to how we added parameters to *constructors*, we can customize all other methods to accept parameters. For example, in the following code, we create a `startRadio()` method that accepts a `double` parameter, `stationNum`, and a `String` parameter called `stationName`:

```
class Car {

  String color;

  public Car(String carColor) {
    color = carColor;
  }

  public void startRadio(double stationNum, String stationName) {
    System.out.println("Turning on the radio to " + stationNum + ", " + stationName + "!");
    System.out.println("Enjoy!");
  }

  public static void main(String[] args){
    Car myCar = new Car("red");
    myCar.startRadio(103.7, "Meditation Station");
  }
}
```

Adding parameter values impacts our method's signature. Like constructor signatures, the method signature includes the method name as well as the parameter types of the method. The signature of the above method is `startRadio(double, String)`.

In the `main()` method, we call the `startRadio()` method on the `myCar` object and provide a `double` argument of `103.7` and `String` argument of `"Meditation Station"`, resulting in the following output:

```
Turning on the radio to 103.7, Meditation Station!
Enjoy!
```

Note that when we call on a method with multiple parameters, the arguments given in the call must be placed in the same order as the parameters appear in the signature. If the argument types do not match the parameter types, we'll receive an error.

Keep Reading: AP Computer Science A Students

Through method overloading, our Java programs can contain multiple methods with the same name as long as each method's parameter list is unique. For example, we can recreate our above program to contain two `startRadio()` methods:

```
// Method 1
public void startRadio(double stationNum, String stationName) {
  System.out.println("Turning on the radio to " + stationNum + ", " + stationName + "!");
  System.out.println("Enjoy!");
}

// Method 2
public void startRadio(double stationNum) {
  System.out.println("Turning on the radio to " + stationNum + "!");
}

public static void main(String[] args){
  Car myCar = new Car("red");
  // Calls the first startRadio() method
  myCar.startRadio(103.7, "Meditation Station");

  // Calls the second startRadio() method
  myCar.startRadio(98.2);
}
```

Example

1.Let's create a method to greet the customers.

In Store.java, add a greetCustomer() method to the Store class that is accessible by other classes and does not return anything.

*Note: it is okay to get an error regarding a missing main() method. The main() method is defined in **Main.java**.*

2. Now, add a **String** parameter to the **greetCustomer()** method called **customerName**.

3. Inside greetCustomer(), print a string that has the following format:

"Welcome to the store, " + customerName + "!"

4. Inside the **main()** method of **Main.java**, call the greetCustomer() method on the **lemonadeStand** object. Pass in a name of your choice!

```
// store.java
public class Store {
  // instance fields
  String productType;

  // constructor method
  public Store(String product) {
    productType = product;
  }

  // advertise method
  public void advertise() {
    String message = "Selling " + productType + "!";
    System.out.println(message);
  }

  // greetCustomer() method
  public void greetCustomer(String customerName)
  {
System.out.println("Welcome to the store, " + customerName + "!");
  }
}
```

```
// main.java
public class Main{
  public static void main(String[] args) {
    Store lemonadeStand = new Store("Lemonade");
    lemonadeStand.greetCustomer("yashwanth hk");
  }
}
```

Reassigning Instance Fields

Earlier, we thought about a Savings Account as a type of object we could represent in Java.Two of the methods we need are depositing and withdrawing:

```
public class SavingsAccount{
  double balance;
  public SavingsAccount(double startingBalance){
    balance = startingBalance;
  }

  public void deposit(double amountToDeposit){
    //Add amountToDeposit to the balance
  }

  public void withdraw(double amountToWithdraw){
    //Subtract amountToWithdraw from the balance
  }

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

These methods would change the value of the variable `balance`. We can *reassign* balance to be a new value by using our assignment operator, `=`, again.

```
public void deposit(double amountToDeposit){
  double updatedBalance = balance + amountToDeposit;
  balance = updatedBalance;
}
```

Now, when we call `deposit()`, it should change the value of the instance field `balance`:

```
public static void main(String[] args){
  SavingsAccount myAccount = new SavingsAccount(2000);
  System.out.println(myAccount.balance);
  myAccount.deposit(100);
  System.out.println(myAccount.balance);
}
```

This code first prints `2000`, the initial value of `myAccount.balance`, and then prints `2100`, which is the value of `myAccount.balance` after the `deposit()` method has run.

Changing instance fields is how we change the state of an object and make our objects more flexible and realistic.

Example

1.We have added a new instance field to our Store class called price and an empty increasePrice() method.

Add a parameter double priceToAdd to the increasePrice() method.

*Note: it is okay to get an error regarding a missing main() method. The main() method is defined in **Main.java**.*

2.We need to create a variable to store the increased price of our product.

Inside increasePrice(), create a new variable called double newPrice and set it equal to the addition of price and priceToAdd.

newPrice = price + priceToAdd

3.Inside the **increasePrice()** method, assign the **newPrice** to the instance field **price**.
4.Let's use our increasePrice() method inside main() in **Main.java**.

Increase the price of lemonadeStand by 1.50 using increasePrice() method.

Print the instance field price to see the updated price of lemonadeStand.

