Preface

Let's start by understanding exactly where Java starts—after some words of wisdom from Humpty Dumpty (by way of Lewis Carroll):

> "When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."
>
> "The question is," said Alice, "whether you can make words mean so many different things."
>
> "The question is," said Humpty Dumpty, "which is to be master—that's all."

—Charles Dodgson (aka Lewis Carroll), *Through the Looking-Glass*

Common usage of "application" and "app"

Often, we use the word "application"—even in a computing context—in an informal, imprecise way. The term may be applied to almost any significant bundle of processing functionality that we can run on a computer, particularly when it includes a GUI or can be launched from a GUI-based environment. (Similarly, we might distinguish "application" from "script", where the latter is generally used to refer to a smaller package of functionality, is often—but not always—launched from a console/terminal/command line interface, and usually doesn't have a GUI of its own). This common usage has even led to a related term, "app", being used to refer either to a tool that is more limited in functionality (but still with a GUI), or to one running on a more specialized, less general-purpose computing environment than a desktop workstation or laptop computer—for example, a phone or tablet.

Unlike Humpty Dumpty in *Through the Looking-Glass*, we rarely have the luxury of declaring new meaning for words, especially when we are entering a field (in this case, software development, and Java programming in particular) in which many terms already have domain-specific meanings. However, we can still become the master of such words, by treating those domain-specific meanings with care and attention. In this case, we'll begin to do so by becoming very specific in our use of the word "application", as it applies to Java.

Minimal elements of a Java application

In Java, *application* has a very specific meaning: It's a collection of one or more Java classes (and possibly interfaces), with at least one class (almost always declared `public`) in which there is a method called `main`, which in turn satisfies (at a bare minimum) these conditions:

- Declared with an access level of `public`.
- Declared with `static` scope.
- Returns a `void` result (i.e. no result at all).

- Has a single parameter of type `String[]` (array of `String`) or `String...` (the *varargs* form of a `String` array parameter declation).

(Note that a GUI is not included in these conditions. A Java application may or may not have a GUI—it might even run with no visible output of any kind.)

The `main` method serves as the *entry point* of the application. That is, when the application is run (by the Java application launcher, see below), execution of the application's code begins at the start of the `main` method. Thus, if we want to program some functionality in Java, and we want that functionality to be invoked as a Java application, that functionality **must** be programmed in the `main` method, or in some other methods (possibly residing in other classes or interfaces) that are called (directly or indirectly) from the `main` method.[1]

(We sometimes refer to a class containing these minimum necessary elements as a *Java application class*, *startup class*, or *main class*. We'll use these terms interchangeably, since they all mean essentially the same thing: a class with a `main` method that can be run to start a Java application.)

"Hello World!" redux

In the Oracle Java tutorials, and in our customized version of the "Hello World!" exercise, you wrote the code (or added to already generated code) to print a simple message to the console window. Your code looks something like this:

```
package edu.cnm.deepdive.prework;

public class HelloWorld {

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

}
```

We can see that "Hello World!" satisfies the requirements listed above. To summarize:

- We have a `public` class, called `HelloWorld` in this case.
- The `HelloWorld` class contains a method `main`, with
  - `public` access level.
  - `static` scope;
  - `void` return type;
  - One parameter (`args`) of type `String[]`.

Here, the code to write the "Hello World!" message to the console window (more generally, the *standard output device*) is included in the body of the `main` method, but it could have been written in a different method, which would then have been invoked from `main`, e.g.

```
package edu.cnm.deepdive.prework;

public class HelloWorld {

  public static void main(String[] args) {
    emitMessage("Hello World!");
  }

  private static void emitMessage(String message) {
    System.out.println(message);
  }

}
```

Note that in this second example, the "Hello World!" message is being written to standard output by the `emitMessage` method, which is invoked by the `main` method. Also note that `emitMessage` is *not* declared `public`. For the Java application launcher to be able to run a Java application, the `main` method must be visible outside the class—i.e. it must be `public`. However, if `main` then invokes other methods, it's not necessary for those methods to be visible to the Java application launcher.

The Java application launcher

The Java application launcher is an executable program, called `java` (`java.exe` or `javaw.exe` on Windows—the main difference between the two is simply that the latter does not associate the launched application with a console; thus, it's better suited to launching Java applications that include a GUI), whose primary function is to load and run Java applications.

To launch a Java application using the basic functionality of the Java application launcher, we run the latter from a command line, specifying the name of the `public` class containing the `main` method to run (i.e. the application entry point), with any additional options modifying the application's runtime environment specified before the class name, and any arguments that will be passed to `main` specified after the class name. The general syntax is

```
java [options] classname [arguments]
```

An alternative syntax, when the main class is located in a JAR (Java archive) file, and the latter contains a manifest referencing the main class, is

```
java [options] -jar jarfilename [arguments]
```

(In both forms, the square brackets indicate that that part of the command line is optional.)

The Java application launcher executes such a command by following these steps:

1. Load the *Java runtime environment* (JRE), setting its properties as dictated by any options specified on the command line;

2. Search for and load a `public` class with the name given on the command line;
3. Invoke the `main` method of the class, passing to it any arguments specified on the command line.

Given the above, we might try to run the "Hello World!" application above via:

```
java HelloWorld
```

(In Java, identifiers—including class names—are case-sensitive; thus, `HelloWorld` and `Helloworld`, for example, are not equivalent.)

But wait: In the code above (as in your "Hello World!" pre-work assignment), we created the `HelloWorld` class inside a package— `edu.cnm.deepdive.prework`, to be precise. Thus, the *fully qualified* name (the required form) of the class is actually `edu.cnm.deepdive.prework.HelloWorld`, so we would run the application from the command line this way:

```
java edu.cnm.deepdive.prework.HelloWorld
```

(There's a very important detail we haven't yet addressed: How does the Java application launcher go about finding the specified class? We'll get to this soon enough. In the meantime, if you look at the contents of the `out` and `src` directories inside your "Hello World!" project, you may figure out some portion of the answer.)

References

- C. Dodgson, "Humpty Dumpty," Through the Looking-Glass, ch. 6..
- H. Schildt, "Java Fundamentals," *Java: A Beginner's Guide* (8th ed.), ch. 1, pp. 2–31.

---

1. This is not precisely true: static initialization code runs as soon as a class is loaded, before any of its methods can be invoked. However, many classes don't include static initialization code; even when they do, the processing done by it is minor in most cases (e.g. assigning initial values to static fields). ↩