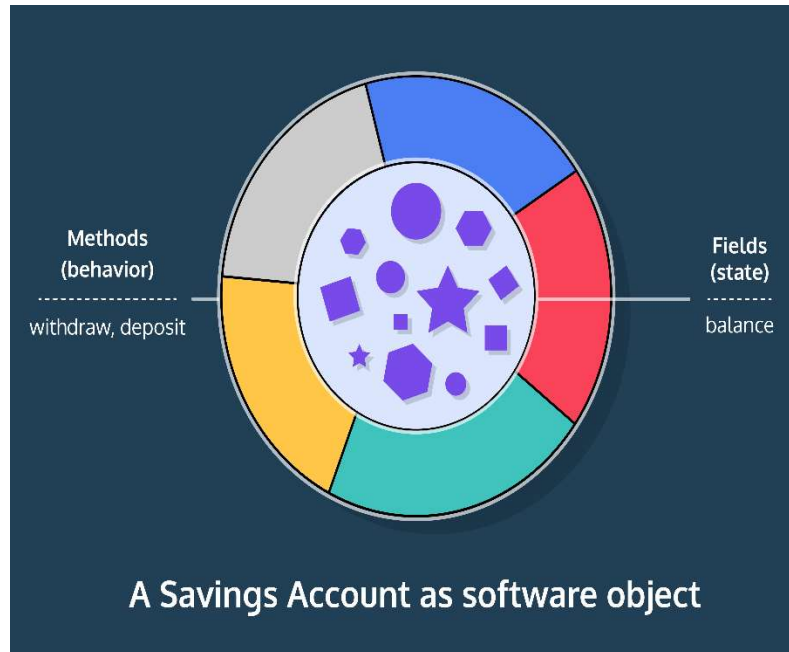


# LEARN JAVA: METHODS

## 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!

```
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 + "!"  
    }  
}
```

```
// store.java
```

```
// 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`.

$$\text{newPrice} = \text{price} + \text{priceToAdd}$$

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;
    }
}
```

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

lemonadeStand.increasePrice(1.50);

System.out.println(lemonadeStand.price);

}
}
```

```
// Main.java
```

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

## 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();

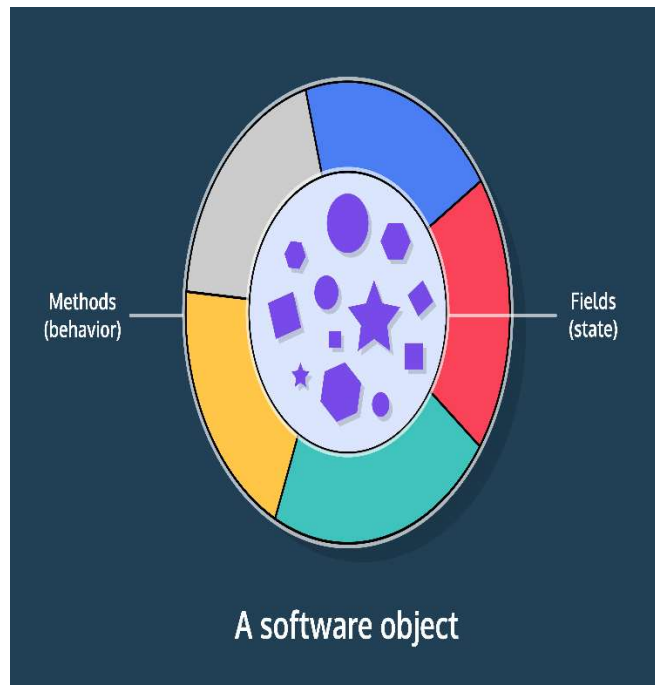
System.out.println(savings);
}
}
```

## 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:

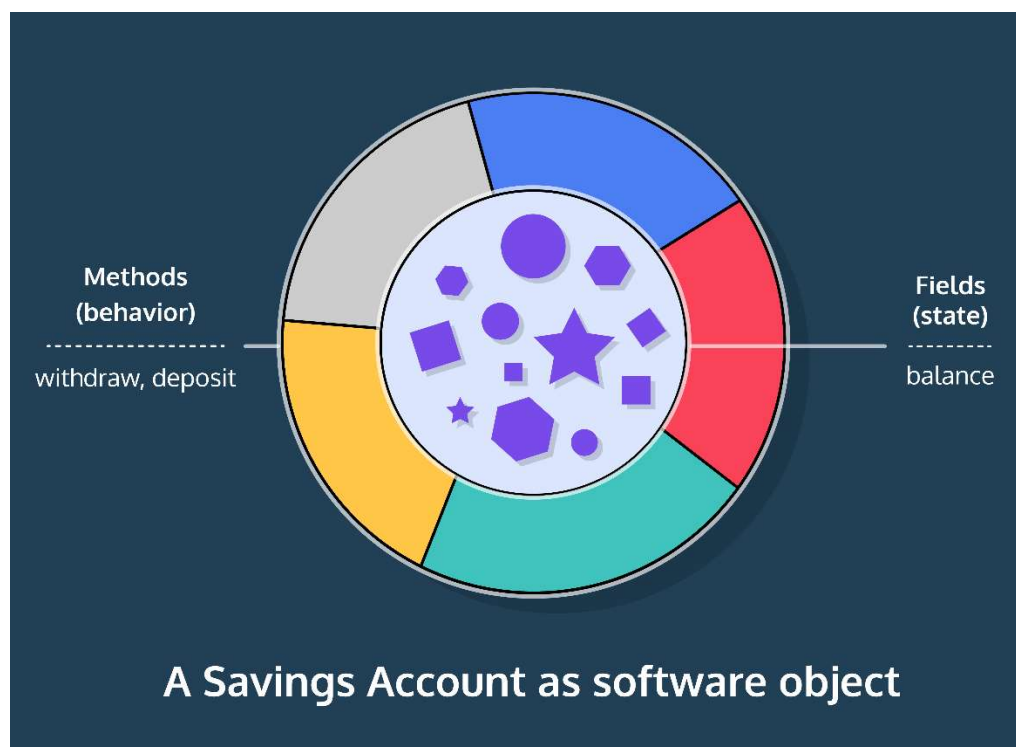
- Name of the owner
- Bank account number
- Amount of money in the account

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.

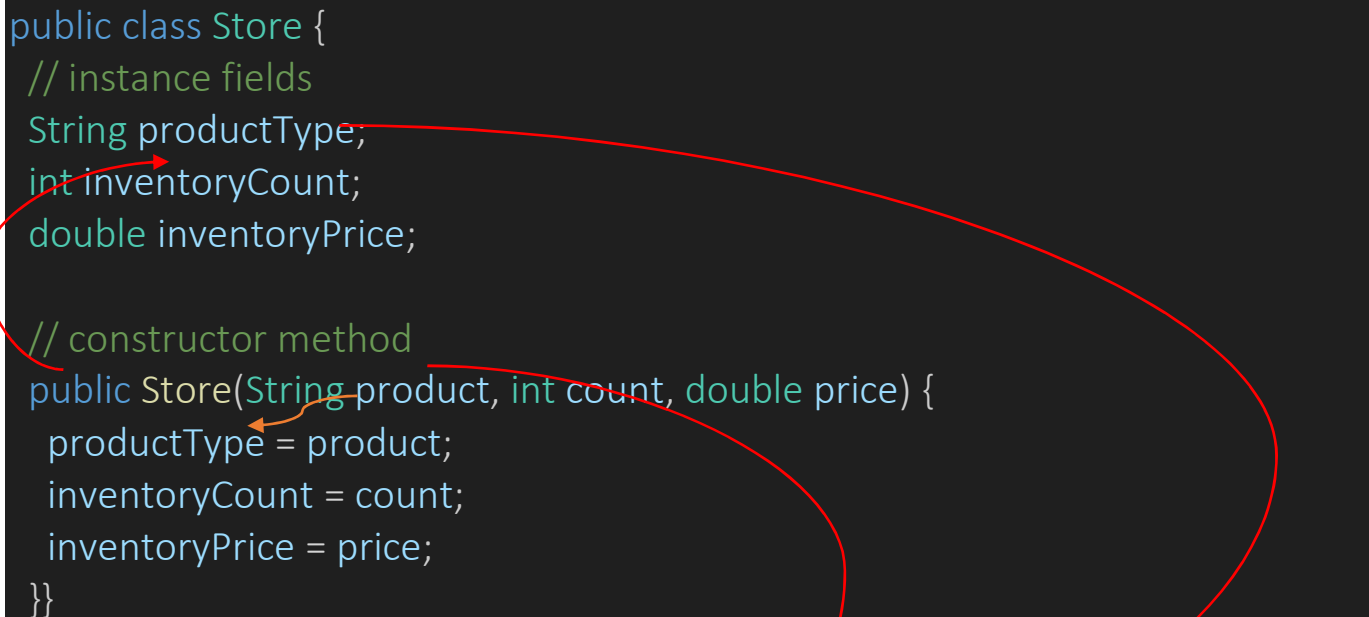
When we pass an argument, a copy of the argument value is passed to the parameter rather than the actual variables. This process of calling a method with an argument value is called a call-by-value.

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;  
    }  
}
```

A diagram with red arrows originates from the `productType` field in the `Store` class. One arrow points to the `productType` assignment in the constructor, and another points to the `lemonadeStand.productType` access in the `Main` class's `main` method.

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
    public Dog(String dogBreed) {
```

```
// value of parameter dogBreed assigned to instance field breed
breed = dogBreed;
}}
```

```
public static void main(String[] args) {
    // create instance: use 'new' operator and invoke constructor
    Dog fido = new Dog("poodle");

    // fields are accessed using: the instance name, `.` operator, and the field
    name.
    System.out.println(fido.breed); // Prints "poodle"
}
```

Example Let's make some modifications to our code.

Think about what instance fields you can create for the `Dog` class. Try to add and remove instance fields. Create instances with different values. Access and print different fields.

```
public class Dog {
    // fields
    String breed;
    boolean hasOwner;
    int age;

    public Dog(String dogBreed, boolean dogOwned, int dogYears) {
        // parameters are assigned to instance fields
        System.out.println("Constructor invoked!"); //2
        breed = dogBreed;
        hasOwner = dogOwned;
        age = dogYears;
    }

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

```
System.out.println("Main method started"); // 1

Dog fido = new Dog("poodle", false, 4);
// arguments are stored to constructors parameters
Dog nunzio = new Dog("shiba inu", true, 12);

boolean isFidoOlder = fido.age > nunzio.age;
// use of relational operators using (.) operators

int totalDogYears = nunzio.age + fido.age;
// object or instance variable name.instancefield

System.out.println("Two dogs created: a " + fido.breed + " and a " +
nunzio.breed);

System.out.println("The statement that fido is an older dog is: " +
isFidoOlder);

System.out.println("The total age of the dogs is: " + totalDogYears);

System.out.println("Main method finished");
}
}
```

## Output

```
Main method started
Constructor invoked!
Constructor invoked!

Two dogs created: a poodle and a shiba inu
The statement that fido is an older dog is: false

The total age of the dogs is: 16
Main method finished
```

# CONDITIONALS AND CONTROL FLOW

## Introduction to Control Flow

Imagine we're writing a program that enrolls students in courses.

- *If* a student has completed the prerequisites, *then* they can enroll in a course.
- *Else*, they need to take the prerequisite courses.

They can't take Physics II without finishing Physics I.

We represent this kind of decision-making in our program using conditional or *control flow* statements. Before this point, our code runs line-by-line from the top down, but conditional statements allow us to be selective in which portions will run.

Conditional statements check a **boolean** condition and run a *block* of code depending on the condition. Curly braces mark the scope of a conditional block similar to a method or class.

Here's a complete conditional statement:

```
if (true) {  
    System.out.println("Hello World!");  
}
```

If the condition is **true**, then the block is run. So **Hello World!** is printed.

But suppose the condition is different:

```
if (false) {  
    System.out.println("Hello World!");  
}
```

If the condition is **false**, then the block does not run.

This code is also called *if-then* statements: "If **(condition)** is **true**, then do something".

## If-Then Statement

The *if-then* statement is the most simple control flow we can write. It tests an expression for truth and executes some code based on it.

