How my code satisfies the “Effective Java Code Principles” according to Joshua Bloch’s “Effective Java” book.

Item 1:   Consider static factory methods instead of constructors

All of objects creation were through factory methods written in factory classes in com.atypon.factory package such as:

- `BlockchainFactory`, for creating a `Blockchain`.
- `BlockFactory`, for creating a `Block`.
- `ClientFactory`, for creating a `Client` and a `ClientSocket`.
- `KeyFactory`, for creating `Public & Private Key` pairs as well as unique IDs for `Transactions`.
- `TransactionFactory`, for creating `Transactions` and `MinedTransactions`.
- `UserFactory`, for creating a `User`.

or inside the class using static methods if the class was a GUI Interface in `com.atypon.gui` package such as:

- `runLogin(…)`, which instantiates and runs the login window.
- `runWindow(…)`, which instantiates and runs the main window.
- `runNewTransaction (…)`, which instantiates and runs the new transaction window.
- `runShowTable (…)`, which instantiates and runs the show table window.

Item 2:   Consider a builder when faced with many constructor parameters

I didn’t have to use such types because most of the objects had either 1 or 2 constructors, and that’s especially because I tried to keep the objects and fields immutable, so the data are thread safe.

Item 3:   Enforce the singleton property with a private constructor or an enum type

There was no need for a singleton in my project, so I didn’t create any.

User class could have been made a singleton as it doesn’t make any sense that you create multiple users, but I benefit from creating multiple users while adding the unit testing.

Item 4:   Enforce non-instantiability with a private constructor

For all the utility classes in the package `com.atypon.utility` I made sure the classes are final and have their default constructor private to enforce non-instantiability.

I also used this idea to enforce using the factory methods instead of straight up calling the constructors.

Item 5: Prefer dependency injection to hardwiring resources

In the server, the connections to the database were handled through a DAO object instead of directly working with the static methods of the `DatabaseUtility` class.

Item 6: Avoid creating unnecessary objects

In general, I tried to create as little objects as possible, anyway `Intellij` detects and warns most of unnecessary objects creations.

Item 7: Eliminate obsolete object references

I mostly used `Lists` which is supposed to handle the memory issues, I used `Arrays` when creating the tables in the interface, but that one didn't have any memory leak as its size was fixed to exactly what I needed.

Item 8: Avoid finalizers and cleaners

My code didn't use any of them.

Item 9: Prefer try-with-resources to try-finally

All the `try-catch` statements that I used with `Closable` objects were `try-with-resources`, such as:

- `Socket` used to establish a connection on the client side.
- `ServerSocket` used to establish a connection on the server side.
- `ObjectOutputStream`, `ObjectInputStream` used to read/write objects from files and connections.

Item 10: Obey the general contract when overriding equals

All the `equals(…)` methods were implemented through `Intellij` auto-generated code which insures all the general contract for overriding `equals(…)`.

Item 11: Always override hashCode when you override equals

All the classes were `equals(…)` were overridden, `hashCode(….)` was overridden as well.

Item 12: Always override toString

All the instantiatable classes has a `toString()` method.

Item 13: Override clone judiciously

I only have the `Blockchain` objects as ` Cloneable` but the cloning here is safe, because a blockchain only consists of a vector which can be cloned using the copy constructor.

Item 14: Consider implementing Comparable

I didn't need to implement `Comparable`, the only things that needs comparing is the amount for the `Transactions` which already use the built-in class `BigDecimal`.

Item 15: Minimize the accessibility of classes and members

The access for all the classes and fields were set to minimum, unless some of them could be used later on if the application were to be expanded or the basic classes for the `Blockchain` were to be used in another project, that's why none of the functions had a `package-private` access although some of them were only used in the same package.

Item 16: In public classes, use accessor methods, not public fields

All non-final fields were accessed through `getters/setters` in all the classes (if any outer access is needed), the exceptions are the final static fields holding the constants.

Item 17: Minimize mutability

I tried to keep my classes and fields immutable, such as:

- `Transaction`, this class is Immutable, once created there is no way to change its values.
- `MinedTransaction`, this class is also Immutable, the extension upon `Transaction` are all un-modifiable.
- `Block`, is almost Immutable, the only thing that can change is the `nonce` which is only changed while creating a `MinedBlock` instance.
- `Blockchain`, the class is mutable, because it would be too inefficient to re-create the whole chain when adding or removing any blocks from it.
- `ClientSocket` is Immutable where the data cannot be changed after object creation.