```
public class Store {
  // instance fields
  String productType;
  public double price;

  // constructor method
  public Store(String product, double initialPrice)
{
    productType = product;
    price = initialPrice;
  }
  // increase price method
  public void increasePrice(double priceToAdd){
      double newPrice = price + priceToAdd;
      price = newPrice;
  }
}
```

```
// Main.java

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

```
 Store lemonadeStand = new Store("Lemonade",
3.75);

lemonadeStand.increasePrice(1.50);

System.out.println(lemonadeStand.price);

  }
}
```

Returns

Remember, variables can only exist in the *scope* that they were declared in. We can use a value outside of the method it was created in if we *return* it from the method.

We return a value by using the keyword `return`:

```
public int numberOfTires() {
  int tires = 4;
  // return statement
  return tires;
}
```

This method, called `numberOfTires()`, returns `4`. Once the return statement is executed, the compiler exits the function. Any code that exists after the return statement in a function is ignored.

In past exercises, when creating new methods, we used the keyword `void`. Here, we are replacing `void` with `int`, to signify that the *return type* is an `int`.

The void keyword (which means "completely empty") indicates that no value is returned after calling that method.

A non-void method, like `numberOfTires()` returns a value when it is called. We can use datatype keywords (such as `int`, `char`, etc.) to specify the type of value the method should return. The return value's type must match the return type of the method. If the return expression is compatible with the return type, a copy of that value gets returned in a process known as *return by value*.

Unlike void methods, non-void methods can be used as either a variable value or as part of an expression like so:

```
public static void main(String[] args){
   Car myCar = new Car("red");
   int numTires = myCar.numberOfTires();
}
```

Within `main()`, we called the `numberOfTires()` method on `myCar`. Since the method returns an `int` value of `4`, we store the value in an integer variable called `numTires`. If we printed `numTires`, we would see `4`.

Keep Reading: AP Computer Science A Students

We learned how to return primitive values from a method, but what if we wanted our method to return an object? Returning an object works a little differently than returning a primitive value. When we return a primitive value, a copy of the value is returned; however, when we return an object, we return a reference to the object instead of a copy of it.

Let's create a second class, `carLot`, that takes in a `Car` as a parameter and contains a method which returns a `Car` object.

```
class CarLot {
    Car carInLot;
    public CarLot(Car givenCar) {
        carInLot = givenCar;
    }

    public Car returnACar() {
        // return Car object
        return carInLot;
    }

    public static void main(String[] args) {
        Car myCar = new Car("red", 70);
        System.out.println(myCar);
        CarLot myCarLot = new CarLot(myCar);
        System.out.println(myCarLot.returnACar());
    }
}
```

This code outputs the same memory address because `myCar` and `carInLot` have the same reference value:

```
Car@2f333739
Car@2f333739
```

example

1.Add a new method called `getPriceWithTax()` in the `Store` class that will return the price of the product including the tax.

Set the return type to `double` to indicate that it returns a double value, and it should take in no parameters.

For now, leave the body of the method empty.

2. Inside the `Store` class, we have an instance field called `double tax` with the value of `0.08`.

We need another variable to calculate the total price of the product including tax.

Inside the `getPriceWithTax()` method, create a variable called `totalPrice` of type `double` and set it equal to the new price with tax. The formula for calculating a price with tax is the following:

$$totalPrice = price + price \times tax$$

Note: for testing purposes, please write the formula exactly as we have shown it. The tests may fail otherwise.

3. At the end of `getPriceWithTax()` method, return the variable `totalPrice`.
4. Inside `main()` in **Main.java**, create a `double` variable `lemonadePrice` and set it to `lemonadeStand.getPriceWithTax()`.