```
if (flip == 1) {  
  
    System.out.println("Heads!");  
  
}
```

- The **if** keyword marks the beginning of the conditional statement, followed by parentheses **()**.
- The parentheses hold a **boolean** datatype.

For the condition in parentheses we can also use [variables](#) that reference a boolean, or comparisons that evaluate to a boolean.

The boolean condition is followed by opening and closing curly braces that mark a block of code. This block runs if, and only if, the boolean is **true**.

```
boolean isValidPassword = true;  
  
if (isValidPassword) {  
    System.out.println("Password accepted!");  
}  
// Prints "Password accepted!"  
  
int numberOfItemsInCart = 9;  
  
if (numberOfItemsInCart > 12) {  
  
    System.out.println("Express checkout not available");  
  
}  
// Nothing is printed.
```

If a conditional is brief we can omit the curly braces entirely:

```
if (true) System.out.println("Brevity is the soul of wit");
```

**Note:** Conditional statements do not end in a semicolon.

The code editor contains an `Order` class to track retail shipments.

Write an if-then statement that prints `High value item!` when `itemCost` is greater than `24.00`.

```
public class Order {  
  
    public static void main(String[] args) {  
  
        double itemCost = 30.99;  
  
        // Write an if-then statement:  
        if (itemCost > 24.00){  
            System.out.println("High value item !");  
        }  
    }  
}
```

```
High value item !
```

## If-Then-Else

We've seen how to conditionally execute one block of code, but what if there are two possible blocks of code we'd like to execute?

Let's say *if* a student has the required prerequisite, *then* they enroll in the selected course, *else* they're enrolled in the prerequisite course instead.

We create an alternate conditional branch with the `else` keyword:

```
if (hasPrerequisite) {  
  // Enroll in course  
} else {  
  // Enroll in prerequisite  
}
```

This conditional statement ensures that exactly one code block will be run. If the condition, `hasPrerequisite`, is `false`, the block after `else` runs.

There are now two separate code blocks in our conditional statement. The first block runs if the condition evaluates to `true`, the second block runs if the condition evaluates to `false`.

This code is also called an *if-then-else* statement:

- If *condition* is true, then do something.
- Else, do a different thing.

## Instructions

1. In the code editor, there is an `isFilled` value, that represents whether the order is ready to ship.

Write an if-then-else statement that:

- When `isFilled` is `true`, print `Shipping`.
- When `isFilled` is `false`, print `Order not ready`.

```
Public class Order {  
  
    public static void main(String[] args) {  
  
        boolean isFilled = false;  
  
        // Write an if-then-else statement:  
  
        if (isFilled)  
        {  
            System.out.println("ready for shipping.");  
        }  
        else {  
            System.out.println("Order not ready .");  
        }  
    }  
}
```

Order not ready .



## If-Then-Else-If

The conditional structure we've learned can be chained together to check as many conditions as are required by our program. Imagine our program is now selecting the appropriate course for a student. We'll check their submission to find the correct course enrollment. The conditional statement now has multiple conditions that are evaluated from the top down:

```
String course = "Theatre";

if (course.equals("Biology")) {

    // Enroll in Biology course

} else if (course.equals("Algebra")) {

    // Enroll in Algebra course

} else if (course.equals("Theatre")) {

    // Enroll in Theatre course

} else {
    System.out.println("Course not found!");
}
```

The first condition to evaluate to **true** will have that code block run. Here's an example demonstrating the order:

```
int testScore = 72;

if (testScore >= 90) {

    System.out.println("A");

} else if (testScore >= 80) {

    System.out.println("B");

}
```

```
} else if (testScore >= 70) {  
    System.out.println("C");  
} else if (testScore >= 60) {  
    System.out.println("D");  
} else {  
    System.out.println("F");  
}  
// prints: C
```

This chained conditional statement has two conditions that evaluate **true**. Because **testScore >= 70** comes before **testScore >= 60**, only the earlier code block is run.

**Note:** Only one of the code blocks will run.

## Instructions

1. We need to calculate the shipping costs for our orders.

There's a new instance field, **String shipping**, that we use to calculate the cost.

Use a chained **if-then-else** to check for different values within the **calculateShipping()** method.

When the **shipping** instance field equals **"Regular"**, the method should return **0**.

When the **shipping** instance field equals **"Express"**, the method should return **1.75**.

Else the method should return **.50**.

```

public class Order {
    boolean isFilled;
    double billAmount;
    String shipping;

    public Order(boolean filled, double cost, String shippingMethod) {
        if (cost > 24.00) {
            System.out.println("High value item!");
        }
        isFilled = filled;
        billAmount = cost;
        shipping = shippingMethod;
    }

    public void ship() {
        if (isFilled) {
            System.out.println("Shipping");
            System.out.println("Shipping cost: " + calculateShipping());
        } else {
            System.out.println("Order not ready");
        }
    }

    public double calculateShipping() {
        // declare conditional statement here
        if (shipping.equals("Regular")){
            return 0;
        }else if(shipping.equals("Express")){
            return 1.75;
        } else {
            return 0.50;
        }
    }

    public static void main(String[] args) {
        // do not alter the main method!
        Order book = new Order(true, 9.99, "Express");
        Order chemistrySet = new Order(false, 72.50, "Regular");

        book.ship();
        chemistrySet.ship();
    }
}

```

```

High value item!
Shipping
Shipping cost: 1.75
Order not ready

```

## Nested Conditional Statements

We can create more complex conditional structures by creating *nested conditional statements*, which is created by placing conditional statements inside other conditional statements:

```
if (outer condition) {  
    if (nested condition) {  
        Instruction to execute if both conditions are true  
    }  
}
```

When we implement nested conditional statements, the outer statement is evaluated first. If the outer condition is **true**, then the inner, nested statement is evaluated.

Let's create a program that helps us decide what to wear based on the weather:

```
int temp = 45;  
boolean raining = true;  
  
if (temp < 60) {  
    System.out.println("Wear a jacket!");  
    if (raining == true) {  
        System.out.println("Bring your umbrella.");  
    } else {  
        System.out.println("Leave your umbrella home.");  
    }  
}
```

In the code snippet above, our [compiler](#) will check the condition in the first `if-then` statement: `temp < 60`. Since `temp` has a value of `45`, this condition is `true`; therefore, our program will print `Wear a jacket!`.

Then, we'll evaluate the condition of the nested `if-then` statement: `raining == true`. This condition is also `true`, so `Bring your umbrella` is also printed to the screen.

Note that, if the first condition was `false`, the nested condition would not be evaluated.

## Instructions

1. The company offers a temporary deal that, if the consumer uses the coupon `"ship50"`, the company will reduce the express shipping price.

Let's rewrite the body of `else-if` statement from the last exercise. Inside the `else-if` statement, create a nested `if-then` statement that checks if `couponCode` equals `"ship50"`.

If the nested condition is `true`, return the value `.85`.

If the condition is `false`, use a nested `else` statement to return the value `1.75`.

```
public class Order {
    boolean isFilled;
    double billAmount;
    String shipping;
    String couponCode;

    public Order(boolean filled, double cost, String shippingMethod, String coupon) {
        if (cost > 24.00) {
            System.out.println("High value item!");
        }
        isFilled = filled;
        billAmount = cost;
    }
}
```

```

shipping = shippingMethod;
couponCode = coupon;
}

public void ship() {
    if (isFilled) {
        System.out.println("Shipping");
        System.out.println("Shipping cost: " + calculateShipping());
    } else {
        System.out.println("Order not ready");
    }
}

public double calculateShipping() {
    if (shipping.equals("Regular")) {
        return 0;
    } else if (shipping.equals("Express")) {
        // Add your code here
        if (couponCode.equals("ship50")) {
            return .85;
        } else {
            return 1.75;
        }
    } else {
        return .50;
    }
}

public static void main(String[] args) {
    // do not alter the main method!
    Order book = new Order(true, 9.99, "Express", "ship50");
    Order chemistrySet = new Order(false, 72.50, "Regular", "freeShipping");

    book.ship();
    chemistrySet.ship();
}

```

High value item!  
Shipping  
Shipping cost: 0.85  
Order not ready

## Switch Statement

An alternative to chaining if-then-else conditions together is to use the `switch` statement. This conditional will check a given value against any number of conditions and run the code block where there is a match.

Here's an example of our course selection conditional as a `switch` statement instead:

```
String course = "History";

switch (course) {
    case "Algebra":
        // Enroll in Algebra
        break;
    case "Biology":
        // Enroll in Biology
        break;
    case "History":
        // Enroll in History
        break;
    case "Theatre":
        // Enroll in Theatre
        break;
    default:
        System.out.println("Course not found");
}
```

This example enrolls the student in History class by checking the value contained in the parentheses, `course`, against each of the `case` labels. If the value after the case label matches the value within the parentheses, the *switch block* is run.

In the above example, `course` references the string `"History"`, which matches `case "History":`.

When no value matches, the `default` block runs. Think of this as the `else` equivalent.

Switch blocks are different than other code blocks because they are not marked by curly braces and we use the `break` keyword to exit the switch statement.

Without `break`, code below the matching `case` label is run, *including code under other case labels*, which is rarely the desired behavior.

```
String course = "Biology";

switch (course) {
  case "Algebra":
    // Enroll in Algebra
  case "Biology":
    // Enroll in Biology
  case "History":
    // Enroll in History
  case "Theatre":
    // Enroll in Theatre
  default:
    System.out.println("Course not found");
}
// enrolls student in Biology... AND History and Theatre!
```

## Instructions

1. We'll rewrite the `calculateShipping()` method so it uses a `switch` statement instead.

There's an uninitialized variable `shippingCost` in `calculateShipping()`. Assign the correct value to `shippingCost` using a `switch` statement:

We'll check the value of the instance field `shipping`.

- When `shipping` matches `"Regular"`, `shippingCost` should be `0`.
- When `shipping` matches `"Express"`, `shippingCost` should be `1.75`.
- The default should assign `.50` to `shippingCost`.

**Make sure the method returns `shippingCost` after the `switch` statement.**



```

public class Order {
    boolean isFilled;
    double billAmount;
    String shipping;

    public Order(boolean filled, double cost, String shippingMethod) {
        if (cost > 24.00) {
            System.out.println("High value item!");
        }
        isFilled = filled;
        billAmount = cost;
        shipping = shippingMethod;
    }

    public void ship() {
        if (isFilled) {
            System.out.println("Shipping");
            System.out.println("Shipping cost: " + calculateShipping());
        } else {
            System.out.println("Order not ready");
        }
    }

    public double calculateShipping() {
        double shippingCost;
        // declare switch statement here
        switch (shipping) {
            case "Regular": shippingCost = 0; break;
            case "Express": shippingCost = 1.75; break;
            default:
                shippingCost = 0.50;
        }
        return shippingCost;
    }

    public static void main(String[] args) {
        // do not alter the main method!
        Order book = new Order(true, 9.99, "Express");
        Order chemistrySet = new Order(false, 72.50, "Regular");

        book.ship();
        chemistrySet.ship();
    }
}

```

High value item!  
 Shipping  
 Shipping cost: 1.75  
 Order not ready

## Review

Before this lesson, our code executed from top to bottom, line by line.

Conditional statements add branching paths to our programs. We use Conditionals to make decisions in the program so that different inputs will produce different results.

Conditionals have the general structure:

```
if (condition) {  
    // consequent path  
} else {  
    // alternative path  
}
```

Specific conditional statements have the following behavior:

**if-then**: code block runs if condition is true

**if-then-else**: one block runs if condition is true----another block runs if condition is false

**if-then-else** chained: same as **if-then** but an arbitrary number of conditions

**switch**: switch block runs if condition value matches **case** value

## Instructions

Our complete **Order** program is in the text editor but the **main()** method is empty.

Create different **Order** instances and see if you can run the code in all the different conditional blocks!

