**JDK.**

Java Standard Edition (Java SE) Development Kit (JDK). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

**JRE.**

The Java Runtime Environment (JRE) allows to run application written in the Java programming language. Like the JDK, it contains the Java Virtual Machine (JVM), classes comprising the Java platform API, and supporting files. Unlike the JDK, it does not contain development tools such as compilers and debuggers.

You can freely redistribute the JRE with your application, according to the terms of the JRE license. Once you have developed your application using the JDK, you can ship it with the JRE so your end-users will have a Java platform on which to run your software.

**DESIGN PATTERNS**

Are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can’t just find a pattern and copy it into your program, that way you can with off-the shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is more a high-level description of a solution. The code of the same pattern applied to different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Here are the sections that are usually present in a pattern description:

* **Intent** of the briefly describes both the problem and the solution.
* **Motivation** further explains the problem and the solution the pattern makes possible.
* **Structure** of classes shows each part of the pattern and how they are related
* **Code** example in one of the popular programming languages makes it easier to grasp the idea behind the pattern

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

**Why should I learn patterns?**

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern.

* Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing pattern is still useful because it teaches you how to solve all sort of problems using principles of object-oriented design.
* Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that”, and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern.

**Classification of patterns**

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.

The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

Mainly patterns are classified as:

* **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
* **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
* **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

**Creational patterns.**

* **Factory method**
* **Abstract factory**
* **Builder**
* **Prototype**
* **Singleton**

**Structural patterns.**

* **Adapter**
* **Bridge**
* **Composite**
* **Decorator**
* **Facade**
* **Flyweight**
* **Proxy**

**Behavioral patterns.**

* **Chain of Responsibility**
* **Command**
* **Memento**
* **Observer**
* **Iterator**
* **Mediator**
* **State**
* **Strategy**
* **Template Method**
* **Visitor**

**FACTORY METHOD**

**Intent:** Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

Suggests that you replace direct object construction calls with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.

This pattern allows you to override the factory method in a subclass and change the class of products being created by the method.

There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

*Applicability*

* **Use it when you don’t know beforehand the exact types and dependencies of the objects your code should work with**.

Separates product construction code from the code that uses the product. Therefore, it’s easier to extend the product construction code independently from the rest of the code.

* **Use it when you want to provide users of your library or framework with a way to extend its internal components.**

Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component itself.

* **Use it when you want to save system resources by reusing existing objects instead of rebuilding them each time.**

You often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.