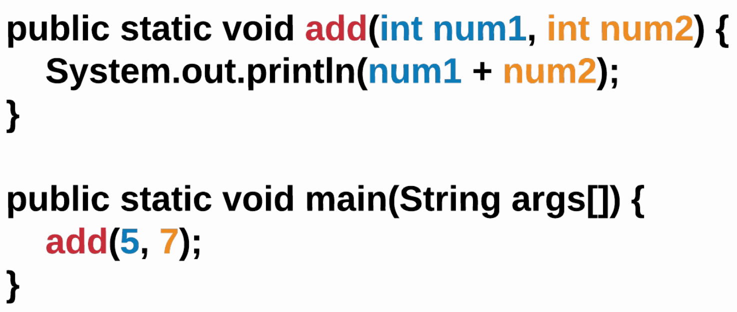
Learning Objectives: Parameters

- **Demonstrate how to define a method with parameters**
- **Identify which value is assigned to each parameter**
- **Catch exceptions with a try catch block**
- **Define methods with the same name but different parameters**
- **Demonstrate how to declare a method with an array and ArrayList as a parameter**

Passing Parameters

Parameters

If a method contains parameters within its definition, they are required to be present when the method is called. In the example below, the method, `add()`, adds two integer parameters together. Parameters are the types or values located in between the parentheses. Multiple parameters are separated by commas.



```
public static void add(int num1, int num2) {
   System.out.println(num1 + num2);
}

public static void main(String args[]) {
   add(5, 7);
}
```

.guides/img/Parameters1

Copy and paste the following method into the text editor to your left between the lines `//add method definitions below this line` and `//add method definitions above this line`. **DO NOT** modify `main()` yet, or the code will not print correctly!

```
/**
 * This method adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
public static void add(int num1, int num2) {
  System.out.println(num1 + num2);
}
```

challenge

What happens if you:

- Change the method call in `main()` to `add(5, "seven");`?
- Change the method call to `add();` without any parameters?
- Change the method call to `add(5, 10, 15);`?
- Change the entire method to...

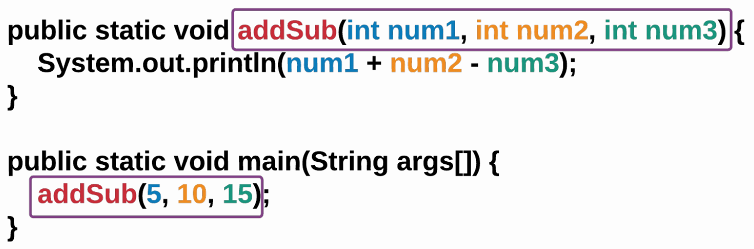
```java
public static void add(int num1, int num2, int num3) {
  System.out.println(num1 + num2 + num3);
}
```

important

IMPORTANT

- The **number** of arguments within `main()` should match the number of parameters specified in the method. If there are three parameters, then there should be three arguments as well.
- The argument **type** should also match the parameter type. If the method requires three integers, then the arguments should also consist of three integers. You cannot provide a string argument for an integer parameter, etc.

Order of Parameters



```java
public static void addSub(int num1, int num2, int num3) {
   System.out.println(num1 + num2 - num3);
}

public static void main(String args[]) {
   addSub(5, 10, 15);
}
```

.guides/img/Parameters2

Much like how Java programs run code from left to right and then top to bottom, parameters are also read the same way. Because of this, the order of parameters is important. The first argument in the method call will be matched with the first parameter in the method header, the second argument from the method call will be the second parameter in the method header, etc. Copy the entire code below into the text editor and then click `TRY IT`. What do you predict the output will be?

```
public class Parameters {

  /**
   * This method adds the first two integers together,
   * then subtracts the third integer
   *
   * @param num1 The first integer
   * @param num2 The second integer
   * @param num3 The third integer
   */
  public static void addSub(int num1, int num2, int num3) {
    System.out.println(num1 + num2 - num3);
  }

  public static void main(String args[]) {
    addSub(5, 10, 15);
  }

}
```

challenge

What happens if you:

- Change the method call in `main()` to `addSub(10, 15, 5);`?
- Change the method call in `main()` to `addSub(15, 5, 10);`?
- Change the method call in `main()` to `addSub(10 + 5, 20 / 4, 5 * 2);`?

