Tips on Java

Noman Ahmed Sheikh
http://www.cse.iitd.ac.in/~cs1130244

Abstract I have compiled this material while reading some of the commonly recommended books for Java. I see this material as a compendium for the resources I have read. Aimed to save the readers time by presenting the most important point in sweet and simple manner. While writing this text, the motivation was to keep track of some specific points concerning programming in Java. These points often remain unnoticed by programmers.
Hope you will enjoy this, Cheers!!!

Noman Ahmed Sheikh
Indian Institute of Technology Delhi, India, e-mail: nomanahmedsheikh@outlook.com

1 Head-First Java

1.1 Basics

1.1.1 Variables

- Instance variables[1] always get a default value. If you don't explicitly assign a value to an instance variable, or you don't call a setter method, the instance variable still has a value! Default values for various datatypes are as follows:

integers	0
floating points	0.0
booleans	false
references	null

- Local variables[2] do NOT get a default value when declared! The compiler complains if you try to use a local variable before the value is initialized. **Local Variable must be initialized before use**
- Method parameters are mostly treated as local variables but you will never get a compilation error saying that the method parameter is un-initialized. Because while calling the method, compiler makes sure that all the parameters of the method are initialized.

1.1.2 Short-circuit operators

- && and || are Boolean operators also known as short-circuit operators.
- While using && operator in `cond1 && cond2 && cond3`, Condition1 is evaluated first and if it turns out to be *false* then condition2 and condition3 are not evaluated and skipped.
- While using || operator in `cond1 || cond2 || cond3`, Condition1 is evaluated first and if it turns out to be *true* then condition2 and condition3 are not evaluated and skipped.
- & and | are nonshort-circuit operators and they evaluate the entire expression. They operate on bit level and not the boolean level.

1.1.3 Packages

- If you are a C/C++ programmer, an `import` is not same as `include`. In C/C++, `#include` will take lot of time in compilation whereas in Java `import` does not add very much to the compilation time.

---

[1] Instance variable are variables that are declared inside a class but not in the method

[2] Local Variables are variables that are declared inside the method.

- Remember, you get the `java.lang` package sort of pre-imported for free. Because the classes In `java.lang` are so fundamental, you don't have to use the full name. There is only one `java.lang.String` class, and one `java.lang.System` class, and Java well knows where to find them.

1.1.4 Object Hashcode

There is a method `int hashcode()` which is present in Java object class which returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by `java.util.Hashtable`. This method is used in bucketing operations of hashtables to assign bucket number. Whenever we do a find query on hashtable, bucket corresponding to `hashcode` value is checked. If there are more than 2 Objects present in the same bucket then `equals(Object)` method is called on each one of them to find the exact Object.

- Whenever this method is invoked on the same object more than once, it must always return the same integer value in the entire execution cycle of the application. Value can change across different execution of the application.
- If two Objects are equal with respect to `equals(Object)` method, then it is mandatory for `hashcode()` method to return same value for both objects.
- Note that it is quite possible that two Objects which are not equal with respect to `equals(Object)` method have same hashcode value. Although it is always preferable to have a hashcode— method which assigns unique value to non-equal Objects.
- If `toString()` method is not implemented for an object, then calls such as `System.out.print(Object)` will print hashcode value for the Object

1.2 Object-Oriented Concepts

1.2.1 Encapsulation

- Encapsulation in java is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines.
- We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
- It provides you the control over the data. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

1.2.2 Inheritance

- subclass can add new methods and instance variables of its own, and it can override[3] the methods it inherits from the superclass.
- Instance variables are not overridden because they don't need to be. They don't define any special behaviour and subclass will have its own value of instance variable.
- **IS-A, HAS-A tests** are used to check and verify whether the object extends other object?
  - If class B extends class A, then class B **IS A** class A.
  - IS-A can be thought of as a relation in Set Theory. IS-A is relation which is transitive[4] and anti-commutative[5].
  - HAS-A test is used to design instance variables for a class.
  - A Bathroom HAS-A Tub. That means Bathroom has a Tub as instance variable.
- In a subclass, public members of superclass are inherited but private members of superclass are not inherited.

