Basic code review

For a basic code review, when reviewing the code you should ask yourself the following questions and if the answer to them is yes, you should be able to approve the code.

1. Can I understand the code with ease?
2. Are the coding standards/guidelines followed?
3. There is no duplicate code?
4. Can I unit test this code easily, would I be able to debug it and find what the issue is?
5. There are no functions/methods/classes that are too big?

If you feel that there are things that could be done better, leave a comment for the person asking him to fix it, or discuss why would it have to be that particular implementation, if the person can provide you with satisfactory arguments you could say yes, otherwise ask for a change.

Detailed code review

We will now look at a more advanced code review version that will consist of a few parts like code formatting, architecture, best practices, non-functional requirements, object oriented analysis and design principles.

This is a general version of a code review that should fit for most of the programming languages or day to day uses, although most companies will have their own code review guidelines that you will have to follow and abide to them.

Code formatting

When reviewing a person's code, look at how the code is formatted and check that it is easily readable, a couple of general things to look for would be:

- Use of alignments (the left margin), white spaces used properly and check that the code block starting and ending points are easy to identify
- Proper naming conventions should be used like (Pascal, CamelCase etc.)
- Imagine that all of the code should fit into a 14" laptop screen, therefore there should be no need to scroll horizontally to see the code
- There shouldn't be any commented out code, unless that code will be used in the next functionality implementation, or some other really good reason why it should be kept there

Architecture

The architecture selected for the project should be followed.

- Separate the concerns out
    - Use the multiple layers (presentation, business, data)
    - Respective files should be used (HTML, JavaScript, CSS)
- Technologies/patterns are in sync with the code
- Design patterns should be followed

Best practices

- There shouldn’t be any hard coded values, constants or configuration values should be used
- Enumeration (enum) should be used to group similar values together
- Comments should not be used for general explanations of what your code is doing
    - Use comments for any workarounds/hacks/temporary fixes you have made
    - Use TODO comments to keep track of pending tasks
- Multiple if/else blocks is a bad coding practice and shouldn’t be used
- If possible instead of writing custom features, try to use the ones that come with the framework

Non-functional requirements

Maintainability – application should not require a big amount of effort to be supported in the future. Issues or defects should be easily identifiable

- **Readability** – code should be self-explanatory and should read like similarly if you were reading an interesting book and it was easy to follow the story. The use of appropriate naming for functions, classes, variables and the rest is important if we want to achieve that story reading feeling. If you’re unable to understand the code in and it’s taking you quite a while to do it, the code should be either refactored or some supporting comments should be added
- **Testability** – the code should not be hard to test, in order to simplify things try to refactor the code into separate functions, that would allow them to be tested easier. Static functions, singleton classes are not easy to test, therefore you might want to try and avoid them
- **Debugging** – appropriate logging should support the code in order to make the code easier to debug and fix the issues that arose.
- **Configuration** – configurable values are best to be kept in a certain place (DB table, XML file) so that the code changes wouldn’t be needed to change the values.

Reusability

- The same code should not be repeated more than twice, follow the DRY principle (Do not Repeat Yourself)
- Try to code in a way where you have reusable services, functions, components.
- Generic classes and functions is also a good way to achieve reusability

Reliability – appropriate exception handling should exist and handle the exceptions

Extensibility – it should require minimal changes to the code in order to add enhancements to the code. Components should be easily replaceable by a better component.

Security – the application should be secure and things like authentication, authorization, input data should be able to deal with threats like SQL injections, cross site scripting, as well as sensitive data should be encrypted.

Performance

- Try to use the most appropriate data types, consider each types advantages/disadvantages
- Consider things like lazy loading, asynchronous and parallel processing, caching, session/application data

Scalability – try to consider things like: is the application going to have a large number of users/data. Can this application scale if deployed to something like AWS?

Usability – try to imagine that you're the end user of the application and evaluate the user interface/API, how easy they are to use/navigate. If you think that there are issues, try to discuss them with your business analyst and see if you can address them.

Object oriented design principles and analysis

- **Single Responsibility Principle (SRS)** – this principle refers to not placing more than one responsibility into a single class, method or a function, if they have more than one they should be separated out. An example of it would be, if we had a method that read a file made changes and then wrote the changes back into the file, this method now has three responsibilities, one of them is to read the file, second is to make changes, the third is to update the file
- **Open Closed Principle** – Existing code should not be modified when adding new functionality. New features should be added by creating new functions and classes
- **Liskov substitution principle** – The subclass should not change the behaviour (meaning) of the parent class.
- **Interface (trait) segregation** – Interfaces (traits) shouldn't grow too big, functionality should be split into multiple interfaces (traits) grouped on the functionality. There shouldn't be any dependencies in the interface (trait) if it's not required to achieve the desired functionality
- **Dependency injection** – dependencies should be injected, avoid hardcoding them

There will be cases where the requirements for the functionality contradict with the principles, in those cases you need to decide which outweighs the other and find a compromise for the problem.

Tools for code reviews

- The first step would be to use static code analysis tools. Depending on the language the technologies used for doing change but the main one would be SonarQube. IntelliJ has its own inbuilt analysis tool that you can trigger by going to Analyze -> Inspect code and then selecting if you want the whole project or a smaller part of it
- To keep track of the comments for changes requested on a reviewed code, most version control tools like GitHub, BitBucket, Gogs allow you to do that, the format of it more or less similar