When is a Class a Program?

Do you find it odd that the Java programs we write, compile, and execute are themselves classes? We have just learned how to define our own classes, and, previously, we used classes in the Java API extensively. In most cases, we use classes to define and work with objects. But we always do this work within a class that we loosely referred to as a "Java program"? So, how do Java "programs" relate to "classes"? Understanding this is important in gaining a more complete and consistent picture of the Java language.

Let's pick any one of our Java demo programs as an example, such as RandomGen. Within the RandomGen.java file there is the definition of class called RandomGen, and in this class there is the definition of a method called main(). However, there is no constructor method called RandomGen(). In this sense, RandomGen is completely typical of our Java programs. So, how does the RandomGen "class" compare with a class such as City, defined at the beginning of this chapter? Until now, we have thought of classes like RandomGen as "programs", and classes like City as a blueprint for objects. These seem like two very different concepts. Since they are both "classes", however, we expect that they are examples of the same thing. And they are.

To harmonize our understanding of classes like RandomGen and City, let's focus our attention on two questions:

Can we "execute" the City class?

Can we "instantiate" a RandomGen object?

The answer to both questions is "yes". Before you dash off to your Java programming environment to check for yourself, note the following. A method was included in the definition of the City class that was omitted from the listing given earlier. This method is called main(). It was omitted for two reasons: to keep the listing short, and to avoid the present discussion. In fact, it is perfectly reasonable to include a main() method in all the classes we normally define as blueprints for objects. Once a class has a main() method, it is, of course, "executable" as a Java application. The main() method can serve as a simple self-test of the class definition. Of course, we also write more substantial methods to fully exercise and test classes, and these appear in separate classes with names like CityTest. Here's the main() method that was omitted from the earlier listing of the City class:

---

1 This practice is recommended by the designers of the Java language. See p. 56 of Arnold and Gosling's *The Java Programming Language* (2nd ed.).
// self-test method
public static void main(String[] args) {
  if (args.length != 2) {
    System.out.println("Usage: java City name pop");
    return;
  }
  String testName = args[0];
  int testPop = Integer.parseInt(args[1]);
  City c = new City(testName, testPop);
  System.out.println(c);
}

With this as part of the City class, it is perfectly reasonable to "execute" the City class:

```java
PROMPT> java City Tinyville 150
City [name: Tinyville pop: 150]
```

This illustrates why the answer to the first question above is "yes". Note that the instantiation of a City object occurs within the main() method which, itself, is defined within the definition of the City class. Besides this, the definition of the main() method is straightforward.

So, what about our second question? Can we instantiate a RandomGen object? We learned earlier that classes do not require constructor methods (although they are recommended). We also learned that all classes are subclasses of the Object class, and, therefore, inherit methods of the Object class, such as toString(). (Recall that the toString() method is automatically invoked to generate a string representation of an object when it is printed.) Keeping these points in mind, the following is a reasonable way to check if a RandomGen object can be instantiated:

```java
public class Test {
  public static void main(String[] args) {
    RandomGen rg = new RandomGen();
    System.out.println(rg);
  }
}
```

If the code above is put in a file called Test.java, it will compile and execute without errors. The output will look something like this:

```
RandomGen@e9ec971a
```

Of course, as an object that we might wish to instantiate and print, the RandomGen class is pretty useless. Nevertheless, the point is made. A class is a class, whether it is used as a Java application program or as a blueprint for objects.

**The main() Method**

Clearly, the main() method plays a special role in the Java language. Although details vary by system, whenever a Java application is launched there is a class that drives the application. This is the class that is passed as an argument to the Java byte code interpreter named java. The interpreter finds the compiled byte codes for the class, loads them into the Java Virtual Machine, then invokes the main() method, passing as it does, the command-line arguments. The
The `main()` method must be `public` (it is accessible to the interpreter), `static` (it is not invoked through an object), `void` (it returns nothing), and it must accept a string array as an argument. Hence, the `main()` method signature consistently appears as follows:

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

Since an application can have any number of classes, it can, by extension, have any number of `main()` methods (since each class can have a `main()` method.) Of course, only the `main()` method for the class driving the application executes.