https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

1. Variable Scopes, Readability, and Lambda Expression

In Java, every *variable* declared has a *scope*. This means that the visibility and use of the variables must be restricted within the scope only. In case of a local variable, it is visible from the point of its declaration to the end of the method or code block it is declared in. It is a good practice to declare a variable close to the point of its possible use. This not only enhances readability of the code but also makes debugging simpler.

**try**

   Connection con = DriverManager.*getConnection*(DATABASE\_URL,

USERNAME,PASSWORD);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(SQL\_QUERY)

){

*//...*

}**catch**(SQLException e){

   e.printStackTrace();

}

Observe in the code snippet how the scope of the local variables is made limited within the blocks they are declared in. The variables are invisible as soon as the block ends. Also, the code becomes more intuitive, readable, and clean.

However, from Java 8 we can leverage lambda expressions to make the code more concise and intuitive. To illustrate the idea, note how we can club many utilities within a single line of lambda expression.

Integer[] nums={90,71,26,34,42,35,66,57,88,89};

Prints original numbers.

**final** List<Integer> l1 = Arrays.*asList*(nums);

System.***out***.printf(**"Original :%s%n"**,l1);

Now we use lambda expression to print numbers after sorting in ascending order.

**final** List<Integer> l2 = Arrays.*stream*(nums)

.sorted().collect(Collectors.*toList*());

System.***out***.printf(**"After sorting: %s%n"**,l2);

We can apply more utility, such as printing only those numbers that are greater than, say, 50, after sort them in ascending order.

**final** List<Integer> l3 = Arrays.*stream*(nums)

.filter(num -> num > 50).collect(Collectors.*toList*());

System.***out***.printf(**"Sorting only of numbers > 50**

**: %s%n"**, l3);

2. Class Fields

In Java, methods either belong to a *class* or to an *interface*. Therefore, there is a chance that local variables may be given the same name as a class member variable unintentionally by the programmer. The Java compiler, however, is able to pick the right one from the scope. Also, modern IDEs are sophisticated enough to identify the conflict. In any case, programmers themselves should be responsible enough to avoid such conflicts because the result can be quite disastrous. The following code illustrates how we get a very different result if we do not take care of conflicting variable names.

**public class** TestClass {

**private double commission** = 5.5d;

**public double** increaseCommission(**final double** newComm){

**double** commission = newComm;

      commission += commission;

**return** commission;

}

**public double** getCommission(){

**return commission**;

   }

**public static void** {TestClass m = **new** TestClass();

      System.***out***.println(m.increaseCommission(3.3));

System.***out***.println(m.getCommission());

}

}

Another way to avoid conflict with the field names of the class with local variables is by using the *this* keyword. For example *this.commission* would always refer to the class field only. But, experience shows it’s better to use different names altogether in methods other than constructors to avoid field name conflict.

3. Treating Method Arguments as Local Variables

In Java, a variable once declared can be reused. Therefore, the non-final local variables declared in the method arguments can also be reused with a different value. However, this is not a good idea because the variable sends as a method argument what is supposed to hold the value it has brought, the original value that the method with work upon. If we change the value, we’ll completely lose the original content that is brought forth in the method. Instead, what we must do is copy the value to another variable and do the necessary processing. We, however, can completely restrict and make the argument variable a constant by using the *final* keyword as follows.

**public double** calculate(**final double** newVal){

**double** tmp = newVal;

   // ...

**return** tmp;

}

4. Boxing and Unboxing

In Java, boxing and unboxing are the flipside of the same technique called *conversion* between primitive types and their corresponding wrapper classes. Wrapper classes are nothing, but are the “class” version of primitive types such as *int* to *Integer*, *float* to *Float*, *double* to *Double*, *char* to *Character*, *byte* to *Byte*, *Boolean* to *Boolean*, and so on. Boxing is converting, say, *int* to *Integer* and unboxing is converting back from *Integer* to *int*. The compiler often performs the conversion behind the scenes; this is called autoboxing. But, sometimes, this autoboxing may produce an unexpected result and we must be aware of that.

**public static void** update(**final double** newVal){

   // ...

}

**final** Double val = **null**;

*update* (val);

Java does not complain during compilation, but as we execute this code, *NullPointerException* is thrown. This is an example to show that autoboxing does not always produce the result we want. We should be aware and give a thought if autoboxing would suffice in the present context or not.