Que: Can I stop a class from being subclassed?
Ans: Yes, there are 3 possible ways:

1. A non-public class can only be subclassed by the classes present in the same package. The classes present in different package won't be able to see a non-public class. Let alone the task of subclassing.
2. Use of modifier `final`. A final class means that it is the end of inheritance hierarchy. Nobody can extend a final class. If you want to stop a particular method from being overridden, mark that method as final using `final` modifier.
3. If all the constructors of the class are private.

1.2.3 Polymorphism

- The reference type can be a superclass of the actual object.
- In other words, whenever you declare a reference variable, any object that is subclass (lower in hierarchy) can be assigned as value of that reference variable. For example: `Animal animal = new Dog()`
- In above example, `animal.eat()` will automatically call the `eat()` method declared in `Dog` class.

---

[3] **Overriding:** subclass can implement its own specific version of a inherited method. This modified function will override the function implemented in superclass with the same name.

[4] **Transitivity:** Class B extends A and Class C extends B, then we can say that B *IS-A* A and C *IS-A* B. Which implies that C *IS-A* A

[5] **Anti-commutative** means that if B *IS-A* A then it cannot happen that A *IS-A* B

1.2.4 Abstract and Concrete Classes

Sometimes we want a certain class not to be instantiated[6] anywhere in the code because it does not contain sufficient information to be represented as an Object. Take `Animal` class for example. No one knows what an animal looks like. It is just an abstract representation with not the sufficient amount of details.

- By marking the class as `abstract`, the compiler will stop any code, anywhere, from ever creating an instance of that class.
- However, abstract classes can be used as Reference[7] for the purpose of polymorphism. In other words, we can use `Animal` class as a reference for `Dog` object
- Concrete class on the other hand are specific enough to be instantiated.

1.2.5 Abstract Method

An abstract class means that the class must be extended and an abstract method means that the method must be overridden.

- An abstract method has no body
  `public abstract void eat();`
- An abstract method can only be declared in an abstract class.
- An abstract class can have abstract as well as non-abstract methods.
- In class inheritance tree, the first concrete method below the abstract method class has to implement the method.

1.3 Objects

Also called as mother of all classes in Java

- Every class that explicitly does not extend any other class, implicitly extends `Object` class.
- Many of the built-in data structures like `ArrayList`, `HashMaps` etc are implemented with `Object` class and they are simply leveraging the beauty of polymorphism
- This `Object` *superclass* has some methods of its own which are very useful for almost every class. These methods include `equals()`, `getClass()`, `hashCode()`, `toString()`
- `Object` class is a non-abstract class which means that we can create instances of `Object` type.

---

[6] **Instantiation:** new *Animal()*

[7] **Reference:** *Animal* animal

- If you declare an ArrayList as `new ArrayList<Object>()`, then you can add objects of any type in the arraylist, be it Dog, Cat or whatever. But when you retrieve an element from this ArrayList, it will be of the type `Object`.
- But if you still want to access that object as a `|Dog|` then you can perform casting. But before performing casting, you need to be sure that the object indeed is a Dog, otherwise it will give *ClassCastException*[8] at runtime.
  `Dog dog = (Dog) objectVariable`
- If you are not sure whether the object is an instance of Dog class you can check using

```
.if (o instanceof Dog) {
.     Dog dog = (Dog) o;
.}
```

The compiler checks the class of the reference variable not the class of the actual object at the other end of the reference. And this is the reason for runtime exception instead of compile-time exception

1.3.1 Deadly Diamond of Death

Case of multiple inheritance: Suppose Class A extends Class P and Class Q both. P has burn() method of its own and Q has burn() method of its own. When we perform A.burn(), which burn method will get called? There is ambiguity.

Java does not support multiple inheritance. In Java, a class cannot extend multiple classes. Multiple inheritance is supported by C++. *How C++ handles this problem is sheer beauty*[9]

1.3.2 Interfaces

- an interface is 100% pure abstract class. A class with all methods abstract.
- All interface methods are implicitly abstract and public and therefore you do not need to specify it separately.

Que: Do you want an object to be able to save its state to a file?
Ans: Implement the serializable interface.
Que: Do you need objects to run methods in a different thread?
Ans: Implement the Runnable interface.

`super` keyword is used to reference super class of object. Sometimes we need

[8] More detail on Exceptions given in later subsections

[9] link: https://stackoverflow.com/questions/137282/how-can-i-avoid-the-diamond-of-death-when-using-multiple-inheritance

to extend (and not override) methods of superclass. This can be used in such cases. This is most commonly used in constructors (explained later).