Also, print your results.

```
public class Store {
  // instance fields
  String productType;
  double price;
  double tax = 0.08;

  // constructor method
  public Store(String product, double initialPrice) {
    productType = product;
    price = initialPrice;
  }
  // increase price method
  public void increasePrice(double priceToAdd){
    double newPrice = price + priceToAdd;
    price = newPrice;
  }
  // get price with tax method
  public double getPriceWithTax(){
    double totalPrice = price + price * tax;
    return totalPrice;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Store lemonadeStand = new Store("Lemonade", 3.75);
```

```
    double lemonadePrice = lemonadeStand.getPriceWithTax();

    System.out.println(lemonadePrice);
}
}
```

The toString() Method

When we print out Objects, we often see a String that is not very helpful in determining what the Object represents. In the last lesson, we saw that when we printed our `Store` objects, we would see output like:

```
Store@6bc7c054
```

where `Store` is the name of the object and `6bc7c054` is its position in memory.

This doesn't tell us anything about what the `Store` sells, the price, or the other instance fields we've defined. We can add a method to our classes that makes this printout more descriptive.

When we define a *toString()* method for a class, we can return a `String` that will print when we print the object:

```
class Car {

    String color;

    public Car(String carColor) {
        color = carColor;
    }

    public static void main(String[] args){
        Car myCar = new Car("red");
        System.out.println(myCar);
    }

   public String toString(){
       return "This is a " + color + " car!";
   }
}
```

When this runs, the command `System.out.println(myCar)` will print `This is a red car!`, which tells us about the Object `myCar`.

Example

1.The Store class currently does not contain a custom definition of toString(). Let's see what happens when we call System.out.println() on two instances of Store.

Call System.out.println() on lemonadeStand and cookieShop inside the main() method of **Main.java**.

2. When we print our objects, we want them to return a meaningful and informative string.

Create a toString() method inside the Store class. Leave it empty for now.

Note: It is okay to get an error in the output claiming that toString() should return a String. You will fill out the toString() method in the next step.

3. Fill in the toString() method of the Store class such that it returns this string:

"This store sells <productType> at a price of <price>."

Where <productType> and <price> are the values in the instance fields of the same name.

For example, if we had a hat store implementation of Store where hats cost 8, the string would be:

This store sells hats at a price of 8.

*Note: If you click "Run" while you still have the **Store.java** file open, you will get an error in the output regarding a missing main() method. This is an okay error to get because the method is located in the **Main.java** file.*

4. Open **Main.java** and click "Run"

```
public class Store {
  // instance fields
  public String productType;
  public double price;

  // constructor method
  public Store(String product, double initialPrice) {
    productType = product;
    price = initialPrice;
  }
  public String toString(){
    return "This store sells " + productType + " at a price of " +price;
```

```
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Store lemonadeStand = new Store("Lemonade", 3.75);
    Store cookieShop = new Store("Cookies", 5);

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

Review

Great work! Methods are a powerful way to abstract tasks away and make them repeatable. They allow us to define behavior for classes, so that the Objects we create can do the things we expect them to. Let's review everything we have learned about methods so far.

- *Defining a method* : Method declarations will declare a method's return type, name, and parameters
- *Calling a method* : Methods are invoked with a `.` and `()`
- *Parameters* : Inputs to the method and their types are declared in parentheses in the method signature
- *Changing Instance Fields* : Methods can be used to change the value of an instance field
- *Scope* : Variables only exist within the domain that they are created in
- *Return* : The type of the variables that will be output are declared in the method declaration

As you move through more Java material, it will be helpful to frame the tasks you create in terms of methods. This will help you think about what inputs you might need and what output you expect.

Example

We have a `SavingsAccount` class for you to experiment with in the code editor. Feel free to modify it and try out what you learned.

```
public class SavingsAccount {
  int balance;

  public SavingsAccount(int initialBalance){
```

```
    balance = initialBalance;
  }

  public void checkBalance(){
    System.out.println("Hello!");
    System.out.println("Your balance is " + balance);
  }

  public void deposit(int amountToDeposit){
    balance = balance + amountToDeposit;
    System.out.println("You just deposited " + amountToDeposit);
  }

  public int withdraw(int amountToWithdraw){
    balance = balance - amountToWithdraw;
    System.out.println("You just withdrew " + amountToWithdraw);
    return amountToWithdraw;
    }

  public String toString(){
    return "This is a savings account with " + balance + " saved.";
  }

}
```

```
public class Main {
  public static void main(String[] args) {
    SavingsAccount savings = new SavingsAccount(2000);

    //Check balance:
    savings.checkBalance();

    //Withdrawing:
    savings.withdraw(300);

    //Check balance:
    savings.checkBalance();

    //Deposit:
    savings.deposit(600);

    //Check balance:
```

```
    savings.checkBalance();

    //Deposit:
    savings.deposit(600);

    //Check balance:
    savings.checkBalance();

    System.out.println(savings);
  }
}
```