5. Interfaces

Because interfaces are not tied to any method implementation (except default methods), we must use them adequately where we do not need concrete classes. Interfaces are like contracts with ample freedom and flexibility to lay out the contract. Concrete classes can avail themselves of the flexibility to enrich and replenish its business agenda. This is particularly true and can often be seen when the implementation involves an external system or services.

**package** com.mano.examples;

**public interface** Payable {

**public double** calcAmount();

}

**package** com.mano.examples;

**public class** Invoice **implements** Payable {

**private** String **itemNumber**;

**private** String **itemName**;

**private int quantity**;

**private double unitPrice**;

**public** Invoice(String itemName, **int** quantity,

**double** unitPrice) {

*// ...*

   }

**public int** getQuantity() {

*// ...*

**return quantity**;

   }

**public double** getUntPrice() {

*// ...*

**return unitPrice**;

   }

   @Override

**public double** calcAmount() {

**return** getQuantity()\*getUntPrice();

   }

}

6. Strings

No other types in Java have be extended as much as *String*. Java strings are represented in UTF-16 format and are immutable objects. This means that every time we perform an operation like concatenation, that needs modification of the original string, a new string object is created. This intermediate string object, created only to perform the needed operation, is a waste and inefficient. This is because intermediary object creation is extraneous; it involves garbage collection, although we can avoid it. There are two companion string classes, called *StringBuffer* and *StringBuilder*, which can aptly facilitate the kind of string manipulation we may need. These two classes are built for that. The only difference between *StringBuffer* and *StringBuilder* is that the former is thread-safe. We can use these two classes whenever need extensive string manipulation rather than using the immutable *String* instance.

StringBuilder stringBuilder = **new** StringBuilder();

stringBuilder.append( **"Hello"** );

stringBuilder.append( 108 );

stringBuilder.deleteCharAt( 0 );

stringBuilder.insert( 0, **"h"** );

stringBuilder.replace( stringBuilder.length() - 3,

stringBuilder.length(), **"."** );

System.***out***.println(stringBuilder);

7. Naming Conventions

Java naming conventions are set of rules to make Java code look uniform across Java projects and the library. They are not strict rules, but a guideline to adhere to as a good programming practice. It is, therefore, not a good idea to violate the sanctity of the code uniformity either due to haste or rebellion. The rules are pretty simple.

* Package names are types in lowercase: *javax.sql, org.junit, java.lang*.
* Class, enum, interface, and annotation names are typed in uppercase: *Thread, Runnable, @Override*.
* Methods and field names are typed in a camel case: *deleteCharAt, add, isNull*.
* Constants are types in uppercase separated by an underscore: SIZE, MIN\_VALUE.
* Local variables are typed in camel case: *employeeName, birthDate, email.*

These conventions have become so much apart of Java that almost every programmer follows them religiously. As a result, any change in this looks outlandish and wrong. An obvious benefit of following these conventions is that the style of code aligns with the library or framework code. It also helps other programmers to quickly pick up the code when they have to, therefore leveraging overall readability of the code.

8. Standard Libraries

Java is known for its rich set of libraries. However, it is not that every library is perfectly designed but for the most part of it they are optimal. Java releases new libraries every now and then and improves the existing ones actively. Therefore, one should always use classes, methods, interfaces, enums, and annotations from the library as much as possible. This can reduce production time considerably. Moreover, the features included are well tested. It is better to use a them from the library whenever required rather than reinvent the wheel.

Because the Java platform evolves frequently, it is extremely important that we keep an eye on the features which we include from the third-party libraries or frameworks are already present in the Java Standard Library. The rule of thumb is to first exhaust the in-house resource before culling support from external sources.

9. Immutability

The concept of immutability is very important. We must decide whether the classes we intend to design can be made immutable or not, because immutability guarantees that it could be used almost everywhere without any trouble from concurrent modification. Unfortunately, not all classes can be designed as immutable. But, make sure we must do so whenever we can. This makes life much easier. You can thank me later.

10. Testing

Test driven practices (TDD) are a symbol of quality code among the Java community. Testing is a part of modern-day Java development, and the Java Standard Library has the Junit framework to assist in that direction. So, a budding Java programmer should not shy away from writing test cases with the code. Try to keep tests simple and short, focusing on one thing at a time. There can be hundreds of test cases in production environment. An obvious benefit of writing test cases is that they provide immediate feedback of the features under development.