Groovy. Differences with Java.

Groovy tries to be as natural as possible for Java developers. Groovy tries to follow the principle of least surprise, particularly for developers learning Groovy who've come from a Java background.

Here's a list of all major differences between Java and Groovy.

1. Default imports

All these packages and classes are imported by default, i.e. you do not have to use an explicit **import** statement to use them:

- java.io.*
- java.lang.*
- java.math.BigDecimal
- java.math.BigInteger
- java.net.*
- java.util.*
- groovy.lang.*
- groovy.util.*

2. Multi-methods

In Groovy, the methods which will be invoked are chosen at runtime. This is called runtime dispatch or multi-methods. It means that the method will be chosen based on the types of the arguments at runtime. In Java, this is the opposite: methods are chosen at compile time, based on the declared types.

The following code, written as Java code, can be compiled in both Java and Groovy, but it will behave differently:

```
int method(String arg) {
    return 1;
}
int method(Object arg) {
    return 2;
}
Object o = "Object";
int result = method(o);
```

In Java, you would have:

```
assertEquals(2, result);
```

Whereas in Groovy:

```
assertEquals(1, result);
```

That is because Java will use the static information type, which is that **o** is declared as an **Object** , whereas Groovy will choose at runtime, when the method is actually called. Since it is called with a **String** , then the **String** version is called.

3. Array initializers

In Java, array initializers take either of these two forms:

```
int[] array = {1, 2, 3};             // Java array initializer shorthand syntax
int[] array2 = new int[] {4, 5, 6}; // Java array initializer long syntax
```

In Groovy, the `{ ... }` block is reserved for closures. That means that you cannot create array literalas using Java’s array initializer shorthand syntax. You instead borrow Groovy’s literal list notation like this:

```
int[] array = [1, 2, 3]
```

For Groovy 3+, you can optionally use the Java’s array initializer long syntax:

```
// Groovy 3.0+ supports the Java-style array initialization long syntax
def array2 = new int[] {1, 2, 3}
```

4. Package scope visibility

In Groovy, omitting a modifier on a field doesn’t result a package-private field like in Java:

```
class Person {
  String name
}
```

Instead, it is used to create a *property*, that is to say a *private filed*, an associated *getter* and an associated *setter*.

It is possible to create a package-private field by annotation it with `@PackageScope`:

```
class Person {
  @PackageScope String name
}
```

5. ARM blocks

Java 7 introduced ARM (Automatic Resource Management) blocks like this:

```
Path file = Paths.get("/path/to/file");
Charset charset = Charset.forName("UTF-8");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

Such blocks are supported from Groovy 3+. However, Groovy provides various methods relying on closures, which have the same effect while being more idiomatic. For example:

```
new File('/path/to/file').eachLine('UTF-8') {
  println it
}
```

or, if you want a version closer to Java:

```
new File('/path/to/file').withReader('UTF-8') { reader ->
  reader.eachLine {
    println it
  }
}
```

6. Inner classes

The implementation of anonymous inner classes and nested classes follow Java closely, but there are some differences, e.g. local variables accessed from within such classes don't have to be final. Some implementation details, which is used for `groovy.lang.Closure` are piggy-backed when generating inner class bytecode.

6.1. Static inner classes

Here's an example of static inner class:

```
class A {
  static class B {}
}

new A.B()
```

6.2. Anonymous Inner Classes

```
import java.util.concurent.CountDownLatch
import java.util.concurent.TimeUnit

CountDownLatch called = new CountDownLatch(1)

Timer timer = new Timer()
timer.schedule(new TimerTask() {
  void run() {
    called.countDown()
  }
}, 0)

assert called.await(10, TimeUnit.SECONDS)
```

6.3. Creating Instances of Non-Static Inner Classes

In Java you can do this:

```
public class Y {
  public class X {}
  public X foo() {
    return new X();
  }

  public static X createX(Y y) {
    return y.new X();
  }
}
```

Before 3.0.0 Groovy doesn’t support the `y.new X()` syntax. Insted, you have to write `new X(y)`, like in the code below:

```
public class Y {
  public class X {}
  public X foo() {
    return new X()
  }
  public static X createX(Y y) {
    return new X(y)
  }
}
```

Caution though, Groovy supports calling methonds with one parameter without givinig an argument. The parameter will then have the value null. Basically the same rules apply to calling a constructor. There is a danger that you will write new X() instead of new X(this) for example. Groovy 3.0.0 supports Java style syntax for creating instances of non-static inner classes.

7. Lambda expressions and the method reference operator.

Java 8+ supports lambda expressions and the method reference operator (`::`):

```
Runnable run = () -> System.out.println("Run"); // Java
list.forEach(System.out::println);
```

Groovy 3 and above also support these within the Parrot parser. In earlier versions of Groovy you should use closures instead:

```
Runnable run = { println ’run’ }
list.each { println it } // or list.each(this.&println)
```