```
public class Order {  
    boolean isFilled;  
    double billAmount;  
    String shipping;  
  
    public Order(boolean filled, double cost, String shippingMethod) {
```

```

if (cost > 24.00) {
    System.out.println("High value item!");
} else {
    System.out.println("Low value item!");
}
isFilled = filled;
billAmount = cost;
shipping = shippingMethod;
}

public void ship() {
    if (isFilled) {
        System.out.println("Shipping");
    } else {
        System.out.println("Order not ready");
    }

    double shippingCost = calculateShipping();

    System.out.println("Shipping cost: ");
    System.out.println(shippingCost);
}

public double calculateShipping() {
    double shippingCost;
    switch (shipping) {
        case "Regular":
            shippingCost = 0;
            break;
        case "Express":
            shippingCost = 1.75;
            break;
        default:
            shippingCost = .50;
    }
    return shippingCost;
}

public static void main(String[] args) {
    // create instances and call methods here!
    Order o1 = new Order(false, 35.00, "Express");
    o1.calculateShipping();
    o1.ship();
}}

```

High value item!  
Order not ready  
Shipping cost:  
1.75

# CONDITIONAL OPERATORS

## Introduction to Conditional Operators

Java includes [operators](#) that only use boolean values.

These *conditional operators* help simplify expressions containing complex boolean relationships by reducing multiple boolean values to a single value: **true** or **false**.

For example, what if we want to run a code block only if *multiple* conditions are true. We could use the *AND* operator: **&&**.

A	B	A && B
True	True	True
True	False	False
False	True	False
False	False	False

Or, we want to run a code block if *at least one* of two conditions are **true**. We could use the *OR* operator: **||**.

A	B	A    B
True	True	True
True	False	True
False	True	True
False	False	False

Finally, we can produce the opposite value, where **true** becomes **false** and **false** becomes **true**, with the *NOT* operator: **!**.

A	!A
True	False
False	True

Understanding these complex relationships can feel overwhelming at first. Luckily, *truth tables*, like the ones seen to the right, can assist us in determining the relationship between two boolean-based conditions.

In this lesson, we'll explore each of these conditional operators to see how they can be implemented into our conditional statements.

## Instructions

The text editor contains a **Reservation** class we'll build in this lesson.

Note the different conditional statements and operators that we're using to control the execution of the program.

Move on when you're ready!

```
public class Reservation {
    int guestCount;
    int restaurantCapacity;
    boolean isRestaurantOpen;
    boolean isConfirmed;

    public Reservation(int count, int capacity, boolean open) {
        if (count < 1 || count > 8) {
            System.out.println("Invalid reservation!");
        }
        guestCount = count;
        restaurantCapacity = capacity;
        isRestaurantOpen = open;
    }

    public void confirmReservation() {
        if (restaurantCapacity >= guestCount && isRestaurantOpen) {
            System.out.println("Reservation confirmed");
        }
    }
}
```

```

        isConfirmed = true;
    } else {
        System.out.println("Reservation denied");
        isConfirmed = false;
    }
}

public void informUser() {
    if (!isConfirmed) {
        System.out.println("Unable to confirm reservation, please
contact restaurant.");
    } else {
        System.out.println("Please enjoy your meal!");
    }
}

public static void main(String[] args) {
    Reservation partyOfThree = new Reservation(3, 12, true);
    Reservation partyOfFour = new Reservation(4, 3, true);
    partyOfThree.confirmReservation();
    partyOfThree.informUser();
    partyOfFour.confirmReservation();
    partyOfFour.informUser();
}
}

```

```

Reservation confirmed
Please enjoy your meal!
Reservation denied
Unable to confirm reservation, please contact restaurant.

```

## Conditional-And: &&

Let's return to our student enrollment program. We've added an additional requirement: not only must students have the prerequisite, but their tuition must be paid up as well. We have two conditions that must be **true** before we enroll the student.

Here's one way we could write the code:

```
if (tuitionPaid) {  
  if (hasPrerequisite) {  
    // enroll student  
  }  
}
```

We've nested two **if-then** statements. This does the job but we can be more concise with the **AND** operator:

```
if (tuitionPaid && hasPrerequisite) {  
  // enroll student  
}
```

The AND operator, **&&**, is used between two boolean values and evaluates to a single boolean value. If the values **on both sides** are **true**, then the resulting value is **true**, otherwise the resulting value is **false**.

This code illustrates every combination:

```
true && true  
// true  
false && true  
// false  
true && false  
// false  
false && false  
// false
```

## Instructions

1. Our `Reservation` class has the method `confirmReservation()` which validates if a restaurant can accommodate a given reservation.

We need to build the conditional logic into `confirmReservation()` using three instance variables:

- `restaurantCapacity`
- `guestCount`
- `isRestaurantOpen`

Use an `if-then-else` statement:

If `restaurantCapacity` is greater than or equal to `guestCount` and the restaurant is open, print `"Reservation confirmed"` and set `isConfirmed` to `true`.

`else` print `"Reservation denied"` and set `isConfirmed` to `false`.

**Note:** For now, the `informUser()` method will always print `"Please enjoy your meal"` even if the reservation was not confirmed. We will modify this method in an upcoming exercise!

```
public class Reservation {  
    int guestCount;  
    int restaurantCapacity;  
    boolean isRestaurantOpen;  
    boolean isConfirmed;  
  
    public Reservation(int count, int capacity, boolean open) {  
        guestCount = count;  
        restaurantCapacity = capacity;  
        isRestaurantOpen = open;  
    }  
  
    public void confirmReservation() {  
        /*
```



Write conditional

~~~~~

if restaurantCapacity is greater  
or equal to guestCount

AND

the restaurant is open:

print "Reservation confirmed"

set isConfirmed to true

else:

print "Reservation denied"

set isConfirmed to false

\*/

```
if (restaurantCapacity >= guestCount && isRestaurantOpen) {  
    System.out.println("Reservation Confirmed.");  
    isConfirmed = true;
```

```
}else {  
    System.out.println("Reservation denied.");  
    isConfirmed = false;  
}  
}
```

```
public void informUser() {  
    System.out.println("Please enjoy your meal!");  
}
```

```
public static void main(String[] args) {  
    Reservation partyOfThree = new Reservation(3, 12, true);  
    Reservation partyOfFour = new Reservation(4, 3, true);  
    partyOfThree.confirmReservation();  
    partyOfThree.informUser();  
    partyOfFour.confirmReservation();  
    partyOfFour.informUser();  
}
```

Output:

```
Reservation Confirmed.  
Please enjoy your meal!  
Reservation denied.  
Please enjoy your meal!
```

## Conditional-Or: ||

The requirements of our enrollment program have changed again. Certain courses have prerequisites that are satisfied by multiple courses. As long as students have taken **at least one** prerequisite, they should be allowed to enroll.

Here's one way we could write the code:

```
if (hasAlgebraPrerequisite) {  
    // Enroll in course  
}  
if (hasGeometryPrerequisite) {  
    // Enroll in course  
}
```

We're using two different **if-then** statements with **the same code block**. We can be more concise with the **OR** operator:

```
if (hasAlgebraPrerequisite || hasGeometryPrerequisite) {  
    // Enroll in course  
}
```

The OR operator, **||**, is used between two boolean values and evaluates to a single boolean value. If **either of the two values** is **true**, then the resulting value is **true**, otherwise the resulting value is **false**.

This code illustrates every combination:

```
true || true  
// true  
false || true  
// true  
true || false  
// true  
false || false  
// false
```

## Keep Reading: AP Computer Science A Students

On some occasions, the [compiler](#) can determine the truth value of a logical expression by only evaluating the first **boolean** operand; this is known as *short-circuited evaluation*. Short-circuited evaluation only works with expressions that use **&&** or **||**.

In an expression that uses **||**, the resulting value will be **true** as long as one of the operands has a **true** value. If the first operand of an expression is **true**, we don't need to see what the value of the other operand is to know that the final value will also be **true**.

For example, we can run the following code without error despite dividing a number by **0** in the second operand because the first operand had a **true** value:

```
if (1 > 0 || 2 / 0 == 7) {  
    System.out.println("No errors here!");  
}
```

An expression that uses **&&** will only result in **true** if both operands are **true**. If the first operand in the expression is **false**, the entire value will be **false**.

### Instructions

1. Let's write a message inside the **Reservation()** constructor that warns against bad input.

Our restaurants can't seat parties of more than **8** people, and we don't want reservations for **0** or less because that would be silly.

Inside **Reservation()**, write a conditional that uses **||**.

If **count** is less than **1** **OR** greater than **8** we want to write the following message: **Invalid reservation!**

```
public class Reservation {  
    int guestCount;
```

```

int restaurantCapacity;
boolean isRestaurantOpen;
boolean isConfirmed;

public Reservation(int count, int capacity, boolean open) {
    // Write conditional statement below
    if (count < 1 || count > 8){
        System.out.println("Invalid reservation!.");
    }

    guestCount = count;
    restaurantCapacity = capacity;
    isRestaurantOpen = open;
}

public void confirmReservation() {
    if (restaurantCapacity >= guestCount && isRestaurantOpen) {
        System.out.println("Reservation confirmed");
        isConfirmed = true;
    } else {
        System.out.println("Reservation denied");
        isConfirmed = false;
    }
}

public void informUser() {
    System.out.println("Please enjoy your meal!");
}

public static void main(String[] args) {
    Reservation partyOfThree = new Reservation(3, 12, true);
    Reservation partyOfFour = new Reservation(4, 3, true);
    partyOfThree.confirmReservation();
    partyOfThree.informUser();
    partyOfFour.confirmReservation();
    partyOfFour.informUser();
}

```

```

Reservation confirmed
Please enjoy your meal!
Reservation denied
Please enjoy your meal!

```

## Logical NOT: !

The *unary* operator NOT, **!**, works on a **single** value. This operator evaluates to the opposite boolean to which it's applied:

```
!false  
// true  
!true  
// false
```

NOT is useful for expressing our intent clearly in programs. For example, sometimes we need the opposite behavior of an **if-then**: run a code block **only** if the condition is **false**.

```
boolean hasPrerequisite = false;  
  
if (hasPrerequisite) {  
    // do nothing  
} else {  
    System.out.println("Must complete prerequisite course!");  
}
```

This code does what we want but it's strange to have a code block that does nothing!

The logical NOT operator cleans up our example:

```
boolean hasPrerequisite = false;  
  
if (!hasPrerequisite) {  
    System.out.println("Must complete prerequisite course!");  
}
```

We can write a succinct conditional statement without an empty code block.

### Instructions

1. Let's make **informUser()** more informative. If their reservation is not confirmed, they should know!

Write an **if-then-else** statement and use **!** with **isConfirmed** as the

condition.

If their reservation is **not** confirmed, write **Unable to confirm reservation, please contact restaurant.**

Else write: **Please enjoy your meal!**

```
public class Reservation {
    int guestCount;
    int restaurantCapacity;
    boolean isRestaurantOpen;
    boolean isConfirmed;

    public Reservation(int count, int capacity, boolean open) {
        if (count < 1 || count > 8) {
            System.out.println("Invalid reservation!");
        }
        guestCount = count;
        restaurantCapacity = capacity;
        isRestaurantOpen = open;
    }

    public void confirmReservation() {
        if (restaurantCapacity >= guestCount && isRestaurantOpen) {
            System.out.println("Reservation confirmed");
            isConfirmed = true;
        } else {
            System.out.println("Reservation denied");
            isConfirmed = false;
        }
    }

    public void informUser() {
        // Write conditional here
        if(!isConfirmed){
```

```
    System.out.println("Unable to confirm reservation, please contact  
restaurant.");  
}else {  
    System.out.println("Please enjoy your meal!");  
}  
}  
  
public static void main(String[] args) {  
    Reservation partyOfThree = new Reservation(3, 12, true);  
    Reservation partyOfFour = new Reservation(4, 3, true);  
    partyOfThree.confirmReservation();  
    partyOfThree.informUser();  
    partyOfFour.confirmReservation();  
    partyOfFour.informUser();  
}  
}
```

Reservation confirmed  
Please enjoy your meal!  
Reservation denied  
Unable to confirm reservation, please contact restaurant

## Combining Conditional Operators

We have the ability to expand our boolean expressions by using multiple conditional [operators](#) in a single expression.

For example:

```
boolean foo = true && !(false || !true)
```

How does an expression like this get evaluated by the compiler? The order of evaluation when it comes to conditional operators is as follows:

1. Conditions placed in parentheses - `()`
2. NOT - `!`
3. AND - `&&`
4. OR - `||`

Using this information, let's dissect the expression above to find the value of `foo`:

```
true && !(false || !true)
```

First, we'll evaluate `(false || !true)` because it is enclosed within parentheses. Following the order of evaluation, we will evaluate `!true`, which equals `false`:

```
true && !(false || false)
```

Then, we'll evaluate `(false || false)` which equals `false`. Now our expression looks like this:

```
true && !false
```

Next, we'll evaluate `!false` because it uses the NOT operator. This expression equals `true` making our expression the following:



true && true

true && true evaluates to true; therefore, the value of foo is true.