1.3.3 Method Stacks and Space Allocation

The execution of a process maintains a stack which contains the necessary information for the processor to perform instructions.

- Method on top of stack is always the current executing method
- Each method has a stack-frame on stack when it is in action
- All objects are **always** stored on Heap[10]. Non primitive variables only hold the reference to the object and not the actual object
- If a local variable is reference to an object, then the variable will go on stack[11] and object goes on heap.

Local variables live on stack. Instance variables are present inside object and objects live on heap. This implies that instance variables also live on heap. We will discuss about Heap vs Stack later in much detail.

While doing `Dog dog = new Dog()`, JVM finds space for a `Dog` object on heap so that its instance variables can live. The required space is computed based on the datatypes of instance variables. If any instance variable refers to another object, only reference is stored. So there is no need to recursively unwrap the space requirements of all the non-primitive datatypes. For example : `Dog` class has 2 instance variables with space requirements as follows:

Dog class

int	height	4 bytes
Person	owner	20 bytes

Instance variable *person* is itself an object but that does not mean that an instance of `Dog` class will occupy 4 + 20 = 24 bytes. Instance of `Dog` class will only have the reference for `Person` class plus 4 bytes for primitive datatype `int`

1.3.4 Constructors

Constructor is a code that runs when you instantiate an object. In other words, the code that runs when you say `new` on a class type. Every class you create has a constructor, even if you don't write one. Constructors have the same name as the class name. A sample constructor for our `Dog` class is given below

[10] **Heap memory** is part of memory that is used by all the parts of the application

[11] **Stack memory** is used only by one thread of execution

```
.Dog () {
.    this.height = 600;
.    this.owner = new Person();
.}
```

Java allows methods to have same name as the class name. But that does not make that method constructor of that class. Return type distinguishes methods and constructors. It is mandatory for methods to have a return type while constructors do not have return type

- If you do not write any constructor, Java compiler will give you a default no argument constructor. But if you write a constructor that takes arguments and you still want a no-argument constructor, you will have to write it yourself. Compiler will not give default no-argument constructor.
- It is a good practice to provide no-argument constructor if you can, to make life of programmer easy.
- Instance variables are assigned a default value, even when you don't explicitly assign one in constructor. Default values are similar to those mentioned in Section 1.1.1.

1.3.5 Constructors and Inheritance

Whenever a subclass is instantiated, the instance variables of all of its super classes in hierarchy also gets allotted space in heap. When a new object is made, all the constructors in that object's inheritance tree must run. Because they will create the super classy part of that object.

- Even abstract class has constructors because they are also super classes and can have instance variables
- Compiler put `super()` call in every constructor as the first instruction even if you don't put it. This will make sure that instance variables of super classes are allotted space on heap before the creation of instance of the current class.

We can invoke one overloaded constructor from another using `this()` call but `this()` should be the first line of constructor.

- A constructor can have a call to `super()` or `this()` but never both. Because both of them need to be the first instruction of the constructor and at a time only one can be used
- It is good practice to have a single constructor called as *Real Constructor* which actually calls `super()`.

1.3.6 Life of Objects

Instance variable lives as long as object lives on heap. And an object is alive on heap as long as there are live references to it. After last live reference is gone, object becomes eligible for garbage collection.[12]

Local variable on the other hand, lives only within the method that declared it. Note that Life[13] and Scope[14] may or may not be same for local variables.

1.3.7 Wrapping Primitives and Autoboxing

When you need to treat a primitive like an object, wrap it. There are wrappers for all primitive types. For example `Integer`, `Long`, `Boolean` etc. Before Java 5.0 only objects were allowed in Collections like `ArrayList` and `HashMap`.

Autoboxing

- This feature added to Java 5.0 does the conversion from primitive to wrapper object automatically.
- This makes wrappers indistinguishable from their primitive counterparts. They can be used almost anywhere as replacement.
- Besides acting like normal class, wrappers have some useful static utility methods such as `parseInt()` etc. Every such method that parses a string can throw a `NumberFormatException` at runtime.

1.4 Identifiers

There is no such thing as global in Java as you might have seen in C/C++.

If you have a method whose behaviour does not depend on a instance variable value, then why to waste precious heap space by loading entire object every time. For example `Math` class have all the methods as `static` with no instance variables. If you try to create an instance, the compiler will throw error.