Checking Parameters

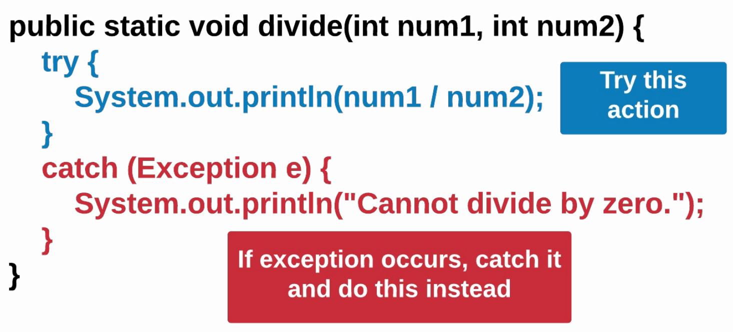
Checking Parameter Usage

Copy and paste the code below into the text editor and then `TRY IT`.

```
/**
 * This method divides one integer by the other
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
public static void divide(int num1, int num2) {
  System.out.println(num1 / num2);
}

public static void main(String args[]) {
  divide(5, 0);
}
```

You'll notice that the code produces an **exception**. An exception occurs when an operation cannot be successfully completed because a rule is broken. For example, dividing by `0` results in an *undefined* answer. Thus when you try to divide `5` by `0`, you get an exception as a response. Not all exception messages are created equal. Some are more clear than others. However, you may choose to clearly define an exception by using a `try catch` block.



```
public static void divide(int num1, int num2) {
   try {
      System.out.println(num1 / num2);
   }
   catch (Exception e) {
      System.out.println("Cannot divide by zero.");
   }
}
```

.guides/img/TryCatchException

```java
/**
 * This method divides one integer by the other
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
public static void divide(int num1, int num2) {
  try {
    System.out.println(num1 / num2);
  }
  catch (Exception e) {
    System.out.println("Cannot divide by zero.");
  }
}

public static void main(String args[]) {
  divide(5, 0);
}
```

challenge

What happens if you:

- Change the method call to `divide(5, 2)`?
- Change the method call to `divide(0, 2)`?
- Change the method call to `divide(14.5, 2)`?

important

IMPORTANT

Note that you can only catch Java **exceptions**, not compilation **errors**. If you attempt to provide an argument that is not the same type as a specified parameter (i.e. `14.5` is a double and not an int), the compiler will simply fail and the compilation error will be returned. `Exception` is the generic exception keyword that will catch any exception produced. `e` is the variable name for which you are calling the exception by. For now, we will only focus on `Exception`. For a list of other exceptions, visit: Java Exceptions.

Parameter Types

Method with Different Parameters

In Java, you are allowed to define methods with the *same* name as long as the parameters are *different* in quantity or type. Copy the code below and `TRY IT`.

```java
/**
 * This method adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
public static void add(int num1, int num2) {
  System.out.println(num1 + num2);
}

/**
 * This method adds three integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 * @param num3 The third integer
 */
public static void add(int num1, int num2, int num3) {
  System.out.println(num1 + num2 + num3);
}

public static void main(String args[]) {
  add(3, 14);
}
```

challenge

What happens if you:

- Change the method call to `add(3, 14, 9);`?
- Change the method call to `add(3, 14, 9, 2);`?

The two `add()` methods above differ in the number of parameters they have. Here is an example of two methods with the same name but different parameter types.

```
/**
 * This method adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
public static void add(int num1, int num2) {
  System.out.println(num1 + num2);
}

/**
 * This method adds an integer to a string
 *
 * @param num1 The integer
 * @param num2 The string
 */
public static void add(int num1, String num2) {
  System.out.println(num1 + num2);
}

public static void main(String args[]) {
  add(3, 14);
}
```

challenge

What happens if you:

- Change the method call to `add(3, "14");`?
- Change the method call to `add("14", 3);`?

Alternative Parameters

Alternative Parameter Types

Method parameters do not necessarily need to belong to one of the four commonly used data types (`int`, `String`, `double`, `boolean`). In fact, parameters can be arrays/ArrayLists and even objects. For now, we will not focus on objects, which will be covered in a future module.

```java
/**
 * This method prints all values of an array
 *
 * @param arr is an array of strings
 */
public static void printArray(String[] arr) {
  for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
  }
}

public static void main(String args[]) {
  String[] names = {"Alan", "Bob", "Carol"};
  printArray(names);
}
```

challenge

What happens if you:

- Change the method parameter from `String[] arr` to `String arr[]`?
- Change the variable `String[] names` to `String[] names = new String[3];`?

  ▼

  Explanation

  The variable `names` is an array of three strings. However, no values have been given for each of the elements in the array. Java uses `null` as a placeholder until a value is given. That is why `null` is printed three times.

- Add `names[0] = "Alan";` to the line below `String[] names = new String[3];`?

```
String[] names = new String[3];
names[0] = "Alan";
```

▼ **Explanation**

The array `names` is initialized with no values. Then the first element is given the value `“Alan”`. That is why the program prints `Alan` followed by `null` two times.