## Instructions

Take a look at the three expressions in **Operators.java**.

Using your understanding of the order of execution, find out whether the value of each expression is true or false.

When you're ready, uncomment the print statements to find out if you are right.

```
public class Operators {  
    public static void main(String[] args) {  
        int a = 6;  
        int b = 3;  
  
        boolean ex1 = !(a == 7 && (b >= a || a != a));  
        System.out.println(ex1);  
  
        boolean ex2 = a == b || !(b > 3);  
        System.out.println(ex2);  
  
        boolean ex3 = !(b <= a && b != a + b);  
        System.out.println(ex3);  
    }  
}
```

true  
true  
false

## Review

Conditional [operators](#) work on boolean values to simplify our code. They're often combined with conditional statements to consolidate the branching logic.

Conditional-AND, `&&`, evaluates to `true` if the booleans on both sides are `true`.

```
if (true && false) {  
    System.out.println("You won't see me print!");  
} else if (true && true) {  
    System.out.println("You will see me print!");  
}
```

Conditional-OR, `||`, evaluates to `true` if one or both of the booleans on either side is `true`.

```
if (false || false) {  
    System.out.println("You won't see me print!");  
} else if (false || true) {  
    System.out.println("You will see me print!");  
}
```

Logical-NOT, `!`, evaluates to the opposite boolean value to which it is applied.

```
if (!false) {  
    System.out.println("You will see me print!");  
}
```

```
public class Reservation {
    int guestCount;
    int restaurantCapacity;
    boolean isRestaurantOpen;
    boolean isConfirmed;

    public Reservation(int count, int capacity, boolean open) {
        if (count < 1 || count > 8) {
            System.out.println("Invalid reservation!");
        }
        guestCount = count;
        restaurantCapacity = capacity;
        isRestaurantOpen = open;
    }

    public void confirmReservation() {
        if (restaurantCapacity >= guestCount && isRestaurantOpen) {
            System.out.println("Reservation confirmed");
            isConfirmed = true;
        } else {
            System.out.println("Reservation denied");
            isConfirmed = false;
        }
    }

    public void informUser() {
        if (!isConfirmed) {
            System.out.println("Unable to confirm reservation, please contact restaurant.");
        } else {
            System.out.println("Please enjoy your meal!");
        }
    }
}
```

```
public static void main(String[] args) {  
    // Create i  
    Reservation groupOf3 = new Reservation(3, 12, true);  
    groupOf3.confirmReservation();  
    groupOf3.informUser();  
  
    Reservation groupOf4 = new Reservation(4, 03, true);  
    groupOf4.confirmReservation();  
    groupOf4.informUser();  
  
}  
}
```

Reservation confirmed

Please enjoy your meal!

Reservation denied

Unable to confirm reservation, please contact restaurant.

# ARRAYS

## Introduction

We have seen how to store single pieces of data in variables. What happens when we need to store a group of data? What if we have a list of students in a classroom? Or a ranking of the top 10 horses finishing a horse race?

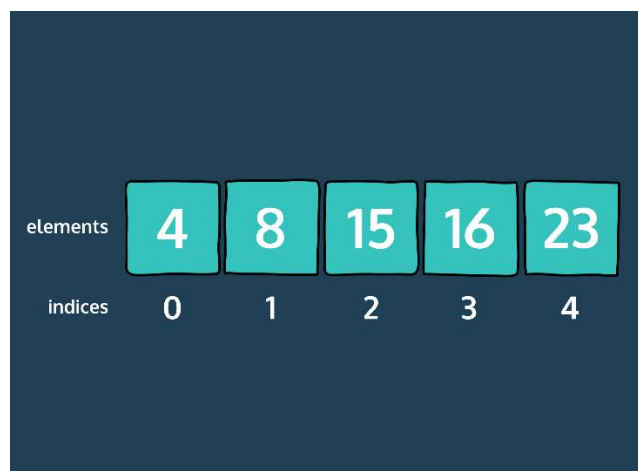
If we were storing 5 lottery ticket numbers, for example, we could create a different variable for each value:

```
int firstNumber = 4;  
int secondNumber = 8;  
int thirdNumber = 15;  
int fourthNumber = 16;  
int fifthNumber = 23;
```

That is a lot of ungainly repeated code. What if we had a hundred lottery numbers? It is more clean and convenient to use a Java array to store the data as a list.

An array holds a fixed number of values of one type. Arrays hold **doubles**, **ints**, **booleans**, or any other primitives. Arrays can also contain **Strings** as well as object references!

Each index of an array corresponds with a different value. Here is a diagram of an array filled with integer values:



Notice that the indexes start at 0! The element at index 0 is 4, while the element at index 1 is 8. This array has a length of 5, since it holds five elements, but the highest index of the array is 4.

Let's explore how to create and use arrays in Java, so that we can store all of our Java data types.

## Instructions

1. In the code editor, we have a **Newsfeed** class to manage trending articles with their views and ratings.

Throughout this lesson, you'll learn how to create such Java programs with Java arrays.

Run the code and see how the arrays are used.

```
public class Main{
    public static void main(String[] args){
        String[] robotArticles = {"Oil News", "Innovative Motors",
        "Humans: Exterminate Or Not?", "Organic Eye Implants", "Path
        Finding in an Unknown World"};
        int[] robotViewers = {87, 32, 13, 11, 7};
        double[] robotRatings = {2.5, 3.2, 5.0, 1.7, 4.3};
        Newsfeed robotTimes = new Newsfeed(robotArticles,
        robotViewers, robotRatings);

        robotTimes.viewArticle(2);
        robotTimes.viewArticle(2);
        System.out.println("The top article is " +
        robotTimes.getTopArticle());
        robotTimes.changeRating(3, 5);
    }
}
```

The article 'Humans: Exterminate Or Not?' has now been viewed 14 times!

The article 'Humans: Exterminate Or Not?' has now been viewed 15 times!

The top article is Oil News

The article 'Organic Eye Implants' is now rated 5.0 stars!

## Creating an Array Explicitly

Imagine that we're using a program to keep track of the prices of different clothing items we want to buy. We would want a list of the prices and a list of the items they correspond to. To create an array, we provide a name and declare the type of data it holds:

```
double[] prices;
```

Just like with [variables](#), we can declare and initialize in the same line. This allows us to explicitly initialize the array to contain the data we want to store :

```
double[] prices = {13.15, 15.87, 14.22, 16.66};
```

We can use [arrays](#) to hold **Strings** and other objects as well as primitives:

```
String[] clothingItems = {"Tank Top", "Beanie", "Funny Socks",  
"Corduroys"};
```

## Instructions

1. For now, our **Newsfeed** class doesn't have any methods.

Create a method **getTopics()** that accepts no parameters, returns a **String** array, and is accessible by other classes.

Leave the body empty.

Note: it is okay for there to be an error claiming there is no `main()` method. This method is defined in **Main.java**.

2. Inside the `getTopics()` method, create a `String` array called `topics` that contains these elements:

"Opinion", "Tech", "Science", "Health"

Remember to keep the order the same.

Then, return the `topics` array at the end of the method!

To see the output of the program, switch over to the **Main.java** file and click "Run".

```
public class Newsfeed {  
    public Newsfeed(){  
  
    }  
    // Create getTopics() below:  
    public String[] getTopics(){  
        String[] topics = {"Opinion", "Tech", "Science", "Health"};  
        return topics;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Newsfeed sampleFeed = new Newsfeed();  
        String[] topics = sampleFeed.getTopics();  
        System.out.println(topics);  
    }  
}
```

```
[Ljava.lang.String;@7ad041f3
```



## Importing Arrays

When we printed out the array we created in the last exercise, we saw a memory address that did not help us understand what was contained in the array.

If we want to have a more descriptive printout of the array itself, we need a `toString()` method that is provided by the `Arrays` package in Java.

```
import java.util.Arrays;
```

We put this at the top of the file, before we even define the class!

When we import a package in Java, we are making all of the [methods](#) of that package available in our code.

The `Arrays` package has many useful methods, including `Arrays.toString()`. When we pass an array into `Arrays.toString()`, we can see the contents of the array printed out:

```
import java.util.Arrays;

public class Lottery(){

    public static void main(String[] args){
        int[] lotteryNumbers = {4, 8, 15, 16, 23, 42};
        String betterPrintout = Arrays.toString(lotteryNumbers);
        System.out.println(betterPrintout);
    }
}
```

This code will print:

```
[4, 8, 15, 16, 23, 42]
```

## Instructions

1.If you run the code now, You'll see something like this: `[Ljava.lang.String;@2aae9190`.

This is the memory address of the array and not the actual `String` array.

To make the code print the actual elements, use the `toString()` method from the `Arrays` package.

Import the `Arrays` package from `java.util` at the top of the `Main.java` file.

2.Now, use the `toString()` method from the `Arrays` package to print array `topics` in the `main()` method of `Main.java`.

```
public class Newsfeed {  
    public String[] getTopics(){  
        String[] topics = {"Opinion", "Tech", "Science", "Health"};  
        return topics;  
    }  
}
```

```
// import the Arrays package here
```

```
import java.util.Arrays;
```

```
public class Main {
```

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

```
        Newsfeed sampleFeed = new Newsfeed();
```

```
        String[] topics = sampleFeed.getTopics();
```

```
        System.out.println(topics);
```

```
        System.out.println(Arrays.toString(topics));
```

```
    }  
}
```

```
[Ljava.lang.String;@7ad041f3  
[Opinion, Tech, Science, Health]
```

## Get Element By Index

Now that we have an array declared and initialized, we want to be able to get values out of it.

We use square brackets, `[` and `]`, to access data at a certain index:

```
double[] prices = {13.1, 15.87, 14.22, 16.66};  
  
System.out.println(prices[1]);
```

This command would print:

```
15.87
```

This happens because `15.87` is the item at the `1` index of the array. Remember, the index of an array starts at `0` and ends at an index of one less than the number of elements in the array.

If we try to access an element outside of its appropriate index range, we will receive an `ArrayIndexOutOfBoundsException` error.

For example, if we were to run the command `System.out.println(prices[5])`, we'd get the following output: `java.lang.ArrayIndexOutOfBoundsException: 5`

## Instructions

1. Inside the `Newsfeed` class, create a method called `getFirstTopic()` that returns a `String` and accepts no parameters.

Inside the `getFirstTopic()` method, return the first element of the `topics` array.

2. We have added an array called `views` to keep track of how many viewers visit a topic.

Every time someone views a topic, we want to increase the value of the corresponding element in `views` by 1.

For example, if someone views an "Opinion" piece (index of 0 in topics), we will increase the value of the 0<sup>th</sup> index of `views` by 1.

Inside the `viewTopic()` method, implement this functionality. The parameter `topicIndex` represents the location of the element in `topics` that was viewed.