- Keyword `static` lets a method run without any instance of the class. If a method is `static`, it means its behaviour does not depend on instance variables.

---

[12] **Garbage Collection** is a process of reclaiming the Runtime unused memory automatically. In other words, it is a way to destroy the unused objects

[13] Local variable is alive as long as its stack-frame is on the stack (until method completes)

[14] When its own method calls another method, variable is alive but it is not in scope

1.4.1 Static Methods

You can't access non-static variables or methods from inside a static method.

- Non-static methods are called using object's reference while static methods are called by class's name. See the difference below:

  t2.size() vs *Animal.size()*

- Therefore, if you access an instance variable from a static context, compiler will have no way of knowing, which object's variable to access to.
- Although, its legal to call a static method using an object's reference variable.

1.4.2 Static Variables

Static Variables gives you a value that is shared by all the instances of a class. In other words, one value per class instead of one value per instance.

Initialization : Static variables in a class are initialized before any object of that class can be created. Static Variables in a class are initialized before any static method of class runs.

1.4.3 Static imports

Static imports exists to save you some typing. For example:

```
import static java.lang.System.out
import static java.lang.Math.*
```

Here * is a wild card.

Static imports can make your code confusing to read. They can also create name conflicts very easily.

1.4.4 Final Variables

A variable marked `final` means that once initialized, it can never change. In other words `static final` variables are constants. It is good practice to name constant variables in all capitals.

There are 2 ways to initialize a `final static` variable

1. At declaration time
2. In static initializer [15]

[15] Page 282 in Head First Java

1.4.5 Finals at a glance

- A final variable means you cannot change its value.
- A final method means you cannot override the method.
- A final class means you cannot extend the class

Note that it is redundant to mark the method as final, if the class is already final.

1.5 Exception Handling

TODO : Typing left

2 Java Environment

Terms like JVM, JRE and JDK are commonly used in the context of Java and are quite confusing at times. Here is a brief description.

2.1 Java Virtual Machine (JVM)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

2.1.1 What is JVM?

It is:

- **A specification**, where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- **Runtime Instance**, Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

2.1.2 Functions

The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

2.2 Java Runtime Environment (JRE)

2.3 Java Development Kit (JDK)

3 Effective Java

3.1 Java Generics

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time. Lets start with basic terms and definitions:

3.1.1 Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets ($<$ and $>$) that precedes the method's return type. For example: `public static <E> void printArray( E[] inputArray )`
- Each type parameter section contains one or more type parameters[16] separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

3.1.2 Generic Classes

A class or interface whose declaration has one or more type parameters is a generic class or interface. These classes are known as parameterized classes or parameterized types because they accept one or more parameters. Each generic type[17] defines a set of parameterized types, which consist of the class or interface name followed by an angle-bracketed list of actual type parameters corresponding to the generic types formal type parameters.

---

[16] **Type Parameter:** In function call shown above, E is the type paratmer

[17] Generic classes and interfaces are collectively known as generic types

Each generic type defines a **raw type**, which is the name of the generic type used without any accompanying actual type parameters.For example, the raw type corresponding to `List<E>` is `List`

3.2

TODO : Notes Incomplete

4 Design Patterns in Java

TODO : Notes Incomplete

4.1 Creational Design Patterns

TODO

4.1.1 Singleton Pattern

4.1.2 Factory Pattern

4.1.3 Abstract Factory Pattern

4.1.4 Builder Pattern

4.1.5 Prototype Pattern

4.2 Structural Design Patterns

TODO

4.2.1 Adapter Pattern

4.2.2 Composite Pattern

4.2.3 Proxy Pattern

4.2.4 Flyweight Pattern

4.2.5 Facade Pattern

4.2.6 Bridge Pattern

4.2.7 Decorator Pattern

4.3 Behavioral Design Patterns

TODO

4.3.1 Template Method Pattern

4.3.2 Mediator Pattern

4.3.3 Chain of Responsibility Pattern

4.3.4 Observer Pattern

4.3.5 Strategy Pattern

4.3.6 Command Pattern

4.3.7 State Pattern

4.3.8 Visitor Pattern

4.3.9 Interpreter Pattern

4.3.10 Iterator Pattern

4.3.11 Memento Pattern

5 Concurrency in Java

TODO : To Read

Appendix

No tables/figures as of now :p

References

To be added later - Refer notes list