Item 18: Favor composition over inheritance

I only figured that `MinedTransaction` is better inheriting `Transaction` over doing a composition, since a `MinedTransaction` is a `Transaction`, and it should have the same methods.

In all other scenarios I used composition where a `Block` has a `Transaction` and a `Blockchain` has a list of `Blocks` … etc.

Item 19: Design and document for inheritance or else prohibit it

All my classes are finals, so they cannot be inherited.

Item 20: Prefer interfaces to abstract classes

I created `Blockable` interface to upper-limit the generic data type for the `Block` and `Blockchain` instead of using an abstract class, thus being able to use `MinedTransaction` object as an extension to `Transaction` while being a `Blockable` object.

I did the same with `ClientSocketDAO` interface in the Server application.

Item 21: Design interfaces for posterity

I don't see why my interfaces would have any problems with this.

Item 22: Use interfaces only to define types

The interfaces `Blockable` and `ClientSocketDAO` defines a type only.

Item 23: Prefer class hierarchies to tagged classes

Not applicable to the project.

Item 24: Favor static member classes over non-static

The only inner-class I had was `Client` which is inside `User` and it's not static because there is no use for `Client` without a user to handle its connections.

Item 25: Limit source files to a single top-level class

`Intellij` doesn't even allow that, so I have only one top-level class in all the files.

Item 26: Don't use raw types

In all the uses of `Vector`, `BlockingQueue` and `ComboBox` or the types I defined as Generic like `Blockchain` and `Block` I specified exactly what the type is.

Item 27: Eliminate unchecked warnings

The warnings I didn't eliminate are the potential `package-private` functions which I already explained why I didn't solve, the `clone()` method in `Blockchain` which says "No call to `super.clone()`" because I don't need to call the super version, other warnings were receiving a generic object through a socket or from a file where I guarantee the type of it, still it's surrounded by try-catch to make sure no errors occur if someone messed with the files or the connection.

Item 28: Prefer lists to arrays

Arrays were only used to create `Model` for the table in `ShowTable` GUI class.

Item 29: Favor generic types

I used generic types in the `Blockchain` to make it applicable to any class by just implementing `Blockable` interface.

Item 30: Favor generic methods

I didn't need that in my project.

Item 31: Use bounded wildcards to increase API flexibility

That's the actual use of `Blockable`.

Item 32: Combine generics and varargs judiciously

Not applicable to the project.

Item 33: Consider typesafe heterogeneous containers

Not applicable to the project.

Item 34: Use enums instead of int constants

Not applicable to the project.

Item 35: Use instance fields instead of ordinals

Not applicable to the project.

Item 36: Use EnumSet instead of bit fields

Not applicable to the project.

Item 37: Use EnumMap instead of ordinal indexing

Not applicable to the project.

Item 38: Emulate extensible enums with interfaces

Not applicable to the project.

Item 39: Prefer annotations to naming patterns

Tagged the Tests and Overridden methods with proper annotations.

Item 40: Consistently use the Override annotation

All Overridden methods were tagged with the `Override` annotation.

Item 41: Use marker interfaces to define types

Used in `Blockable` interface.

Item 42: Prefer lambdas to anonymous classes

All the creations of anonymous classes in the GUI for the listeners were changed to lambdas when possible like the ActionListener for the buttons.

Item 43: Prefer method references to lambdas

Not applicable to the project.

Item 44: Favor the use of standard functional interfaces

Not applicable to the project.

Item 45: Use streams judiciously

The use of streams was at bare minimum, they were only used to send/receive data from/to other clients or to read/write objects into files.

Item 46: Prefer side-effect-free functions in streams

Not applicable to the project.

Item 47: Prefer Collection to Stream as a return type

Not applicable to the project.

Item 48: Use caution when making streams parallel

Not applicable to the project.

Item 49: Check parameters for validity

The values such as `Amount`, `Port`, `IP Address` are always validated before taken into account, other values are all auto-generated.

Item 50: Make defensive copies when needed

Multiple defensive copies are made of the `Blockchain` and the `ClientSocket` vector inside the user when working on them because there’s a very slim chance that if the system overheads that the values change faster than the processer can handle, so safe copies prevent that.

Item 51: Design method signatures carefully

The names were chosen carefully for all the methods.

Item 52: Use overloading judiciously

Overloaded functions used in `Blockchain` and in the factory classes were well defined and explicit.

Item 53: Use varargs judiciously

Used only for sending multiple strings to hash and combine them.

Item 54: Return empty collections or arrays, not nulls

In the communications between the clients, null was only returned once to indicate that the connection failed to return values because empty collection means that the other side responded but with nothing.

Item 55: Return optionals judiciously

Not applicable to the project.