*Note: switch over to the `Main.java` file to see the output of the program you wrote.*

```
public class Newsfeed {
    String[] topics = {"Opinion", "Tech", "Science", "Health"};
    public int[] views = {0, 0, 0, 0};

    public Newsfeed(){

    }
    public String[] getTopics(){
        return topics;
    }

    public String getFirstTopic(){
        return topics[0];
    }
    public void viewTopic(int topicIndex){
        views[topicIndex]++;
    }
}
```

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        Newsfeed sampleFeed = new Newsfeed();
    }
}
```

```
System.out.println("The top topic is " +  
sampleFeed.getFirstTopic());
```

```
sampleFeed.viewTopic(1);  
sampleFeed.viewTopic(1);  
sampleFeed.viewTopic(3);  
sampleFeed.viewTopic(2);  
sampleFeed.viewTopic(2);  
sampleFeed.viewTopic(1);
```

```
System.out.println("The " + sampleFeed.topics[1] + " topic has  
been viewed " + sampleFeed.views[1] + " times!");  
}  
}
```

The top topic is Opinion

The Tech topic has been viewed 3 times!

## Creating an Empty Array

We can also create empty [arrays](#) and then fill the items one by one. Empty arrays have to be initialized with a fixed size:

```
String[] menuItems = new String[5];
```

Once you declare this size, it cannot be changed! This array will always be of size **5**.

After declaring and initializing, we can set each index of the array to be a different item:

```
menuItems[0] = "Veggie hot dog";  
menuItems[1] = "Potato salad";  
menuItems[2] = "Cornbread";  
menuItems[3] = "Roasted broccoli";  
menuItems[4] = "Coffee ice cream";
```

This group of commands has the same effect as assigning the entire array at once:

```
String[] menuItems = {"Veggie hot dog", "Potato salad",  
"Cornbread", "Roasted broccoli", "Coffee ice cream"};
```

We can also change an item after it has been assigned! Let's say this restaurant is changing its broccoli dish to a cauliflower one:

```
menuItems[3] = "Baked cauliflower";
```

Now, the array looks like:

```
["Veggie hot dog", "Potato salad", "Cornbread", "Baked cauliflower",  
"Coffee ice cream"]
```

## Keep Reading: AP Computer Science A Students

When we use `new` to create an empty array, each element of the array is initialized with a specific value depending on what type the element is:

### Data Type    Initialized Value

`int`            `0`

`double`       `0.0`

`boolean`      `false`

`Reference`    `null`

For example, consider the following arrays:

```
String[] my_names = new String[5];  
int[] my_ages = new int[5];
```

Because a `String` is a reference to an Object, `my_names` will contain five `nulls`. `my_ages` will contain five `0s` to begin with.

### Instructions

1. We've declared a `String` array called `favoriteArticles` as an instance field.

We'll keep track of the user's top 10 favorite articles in this string array.

In the constructor, `Newsfeed()`, initialize `favoriteArticles` as a new empty `String` array of size 10.

2. We have created a `setFavoriteArticle()` method that accepts `favoriteIndex` and `newArticle` as parameters.

Inside `setFavoriteArticle()`, set the value of the `favoriteArticles` array at index `favoriteIndex` to the value of `newArticle`.

For example, if we called `setFavoriteArticle(2, "Celebrity Hands Throughout the Decades")`, the value of `favoriteArticles` at index 2 would be set to "Celebrity Hands Throughout the Decades".

Switch over to `Main.java` and run the code.

```
public class Newsfeed {
    String[] topics = {"Opinion", "Tech", "Science", "Health"};
    int[] views = {0, 0, 0, 0};
    String[] favoriteArticles;

    public Newsfeed(){
        // Initialize favoriteArticles here:
        favoriteArticles = new String[10];
    }

    public void setFavoriteArticle(int favoriteIndex, String
newArticle){
        // Add newArticle to favoriteArt
        favoriteArticles[favoriteIndex] = newArticle;
    }
}
```

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        Newsfeed sampleFeed = new Newsfeed();

        sampleFeed.setFavoriteArticle(2, "Humans: Exterminate Or
Not?");
        sampleFeed.setFavoriteArticle(3, "Organic Eye Implants");
        sampleFeed.setFavoriteArticle(0, "Oil News");

        System.out.println(Arrays.toString(sampleFeed.favoriteArticles));
    }
}
```



```
[Oil News, null, Humans: Exterminate Or Not?, Organic Eye  
Implants, null, null, null, null, null]
```

## Length

What if we have an array storing all the usernames for our program, and we want to quickly see how many users we have? To get the length of an array, we can access the `length` field of the array object:

```
String[] menuItems = new String[5];  
System.out.println(menuItems.length);
```

This command would print `5`, since the `menuItems` array has `5` slots, even though they are all empty.

If we print out the length of the `prices` array:

```
double[] prices = {13.1, 15.87, 14.22, 16.66};  
  
System.out.println(prices.length);
```

We would see `4`, since there are four items in the `prices` array!

## Instructions

1. We have created a method `getNumTopics()` to fetch how many topics exist in the array.

Inside `getNumTopics()`, return the length of the `topics` array.

```
public class Newsfeed {  
    String[] topics = {"Opinion", "Tech", "Science", "Health"};  
    int[] views = {0, 0, 0, 0};  
  
    public Newsfeed(){  
  
    }  
    public String[] getTopics(){  
        return topics;  
    }  
}
```

```
}  
public int getNumTopics(){  
    return topics.length;  
}  
}
```

```
import java.util.Arrays;  
public class Main {  
    public static void main(String[] args){  
        Newsfeed sampleFeed = new Newsfeed();  
        System.out.println("The number of topics is "+  
sampleFeed.getNumTopics());  
    }  
}
```

The number of topics is 4

## String[] args

When we write `main()` [methods](#) for our programs, we use the parameter `String[] args`. Now that we know about array syntax, we can start to parse what this means.

A `String[]` is an array made up of `Strings`. Examples of `String` arrays:

```
String[] humans = {"Talesha", "Gareth", "Cassie", "Alex"};  
String[] robots = {"R2D2", "Marvin", "Bender", "Ava"};
```

The `args` parameter is another example of a `String` array. In this case, the array `args` contains the arguments that we pass in from

the terminal when we run the class file. (So far, `args` has been empty.)

So how can you pass arguments to `main()`? Let's say we have this class `HelloYou`:

```
public class HelloYou {  
    public static void main(String[] args) {  
        System.out.println("Hello " + args[0]);  
    }  
}
```

When we run the file `HelloYou` in the terminal with an argument of `"Laura"`:

```
java HelloYou Laura
```

We get the output:

```
Hello Laura
```

The `String[] args` would be interpreted as an array with one element, `"Laura"`.

When we use `args[0]` in the main method, we can access that element like we did in `HelloYou`.

### Instructions

1. Let's make users have the option to personalize the Newsfeed object for either robots or humans.

The `main()` in `Main.java` will take either "Robot" or "Human" as an argument when we run the file.

If the `args[0]` array holds "Human", we will initialize the feed with human topics.

If the `args[0]` array holds "Robot", we will initialize the feed with robot topics.

In the conditional statement on line 6, Replace the blank to check if the `args` has the value `Robot` or not.

2. In the terminal, run the `Main.java` with the argument `Robot`.

*Note: Do not use quotation marks for the argument. Passing in "Robot" as an argument will not work.*

3. Now, run the `Main.java` file with the argument `Human`.

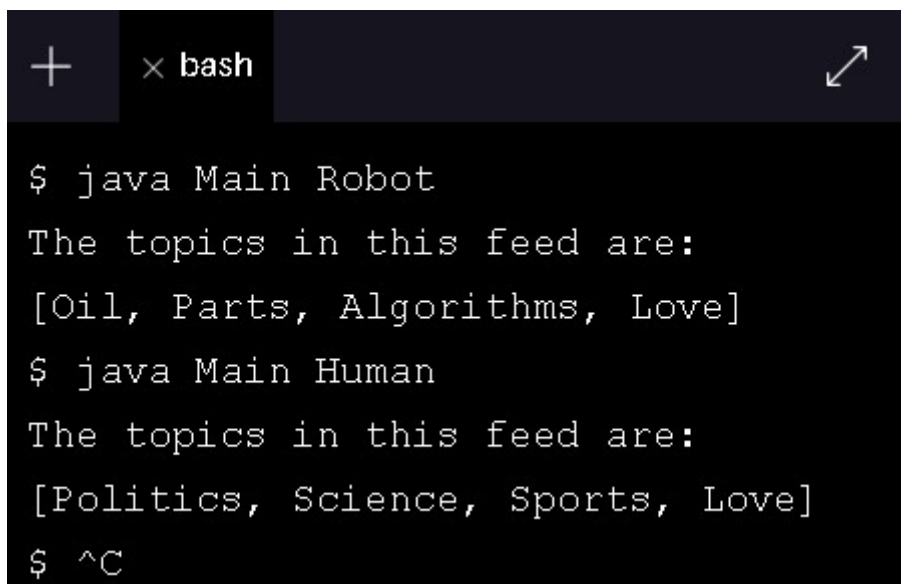
*Note: Do not use quotation marks for the argument. Passing in "Human" as an argument will not work.*

```
public class Newsfeed {  
  
    String[] topics;  
  
    public Newsfeed(String[] initialTopics){  
        topics = initialTopics;  
    }  
}
```

```
import java.util.Arrays;  
  
public class Main{  
    public static void main(String[] args){  
        Newsfeed feed;  
        if(args[0].equals("Robot")){  
            //topics for a Robot feed:  
            String[] robotTopics = {"Oil", "Parts", "Algorithms", "Love"};  
            feed = new Newsfeed(robotTopics);  
        }  
        else if(args[0].equals("Human")){  
            //topics for a Human feed:  
            String[] humanTopics = {"Politics", "Science", "Sports", "Love"};
```

```
    feed = new Newsfeed(humanTopics);
}
else{
    String[] genericTopics = {"Opinion", "Tech", "Science",
"Health"};
    feed = new Newsfeed(genericTopics);
}

System.out.println("The topics in this feed are:");
System.out.println(Arrays.toString(feed.topics));
}
}
```



```
+ x bash ↗

$ java Main Robot
The topics in this feed are:
[Oil, Parts, Algorithms, Love]
$ java Main Human
The topics in this feed are:
[Politics, Science, Sports, Love]
$ ^C
```

## Review

We have now seen how to store a list of values in [arrays](#). We can use this knowledge to make organized programs with more complex [variables](#).

Throughout the lesson, we have learned about:

- Creating arrays explicitly, using `{` and `}`.
- Accessing an index of an array using `[` and `]`.
- Creating empty arrays of a certain size, and filling the indices one by one.

- Getting the length of an array using `length`.
- Using the argument array `args` that is passed into the `main()` method of a class.

## Instructions

1. Let's practice what we've learned throughout this lesson.

First, inside `main()` in `Main.java`, create a String array `students` and initialize it with these elements:

- "Sade"
- "Alexus"
- "Sam"
- "Koma"

2. Now, to store the scores of the recent math test, create an array called `mathScores` of type `int` and set it to an empty array of size 4.

Each spot in this array will represent the math test score of the student in the corresponding spot in the `students` array. For example: `mathScores[2]` will represent the score `students[2]` which is "Sam".

3. The students had the following scores on their most recent math test:

- Sade got a 64
- Sam got a 76
- Alexus got a 57
- Koma got a 98

Store these values in the appropriate spots in the `mathScores` array.

4. Print out the math scores of each student. Print each score on a new line and use the following format:

Scott: 88

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] students = {"Sade", "Alexus", "Sam", "Koma"};
        int[] mathScores = new int[4];
        mathScores[0] = 64;
        mathScores[2] = 76;
        mathScores[1] = 57;
        mathScores[3] = 98;

        for(int i = 0; i < students.length; i++){
            System.out.println(students[i] + ": " + mathScores[i]);
        }
    }
}
```

Sade: 64

Alexus: 57

Sam: 76

Koma: 98

# ARRAYLISTS

## Introduction

When we work with [arrays](#) in Java, we've been limited by the fact that once an array is created, it has a fixed size. We can't add or remove elements.

But what if we needed to add to the book lists, newsfeeds, and other structures we were using arrays to represent?

To create mutable and dynamic lists, we can use Java's

## [ArrayList](#)

class. `ArrayList` allows us to:

- Store object references as elements
- Store elements of the same type (just like arrays)
- Access elements by index (just like arrays)
- Add elements
- Remove elements

Remember how we had to import `java.util.Arrays` in order to use additional array methods? To use an `ArrayList` at all, we need to import them from Java's `util` package as well:

```
import java.util.ArrayList;
```

Let's learn how to make use of this powerful object...

## Instructions

1. In `Shopping.java` we've defined two arrays:

- `groceryItems`, a `String` array
- `prices`, a `double` array

We've tried to add a new item to the end of each. Run the code — does it work?



```
import java.util.Arrays;

class Shopping {

    public static void main(String[] args) {

        String[] groceryItems = {"steak", "milk", "jelly beans"};
        double[] prices = {25.00, 2.95, 2.50};

        // Adding ham to the groceries
        groceryItems[3] = "ham";
        prices[3] = 4.99;

    }

}
```

Exception in thread "main"

java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds  
for length 3

at Shopping.main(Shopping.java:11)

## Creating ArrayLists

To create an `ArrayList`, we need to declare the type of objects it will hold, just as we do with arrays:

```
ArrayList<String> babyNames;
```

We use angle brackets `<` and `>` to declare the type of the `ArrayList`. These symbols are used for *generics*. Generics are a Java construct that allows us to define classes and objects as parameters of an `ArrayList`. For this reason, we can't use primitive types in an `ArrayList`:

```
// This code won't compile:
```

```
ArrayList<int> ages;
```

```
// This code will compile:
```

```
ArrayList<Integer> ages;
```

The `<Integer>` generic has to be used in an `ArrayList` instead. You can also use `<Double>` and `<Character>` for types you would normally declare as `doubles` or `chars`.

We can initialize to an empty `ArrayList` using the `new` keyword:

```
// Declaring:
```

```
ArrayList<Integer> ages;
```

```
// Initializing:
```

```
ages = new ArrayList<Integer>();
```

```
// Declaring and initializing in one line:
```

```
ArrayList<String> babyNames = new ArrayList<String>();
```

## Instructions

1. Import the `ArrayList` package from `java.util`.

2. Create a new `ArrayList` that will contain `String` elements and call it `toDoList`.

```
// import the ArrayList package here:
```

```
import java.util.ArrayList;
```

```
class ToDos {
```

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

```
        // Create toDoList below:
```

```
        ArrayList<String> toDoList = new ArrayList<String>();
```

```
    }}
```

## Adding an Item

Now we have an empty `ArrayList`, but how do we get it to store values?

`ArrayList` comes with an `add()` method which inserts an element into the structure. There are two ways we can use `add()`.

If we want to add an element to the end of the `ArrayList`, we'll call `add()` using only one argument that represents the value we are inserting. In this example, we'll add objects from the `Car` class to an `ArrayList` called `carShow`:

```
ArrayList<Car> carShow = new ArrayList<Car>();  
  
carShow.add(ferrari);  
// carShow now holds [ferrari]  
carShow.add(thunderbird);  
// carShow now holds [ferrari, thunderbird]  
carShow.add(volkswagen);  
// carShow now holds [ferrari, thunderbird, volkswagen]
```

If we want to add an element at a specific index of our `ArrayList`, we'll need two arguments in our method call: the first argument will define the index of the new element while the second argument defines the value of the new element:

```
// Insert object corvette at index 1  
carShow.add(1, corvette);  
// carShow now holds [ferrari, corvette, thunderbird, volkswagen]  
  
// Insert object porsche at index 2  
carShow.add(2, porsche);  
// carShow now holds [ferrari, corvette, porsche, thunderbird, volkswagen]
```

By inserting a value at a specified index, any elements that appear after this new element will have their index value shift over by 1.

Also, note that an error will occur if we try to insert a value at an index that does not exist.

## Keep Reading: AP Computer Science A Students

When using `ArrayList` [methods](#) (like `add()`), the reference parameters and return type of a method must match the declared element type of the `ArrayList`. For example, we cannot add an `Integer` type value to an `ArrayList` of `String` elements.

We've discussed how to specify the element type of an `ArrayList`; however, it is possible to create an `ArrayList` that holds values of different types.

In the following snippet, `assortment` is an `ArrayList` that can store different values because we do not specify its type during initialization.

```
ArrayList assortment = new ArrayList<>();
assortment.add("Hello"); // String
assortment.add(12); // Integer
assortment.add(ferrari); // reference to Car
// assortment holds ["Hello", 12, ferrari]
```

In this case, the items stored in this `ArrayList` will be considered `Objects`. As a result, they won't have access to some of their methods without doing some fancy casting. Although this type of `ArrayList` is allowed, using an `ArrayList` that specifies its type is preferred.

## Instructions

1. We've created an empty `ArrayList` called `todoList`. Time to add some to-dos!

Below where we've initialized `todo1`, initialize two new `String` variables: `todo2` and `todo3`.

Set their values to any tasks you like.

2. Use `.add()` to add `todo1`, `todo2`, and `todo3` to `todoList`.

```
import java.util.ArrayList;

class Todos {

    public static void main(String[] args) {

        ArrayList<String> todoList = new ArrayList<String>();
        String todo1 = "Water plants";
        // Add more to-dos here:
        String todo2 = "fuck like horse";
        String todo3 = "my bottle is red";

        // Add to-dos to todoList
        todoList.add(todo1);
        todoList.add(todo2);
        todoList.add(todo3);
        System.out.println(todoList);
    }
}
```

```
[Water plants, fuck like horse, my bottle is red]
```

## ArrayList Size

Let's say we have an `ArrayList` that stores items in a user's online shopping cart. As the user navigates through the site and adds items, their cart grows bigger and bigger.

If we wanted to display the number of items in the cart, we could find the size of it using the `size()` method:

```
ArrayList<String> shoppingCart = new ArrayList<String>();

shoppingCart.add("Trench Coat");
System.out.println(shoppingCart.size());
// 1 is printed

shoppingCart.add("Tweed Houndstooth Hat");
System.out.println(shoppingCart.size());
// 2 is printed

shoppingCart.add("Magnifying Glass");
System.out.println(shoppingCart.size());
// 3 is printed
```

In dynamic objects like `ArrayLists`, it's important to know how to access the amount of objects we have stored.

### Instructions

1. Detectives do a lot to solve a case. But who has more to do?

Print out the size of each detective's to-do `ArrayList`:

- `sherlocksToDos` for Sherlock Holmes
- `poirotsToDos` for Hercule Poirot

2. So who has more to do? Print the name of the detective whose to-do list is longer. Was it Sherlock or Poirot?

```
import java.util.ArrayList;

class ToDos {
```

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

    // Sherlock
    ArrayList<String> sherlocksToDos = new ArrayList<String>();

    sherlocksToDos.add("visit the crime scene");
    sherlocksToDos.add("play violin");
    sherlocksToDos.add("interview suspects");
    sherlocksToDos.add("solve the case");
    sherlocksToDos.add("apprehend the criminal");

    System.out.println(sherlocksToDos.size());

    // Poirot
    ArrayList<String> poirotsToDos = new ArrayList<String>();

    poirotsToDos.add("visit the crime scene");
    poirotsToDos.add("interview suspects");
    poirotsToDos.add("let the little grey cells do their work");
    poirotsToDos.add("trim mustache");
    poirotsToDos.add("call all suspects together");
    poirotsToDos.add("reveal the truth of the crime");

    // Print the size of each ArrayList below:
    System.out.println(poirotsToDos.size());

    // Print the name of the detective with the larger to-do list:
    if(sherlocksToDos.size() > poirotsToDos.size())
    {
        System.out.println("sherlock");
    } else {
        System.out.println("poirot");
    }
}
```

5

6

poirot

Accessing an Index With [arrays](#), we can use bracket notation to access a value at a particular index:

```
double[] ratings = {3.2, 2.5, 1.7};
```

```
System.out.println(ratings[1]);
```

This code prints `2.5`, the value at index `1` of the array.

For `ArrayLists`, bracket notation won't work. Instead, we use the method `get()` to access an index:

```
ArrayList<String> shoppingCart = new ArrayList<String>();
```

```
shoppingCart.add("Trench Coat");
```

```
shoppingCart.add("Tweed Houndstooth Hat");
```

```
shoppingCart.add("Magnifying Glass");
```

```
System.out.println(shoppingCart.get(2));
```

This code prints `"Magnifying Glass"`, which is the value at index 2 of the `ArrayList`.

## Instructions

1. Use `get()` to access the third to-do element of `sherlocksToDos` and print the result.

2. Use `get()` to access the second to-do element of `poirotsToDos` and print the result.

```
import java.util.ArrayList;
```



```
class ToDos {

    public static void main(String[] args) {

        // Sherlock
        ArrayList<String> sherlocksToDos = new ArrayList<String>();

        sherlocksToDos.add("visit the crime scene");
        sherlocksToDos.add("play violin");
        sherlocksToDos.add("interview suspects");
        sherlocksToDos.add("solve the case");
        sherlocksToDos.add("apprehend the criminal");

        // Poirot
        ArrayList<String> poirotsToDos = new ArrayList<String>();

        poirotsToDos.add("visit the crime scene");
        poirotsToDos.add("interview suspects");
        poirotsToDos.add("let the little grey cells do their work");
        poirotsToDos.add("trim mustache");
        poirotsToDos.add("call all suspects together");
        poirotsToDos.add("reveal the truth of the crime");

        System.out.println("Sherlock's third to-do:");
        // Print Sherlock's third to-do:
        System.out.println( sherlocksToDos.get(2));

        System.out.println("\nPoirot's second to-do:");
        // Print Poirot's second to-do:
        System.out.println(poirotsToDos.get(1));
    }
}
```

Sherlock's third to-do:  
interview suspects

Poirot's second to-do:  
interview suspects

## Changing a Value

When we were using [arrays](#), we could rewrite entries by using bracket notation to reassign values:

```
String[] shoppingCart = {"Trench Coat", "Tweed Houndstooth Hat",  
"Magnifying Glass"};
```

```
shoppingCart[0] = "Tweed Cape";
```

```
// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth  
Hat", "Magnifying Glass"]
```

ArrayList has a slightly different way of doing this, using the `set()` method:

```
ArrayList<String> shoppingCart = new ArrayList<String>();
```

```
shoppingCart.add("Trench Coat");
```

```
shoppingCart.add("Tweed Houndstooth Hat");
```

```
shoppingCart.add("Magnifying Glass");
```

```
shoppingCart.set(0, "Tweed Cape");
```

```
// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth  
Hat", "Magnifying Glass"]
```

## Instructions

1. Modify `sherlocksToDos` so that the value at `"play violin"` becomes `"listen to Dr. Watson for amusement"`.
2. Modify `poirotsToDos` so that the value at `"trim mustache"` becomes `"listen to Captain Hastings for amusement"`.

```
import java.util.ArrayList;

class ToDos {

    public static void main(String[] args) {

        // Sherlock
        ArrayList<String> sherlocksToDos = new ArrayList<String>();

        sherlocksToDos.add("visit the crime scene");
        sherlocksToDos.add("play violin");
        sherlocksToDos.add("interview suspects");
        sherlocksToDos.add("solve the case");
        sherlocksToDos.add("apprehend the criminal");

        // Poirot
        ArrayList<String> poirotsToDos = new ArrayList<String>();

        poirotsToDos.add("visit the crime scene");
        poirotsToDos.add("interview suspects");
        poirotsToDos.add("let the little grey cells do their work");
        poirotsToDos.add("trim mustache");
        poirotsToDos.add("call all suspects together");
        poirotsToDos.add("reveal the truth of the crime");

        // Set each to-do below:
        sherlocksToDos.set(1,"listen to Dr. Watson for amusement");
```

```
poirotsToDos.set(3,"listen to Captain Hastings for amusement");

System.out.println("Sherlock's to-do list:");
System.out.println(sherlocksToDos.toString() + "\n");
System.out.println("Poirot's to-do list:");
System.out.println(poirotsToDos.toString());
}
}
```

Sherlock's to-do list:

[visit the crime scene, listen to Dr. Watson for amusement,  
interview suspects, solve the case, apprehend the criminal]

Poirot's to-do list:

[visit the crime scene, interview suspects, let the little grey cells  
do their work, listen to Captain Hastings for amusement, call all  
suspects together, reveal the truth of the crime]

## Removing an Item

What if we wanted to get rid of an entry altogether? For [arrays](#), we would have to make a completely new array without the value.

Luckily, `ArrayLists` allow us to remove an item by specifying the index to [remove](#):

```
ArrayList<String> shoppingCart = new ArrayList<String>();

shoppingCart.add("Trench Coat");
shoppingCart.add("Tweed Houndstooth Hat");
shoppingCart.add("Magnifying Glass");
```

```
shoppingCart.remove(1);  
// shoppingCart now holds ["Trench Coat", "Magnifying Glass"]
```

We can also remove an item by specifying the value to remove:

```
ArrayList<String> shoppingCart = new ArrayList<String>();  
  
shoppingCart.add("Trench Coat");  
shoppingCart.add("Tweed Houndstooth Hat");  
shoppingCart.add("Magnifying Glass");  
  
shoppingCart.remove("Trench Coat");  
// shoppingCart now holds ["Tweed Houndstooth Hat", "Magnifying Glass"]
```

**Note:** This command removes the FIRST instance of the value "Trench Coat".

## Instructions

1. Sherlock Holmes and Hercule Poirot have each already visited their respective crime scenes.

Remove "visit the crime scene" from `sherlocksToDos` and `poirotsToDos` using `remove()`.