Item 56: Write doc comments for all exposed API elements

Doc comments were written for all exposed and non-exposed elements.

Item 57: Minimize the scope of local variables

I tried doing that by defining the variables inside the loops and if statements, as the code will look cleaner, and for optimization, most modern compilers handle these optimizations perfectly.

Item 58: Prefer for-each loops to traditional for loops

Whenever there was a way to user `for-each` loop instead of the traditional `loop` I used it, mostly to loop over the clients or the blocks.

Item 59: Know and use the libraries

Used the available libraries to generate Hashes and Public & Private Keys and use them, other than that there was no need for any libraries.

Item 60: Avoid float and double if exact answers are required

I used BigDecimal as my main data type to store the data of the transactions.

Item 61: Prefer primitive types to boxed primitives

I always used primitive types since I didn’t need the boxed ones.

Item 62: Avoid strings where other types are more appropriate

Not applicable to the project.

Item 63: Beware the performance of string concatenation

Not applicable to the project.

Item 64: Refer to objects by their interfaces

When working on sending and receiving objects, Serializable type was used.

Item 65: Prefer interfaces to reflection

Not applicable to the project.

Item 66: Use native methods judiciously

Not applicable to the project.

Item 67: Optimize judiciously

Some optimizations for the project are possible but were postponed until the project is fully functional.

Item 68: Adhere to generally accepted naming conventions

I always followed the naming conventions while naming classes, interfaces, fields, variables, methods, packages…

Item 69: Use exceptions only for exceptional conditions

I only throwed exceptions when creation of broken objects happened, like created a signed transaction but the signature is invalid or mining a block but after mining it, the proof of work isn’t correct, these scenarios should only happen if something wrong goes with the program.

Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

The exceptions explained above were runtime exceptions, no checked exceptions were needed.

Item 71: Avoid unnecessary use of checked exceptions

Didn’t use any checked exceptions as they weren’t needed.

Item 72: Favor the use of standard exceptions

The runtime errors I used were of type `RuntimeError` because there was no need for extending it.

Item 73: Throw exceptions appropriate to the abstraction

Not applicable to the project.

Item 74: Document all exceptions thrown by each method

Exceptions are only thrown in the factories and are well documented.

Item 75: Include failure-capture information in detail messages

The stack trace is always printed in the log in all the `try-catch` statements.

Item 76: Strive for failure atomicity

Not applicable to the project.

Item 77: Don’t ignore exceptions

Exceptions are never ignored and well handles, either by showing a message to the user or handling the response to the function that called this function.

Item 78: Synchronize access to shared mutable data

In class User, `transactionPool`, `clients` and `blockchain` may be accessed and modified from multiple threads, so all code blocks using them were synchronized and locked on the object being accessed.

Item 79: Avoid excessive synchronization

In many places copies of the data were made (using synchronized blocks) before using them to avoid synchronizing blocks that takes time to execute like run connections to other clients.

Item 80: Prefer executors, tasks, and streams to threads

The multi-threading used to listen, send and receive data from/to other users ran on a ExecutorService with fixed number of threads to avoid over-heading the system with too many threads if many other users tried to connect to this user.

Item 81: Prefer concurrency utilities to wait and notify

There was no need to use wait and notify or any other alternatives since thread safe data structure like `Vector` and `BlockingQueue` were used.

Item 82: Document thread safety

Documentation added for thread safety for the classes that are thread safe such as:

- `Transaction`, Immutable.
- `MinedTransaction`, Immutable.
- `Block`, Immutable once mined, since `nonce` is only changed while mining.
- `ClientSocket`, Immutable.
- `User`, Designed to be thread safe, the blocks that may cause concurrency problems were synchronized.

Item 83: Use lazy initialization judiciously

Only used in the Sever, which doesn't have any multi-threading before creating the object.

Item 84: Don't depend on the thread scheduler

There was no depending on the thread scheduler.

Item 85: Prefer alternatives to Java serialization

I did use serialization to save the user's information locally on the machine and to send the data over from one client to the other, a future optimization would use something like a JSON object to send or store data.

Item 86: Implement Serializable with great caution

Again, fixing this would be using a more proper way of sending objects.

Item 87: Consider using a custom serialized form

The components of the objects I used were already serializable so there was no need to use a custom form, I wrote a readUser function to make sure the user read is valid.

Item 88: Write readObject methods defensively

Not applicable to the project.

Item 89: For instance control, prefer enum types to readResolve

Not applicable to the project.

Item 90: Consider serialization proxies instead of serialized instances

Not applicable to the project.