Moreover, Sherlock Holmes has also gotten some violin playing done.

So you can remove "play violin" from `sherlocksToDos` as well.

```
import java.util.ArrayList;  
  
class ToDos {  
  
    public static void main(String[] args) {  
  
        // Sherlock  
        ArrayList<String> sherlocksToDos = new ArrayList<String>();
```

```
sherlocksToDos.add("visit the crime scene");
sherlocksToDos.add("play violin");
sherlocksToDos.add("interview suspects");
sherlocksToDos.add("solve the case");
sherlocksToDos.add("apprehend the criminal");
```

```
// Poirot
```

```
ArrayList<String> poirotsToDos = new ArrayList<String>();
```

```
poirotsToDos.add("visit the crime scene");
poirotsToDos.add("interview suspects");
poirotsToDos.add("let the little grey cells do their work");
poirotsToDos.add("trim mustache");
poirotsToDos.add("call all suspects together");
poirotsToDos.add("reveal the truth of the crime");
```

```
// Remove each to-do below:
```

```
poirotsToDos.remove(0);
sherlocksToDos.remove(0);
sherlocksToDos.remove("play violin");
```

```
System.out.println(sherlocksToDos.toString() + "\n");
System.out.println(poirotsToDos.toString());
```

```
}
```

```
}
```

[interview suspects, solve the case, apprehend the criminal]

[interview suspects, let the little grey cells do their work, trim mustache, call all suspects together, reveal the truth of the crime]

## Getting an Item's Index

What if we had a really large list and wanted to know the position of a certain element in it? For instance, what if we had an `ArrayList` `detectives` with the names of fictional detectives in chronological order, and we wanted to know what position `"Fletcher"` was.

```
// detectives holds ["Holmes", "Poirot", "Marple", "Spade",  
"Fletcher", "Conan", "Ramotswe"];  
System.out.println(detectives.indexOf("Fletcher"));
```

This code would print `4`, since `"Fletcher"` is at index `4` of the `detectives` `ArrayList`.

### Instructions

1. After visiting the crime scene, the ever-impatient Dr. Watson wants to know how many to-dos are left until Sherlock solves the case.

To help Dr. Watson figure this out, use `indexOf()` to determine where in the to-do list `"solve the case"` is.

Print this information out. That's the number of to-dos remaining before Sherlock reaches `"solve the case"`.

```
import java.util.ArrayList;  
  
class ToDos {  
  
    public static void main(String[] args) {  
  
        // Sherlock  
        ArrayList<String> sherlocksToDos = new ArrayList<String>();  
  
        sherlocksToDos.add("visit the crime scene");  
        sherlocksToDos.add("play violin");  
    }  
}
```

```
sherlocksToDos.add("interview suspects");
sherlocksToDos.add("listen to Dr. Watson for amusement");
sherlocksToDos.add("solve the case");
sherlocksToDos.add("apprehend the criminal");

sherlocksToDos.remove("visit the crime scene");

// Calculate to-dos until case is solved:
int solved = sherlocksToDos.indexOf("solve the case");

// Change the value printed:
System.out.println("solved");

}
}
solved
```

## Review

Nice work! You now know the basics of

ArrayLists including:

- Creating an `ArrayList`.
- Adding a new `ArrayList` item using `add()`.
- Accessing the size of an `ArrayList` using `size()`.
- Finding an item by index using `get()`.
- Changing the value of an `ArrayList` item using `set()`.
- Removing an item with a specific value using `remove()`.
- Retrieving the index of an item with a specific value using `indexOf()`.

Now if only there were some way to move through an array or `ArrayList`, item by item...



## Instructions

We've included a workspace for you to test out your newfound knowledge of arrays and `ArrayLists`.

Remember: To run your code, enter the following commands in the terminal:

```
javac List.java
```

and then:

```
java List
```

```
import java.util.ArrayList;

class List {

    public static void main(String[] args) {

    }

}
```

# LOOPS

## Introduction to Loops

In the programming world, we hate repeating ourselves. There are two reasons for this:

- Writing the same code over and over is time-consuming.
- Having less code means having less to debug.

But we often need to do the same task more than once. Fortunately, computers are really good (and fast) at doing repetitive tasks. And in Java, we can use loops.

A [loop](#) is a programming tool that allows developers to repeat the same block of code until some condition is met.

First, a starting condition is evaluated. If the starting condition is **true**, then the loop body is executed. When the last line of the loop body is executed, the condition is *re-evaluated*. This process continues until the condition is **false** (if the condition never becomes **false**, we can actually end up with an **infinite loop**!). If the starting condition is **false**, the loop never gets executed.

We employ loops to easily scale programs - saving time and minimizing mistakes.

We'll go over three types of loops that we'll see everywhere:

- **while** loops
- **for** loops
- **for-each** loops

## While We're Here

A **while** loop looks a bit like an **if** statement:

```
while (silliness > 10) {
```

```
// code to run  
}
```

Like an **if** statement, the code inside a **while** loop will only run if the condition is **true**. However, a **while** loop will continue running the code over and over until the condition evaluates to **false**. So the code block will repeat until **silliness** is less than or equal to 10.

```
// set attempts to 0  
int attempts = 0;  
  
// enter loop if condition is true  
while (passcode != 0524 && attempts < 4) {  
  
    System.out.println("Try again.");  
    passcode = getNewPasscode();  
    attempts += 1;  
  
    // is condition still true?  
    // if so, repeat code block  
}  
// exit when condition is not true
```

**while loops** are extremely useful when you want to run some code until a specific change happens. However, if you aren't certain that change will occur, beware the infinite loop!

*Infinite loops* occur when the condition will never evaluate to **false**. This can cause your entire program to crash.

```
int hedgehogs = 5;  
  
// This will cause an infinite loop:  
while (hedgehogs < 6) {
```

```
System.out.println("Not enough hedgehogs!");
```

```
}
```

In the example above, `hedgehogs` remains equal to `5`, which is less than `6`. So we would get an infinite loop.

## Instructions

1. Take a look at `LuckyFive.java`. We've set up a random number generator that allows you to simulate the roll of a die.

Create a `while` loop that will continue to loop while `dieRoll` is NOT `5`.

**Do NOT run your code yet** — you will get an infinite loop here because the value of `dieRoll` is never changed.

Inside the loop, reset `dieRoll` with a new random value between `1` and `6`.

Now you can run the code.

2. Inside the `while` loop, above the line where you reset `dieRoll`, print out `dieRoll` to the terminal.

```
// Importing the Random library
import java.util.Random;

class LuckyFive {

    public static void main(String[] args) {

        // Creating a random number generator
        Random randomGenerator = new Random();

        // Generate a number between 1 and 6
        int dieRoll = randomGenerator.nextInt(6) + 1;

        // Repeat while roll isn't 5
```

```
while(dieRoll != 5){  
  
    System.out.println(dieRoll);  
    dieRoll = randomGenerator.nextInt(6) + 1;  
}  
}}
```

```
2  
4  
2  
3
```

## Incrementing While Loops

When looping through code, it's common to use a counter variable. A *counter* (also known as an *iterator*) is a variable used in the conditional logic of the loop and (usually) incremented in value during each iteration through the code. For example:

```
// counter is initialized  
int wishes = 0;  
  
// conditional logic uses counter  
while (wishes < 3) {  
  
    System.out.println("Wish granted.");  
    // counter is incremented  
    wishes++;  
}
```

In the above example, the counter `wishes` gets initialized before the loop with a value of `0`, then the program will keep printing "`Wish granted.`" and adding `1` to `wishes` as long as `wishes` has a value of less than `3`. Once `wishes` reaches a value of `3` or more, the program will exit the loop.

So the output would look like:

```
Wish granted.  
Wish granted.  
Wish granted.
```

We can also decrement counters like this:

```
int pushupsToDo = 10;  
  
while (pushupsToDo > 0) {  
  
    doPushup();  
    pushupsToDo--;  
  
}
```

In the code above, the counter, `pushupsToDo`, starts at 10, and increments down one at a time. When it hits 0, the loop exits.

### Instructions

1. In `Coffee.java`, initialize an `int` variable called `cupsOfCoffee` with a value of `1`.
2. Create a `while` loop that runs as long as `cupsOfCoffee` is less than or equal to `100`.

**Important:** Inside the `while` loop, increment `cupsOfCoffee` by `1` to prevent an infinite loop.

3. Inside the `while` loop above where you incremented `cupsOfCoffee` print the following:

```
Fry drinks cup of coffee #1
```

The `1` in this statement should correspond with the current value of `cupsOfCoffee`. When `cupsOfCoffee` is `100`, this should be printed:

```
Fry drinks cup of coffee #100
class Coffee {

    public static void main(String[] args) {

        // initialize cupsOfCoffee
        int cupsOfCoffee = 1;
        // add while loop with counter
        while (cupsOfCoffee <= 100){
            System.out.println("Fry drinks cup of coffee
#" + cupsOfCoffee);
            cupsOfCoffee++;
        }
    }
}
```

```
Fry drinks cup of coffee #1
Fry drinks cup of coffee #2
Fry drinks cup of coffee #3
Fry drinks cup of coffee #4
.
.
.
Fry drinks cup of coffee #99
Fry drinks cup of coffee #100
```

## For Loops

Incrementing with loops is actually so common in programming that Java (like many other programming languages) includes syntax specifically to address this pattern: **for** loops.

A **for** loop header is made up of the following three parts, each separated by a semicolon:

1. The initialization of the loop control variable.
2. A boolean expression.
3. An increment or decrement statement.

The opening line might look like this:

```
for (int i = 0; i < 5; i++) {  
  
    // code that will run  
  
}
```

In a **for** loop, an initialization statement is run once in order to initialize the loop control variable. This variable is modified in every iteration, can be referenced in the loop body, and used to test the boolean condition. In the example above, **i** is the loop control variable.

Let's breakdown the above example:

1. **i = 0**: **i** is initialized to **0**
2. **i < 5**: the loop is given a boolean condition that relies on the value of **i**. The loop will continue to execute until **i < 5** is **false**.
3. **i++**: **i** will increment at the end of each loop and before the condition is re-evaluated.

So the code will run through the loop a total of five times.



We'll also hear the term "iteration" in reference to loops. When we *iterate*, it just means that we are repeating the same block of code.

## Instructions

Review the syntax of `for` loops and click Next when you're ready to build some yourself!

### Using For Loops

Even though we can write `while` [loops](#) that accomplish the same task, `for` loops are useful because they help us remember to increment our counter — something that is easy to forget when we increment with a `while` loop.

Leaving out that line of code would cause an infinite loop — yikes!

Fortunately, equipped with our new understanding of `for` loops, we can help prevent infinite loops in our own code.

It's important to be aware that, if we don't create the correct `for` loop header, we can cause the iteration to loop one too many or one too few times; this occurrence is known as an "off by one" error.

For example, imagine we wanted to find the sum of the first ten numbers and wrote the following code:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i
}
```

This code would produce an incorrect value of `45`. We skipped adding `10` to `sum` because our loop control variable started with a value of `0` and stopped the iteration after it had a value of `9`. We were off by one! We could fix this by changing the condition of our loop to be `i <= 10`; or `i < 11`;

These [errors](#) can be tricky because, while they do not always produce an error in the terminal, they can cause some miscalculations in our code. These are called logical errors — the code runs fine, but it didn't do what you expected it to do.

## Instructions

1. We've provided a `while` loop that loops from `1` to `100` outputting a string on each iteration. Refactor (rewrite) this code as a `for` loop. Remember, the syntax of a `for` loop looks like:

```
for (int i = 0; i < 5; i++) {  
  
    // code that will run  
  
}  
class Coffee {  
  
    public static void main(String[] args) {  
  
        int cupsOfCoffee = 1;  
  
        for(int i = 1; i <=100; i++){  
cupsOfCoffee = i;  
System.out.println("Fry drinks cup of coffee #" +cupsOfCoffee );  
        }  
    }  
}
```

```
Fry drinks cup of coffee #1  
Fry drinks cup of coffee #2  
Fry drinks cup of coffee #3  
Fry drinks cup of coffee #4  
.  
.  
.  
Fry drinks cup of coffee #99  
Fry drinks cup of coffee #100
```

## Iterating Over Arrays and ArrayLists

One common pattern we'll encounter as a programmer is *traversing*, or looping, through a list of data and doing something with each item. In Java, that list would be an array or ArrayList and the loop could be a for loop. But wait, how does this work?

In order to traverse an array or ArrayList using a loop, we must find a way to access each element via its index. We may recall that for loops are created with a counter variable. We can use that counter to track the index of the current element as we iterate over the list of data.

Because the first index in an array or ArrayList is 0, the counter would begin with a value of 0 and increment until the end of the list. So we can increment through the array or ArrayList using its indices.

For example, if we wanted to add 1 to every int item in an array secretCode, we could do this:

```
for (int i = 0; i < secretCode.length; i++) {  
    // Increase value of element value by 1  
    secretCode[i] += 1;  
}
```

Notice that our condition in this example is i < secretCode.length. Because array indices start at 0, the length of secretCode is 1 larger than its final index. A loop should stop its traversal before its counter variable is equal to the length of the list.

To give a concrete example, if the length of an array is 5, the last index we want to access is 4. If we were to try to access index 5, we would get an ArrayIndexOutOfBoundsException error! This is a very common mistake when first starting to traverse arrays.

Traversing an `ArrayList` looks very similar:

```
for (int i = 0; i < secretCode.size(); i++) {  
    // Increase value of element value by 1  
    int num = secretCode.get(i);  
    secretCode.set(i, num + 1);  
}
```

We can also use `while` loops to traverse through arrays and `ArrayLists`. If we use a `while` loop, we need to create our own counter variable to access individual elements. We'll also set our condition to continue looping until our counter variable equals the list length.

For example, let's use a `while` loop to traverse through an array:

```
int i = 0; // initialize counter  
  
while (i < secretCode.length) {  
    secretCode[i] += 1;  
    i++; // increment the while loop  
}
```

Traversing through an `ArrayList` with a `while` loop would look like this:

```
int i = 0; // initialize counter  
  
while (i < secretCode.size()) {  
    int num = secretCode.get(i);  
    secretCode.set(i, num + 1);  
    i++; // increment the while loop  
}
```

## Instructions

1. Let's use a **for** loop to iterate over **expenses** and sum up the **total** of all items.

Start with the skeleton of a **for** loop:

- Initialize a counter **i** with a value of **0**.
- The loop should run while **i** is less than the **size()** of **expenses**.
- Increment **i**.

You can leave the body empty for now.

2. Inside the **for** loop, add the item's value to **total**.

```
import java.util.ArrayList;

class CalculateTotal {

    public static void main(String[] args) {

        ArrayList<Double> expenses = new ArrayList<Double>();
        expenses.add(74.46);
        expenses.add(63.99);
        expenses.add(10.57);
        expenses.add(81.37);

        double total = 0;

        // Iterate over expenses
        for(int i = 0; i < expenses.size(); i++){
            total += expenses.get(i);
        }

        System.out.println(total);

    }
}
```

230.39

## break and continue

If we ever want to exit a loop before it finishes all its iterations or want to skip one of the iterations, we can use the `break` and `continue` keywords.

The `break` keyword is used to exit, or break, a loop. Once `break` is executed, the loop will stop iterating. For example:

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
    if (i == 4) {  
        break;  
    }  
}
```

Even though the loop was set to iterate until the condition `i < 10` is `false`, the above code will output the following because we used `break`:

```
0  
1  
2  
3  
4
```

The `continue` keyword can be placed inside of a loop if we want to skip an iteration. If `continue` is executed, the current loop iteration will immediately end, and the next iteration will begin. We can use the `continue` keyword to skip any even valued iteration:

```
int[] numbers = {1, 2, 3, 4, 5};  
  
for (int i = 0; i < numbers.length; i++) {  
    if (numbers[i] % 2 == 0) {  
        continue;  
    }  
}
```

```
}  
    System.out.println(numbers[i]);  
}
```

This program would output the following:

```
1  
3  
5
```

In this case, if a number is even, we hit a `continue` statement, which skips the rest of that iteration, so the print statement is skipped. As a result, we only see odd numbers print.

### Keep Reading: AP Computer Science A Students

Loops can exist all throughout our code - including inside a method. If the `return` keyword was executed inside a loop contained in a method, then the loop iteration would be stopped and the method/constructor would be exited.

For example, we have a method called `checkForJacket()` that takes in an array of `Strings`. If any of the elements are equivalent to the `String` value `"jacket"`, the method will return `true`:

```
public static boolean checkForJacket(String[] lst) {  
    for (int i = 0; i < lst.length; i++) {  
        System.out.println(lst[i]);  
        if (lst[i] == "jacket") {  
            return true;  
        }  
    }  
    return false;  
}  
  
public static void main(String[] args) {
```

```
String[] suitcase = {"shirt", "jacket", "pants", "socks"};
System.out.println(checkForJacket(suitcase));
}
```

As soon as an element equals "jacket", `return true;` is executed. This causes the loop to stop and the [compiler](#) to exit `checkForJacket()`. Running this code would output the following:

```
shirt
jacket
true
```

## Instructions

1. Take a look at the `for` loop in the code editor. It starts its iteration at `0` and continues to iterate until `i < 100` is `false`.

Inside the loop, create a condition that checks if `i` is **not** divisible by `5`. If the condition is `true`, skip the iteration. Outside the condition statement, print `i`. The final solution **should not** contain an `else` statement.

The only numbers that should be printed are those that are divisible by 5!

```
class Numbers {
    public static void main(String[] args) {
        for (int i = 0; i <= 100; i++) {
            // Add your code below
            if(i % 5 != 0){

                continue;
            }
            System.out.println(i);

        }
    }
}
```



0  
5  
10  
15  
20  
25  
30  
35  
40  
45  
50  
55  
60  
65  
70  
75  
80  
85  
90  
95  
100

## For-Each Loops

Sometimes we need access to the elements' indices or we only want to iterate through a portion of a list. If that's the case, a regular `for` loop or `while` loop is a great choice.

For example, we can use a `for` loop to print out each element in an array called `inventoryItems`:

```
for (int inventoryItem = 0; inventoryItem < inventoryItems.length;
inventoryItem++) {
    // Print element at current index
    System.out.println(inventoryItems[inventoryItem]);
}
```

But sometimes we couldn't care less about the indices; we only care about the element itself.

At times like this, for-each [loops](#) come in handy.

*For-each loops*, which are also referred to as *enhanced loops*, allow us to directly loop through each item in a list of items (like an array or `ArrayList`) and perform some action with each item.

If we want to use a for-each loop to rewrite our program above, the syntax looks like this:

```
for (String inventoryItem : inventoryItems) {
    // Print element value
    System.out.println(inventoryItem);
}
```

Our enhanced loop contains two items: an enhanced `for` loop variable (`inventoryItem`) and a list to traverse through (`inventoryItems`).

We can read the `:` as "in" like this: for each `inventoryItem` (which should be a `String`) in `inventoryItems`, print `inventoryItem`.

If we try to assign a new value to the enhanced `for` loop variable, the value stored in the array or `ArrayList` will not change. This is because, for every iteration in the enhanced loop, the loop variable is assigned a copy of the list element.

**Note:** We can name the enhanced `for` loop variable whatever we want; using the singular of a plural is just a convention. We may also encounter conventions like `String word : sentence`.

## Instructions

1. Let's use a for-each loop to find the priciest item in `expenses`.

Build a for-each loop that iterates through each `expense` in `expenses`. For now, leave the body of the loop empty.

2. Inside the for-each loop, check if `expense` is greater than `mostExpensive`.

If it is, set `mostExpensive` equal to `expense`.

```
import java.util.ArrayList;

class MostExpensive {

    public static void main(String[] args) {

        ArrayList<Double> expenses = new ArrayList<Double>();
        expenses.add(74.46);
        expenses.add(63.99);
        expenses.add(10.57);
        expenses.add(81.37);

        double mostExpensive = 0;

        // Iterate over expenses
        for(double expense : expenses){
            if (expense > mostExpensive){
```

```
        mostExpensive = expense;
    }
}
System.out.println(mostExpensive);
}
}
```

81.37

## Removing Elements During Traversal

If we want to remove elements from an `ArrayList` while traversing through one, we can easily run into an error if we aren't careful. When an element is removed from an `ArrayList`, all the items that appear after the removed element will have their index value shift by negative one — it's like all elements shifted to the left! We'll have to be very careful with how we use our counter variable to avoid skipping elements.

### Removing An Element Using `while`

When using a `while` loop and removing elements from an `ArrayList`, we should **not** increment the `while` loop's counter whenever we remove an element. We don't need to increase the counter because all of the other elements have now shifted to the left.

For example, if we removed the element at index `3`, then the element that was at index `4` will be moved to index `3`. If we increase our counter to `4`, we'll skip that element!

Take a look at this block of code that will remove all odd numbers from an `ArrayList`. Think about what the value of `i` is, when we're increasing the value of `i`, and when `i < lst.size()` becomes `False`.

```
int i = 0; // initialize counter

while (i < lst.size()) {
    // if value is odd, remove value
```

```
if (lst.get(i) % 2 != 0){  
    lst.remove(i);  
} else {  
    // if value is even, increment counter  
    i++;  
}  
}
```

### Removing An Element Using for

We can use a similar strategy when removing elements using a **for** loop. When using a **while** loop, we decided to not increase our loop control variable whenever we removed an element. This ensured that we would not skip an element when all of the other elements shifted to the left.

When using a **for** loop, we, unfortunately, *must* increase our loop control variable — the loop control variable will always change when we reach the end of the loop (and it will usually change by **1** because we often use something like **i++**.) Since we can't avoid increasing our loop control variable, we can take matters into our own hands and decrease the loop control variable whenever we remove an item.

For example:

```
for (int i = 0; i < lst.size(); i++) {  
    if (lst.get(i) == "value to remove"){  
        // remove value from ArrayList  
        lst.remove(lst.get(i));  
        // Decrease loop control variable by 1  
        i--;  
    }  
}
```

Now whenever we remove an item, we'll decrease `i` by `1`. Then when we reach the end of the loop, `i` will increase by `1`. It will be like `i` never changed!

**Note:** Avoid manipulating the size of an `ArrayList` when using an enhanced `for` loop. Actions like adding or removing elements from an `ArrayList` when using a `for each` loop can cause a `ConcurrentModificationException` error.

**Instructions :**

1. Take a look at the code placed in the `main()` method of the `Lunch` class.

Inside the method `removeAnts()`, use a `for` loop or a `while` loop to iterate through `lunchBox` and remove any element that has the value `"ant"`.

Outside the loop, return the value of `lunchBox`.

```
import java.util.ArrayList;

class Lunch {

    public static ArrayList<String> removeAnts(ArrayList<String>
lunchBox) {
    // Add your code below
    for(int i=0;i<lunchBox.size();i++){
    if(lunchBox.get(i) == "ant"){

lunchBox.remove(lunchBox.get(i));
i--;
}
    }
    return lunchBox;
}

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

```

ArrayList<String> lunchContainer = new ArrayList<String>();
lunchContainer.add("apple");
lunchContainer.add("ant");
lunchContainer.add("ant");
lunchContainer.add("sandwich");
lunchContainer.add("ant");
lunchContainer = removeAnts(lunchContainer);
System.out.println(lunchContainer);
}
}
[apple, sandwich]

```

## Review

Nice work! Let's iterate over what you've just learned about loops:

- **while** loops: These are useful to repeat a code block an unknown number of times until some condition is met. For example:

```

int wishes = 0;

while (wishes < 3) {
    // code that will run
    wishes++;
}

```

- **for** loops: These are ideal for when you are incrementing or decrementing with a counter variable. For example:

```

for (int i = 0; i < 5; i++) {

    // code that will run
}

```

- For-each loops: These make it simple to do something with each item in a list. For example:

```
for (String inventoryItem : inventoryItems) {  
  
    // do something with each inventoryItem  
  
}  
import java.util.ArrayList;  
import java.util.Arrays;  
  
class Playground {  
  
    public static void main(String[] args) {  
  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Congrats on finishing Java loops!");  
        }  
    }  
}
```

```
Congrats on finishing Java loops!  
Congrats on finishing Java loops!  
Congrats on finishing Java loops!  
Congrats on finishing Java loops!
```