1. **Factory Method** is a creational design pattern that provides a methodology for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
2. **In simple terms** - it’s a way to create different types of objects without exposing the creation logic to the client code. The client just asks for what it needs, and the factory gives back the correct object based on the requirement.
3. Objects returned by a factory method are often referred to as ***products****.*
4. 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.
5. In the **Factory Pattern**, the term **"Creator"** refers to the **class that contains the factory method** — i.e., the method responsible for **creating and returning objects/ different factories**.
6. Different terminologies in Factory Pattern:  
   **a. Real Client Code**: Who actually use the factory to get the required product.  
    For example:  
   In layman language - actual customer who ordered chai in a restaurant through waiter.   
   In Programming World - Main(), a controller, or a test is the real client who uses the Factory Client.  
     
   **b. Factory Client Code (Simulator):** It is the one who uses the factory to get the real product which should be consumed by the real client in above.  
   For example:  
   In layman language – Suppose if actual customer order the adrak chai to waiter in above example, then waiter(simulator) will go to the adrak factory to get the adrak chai for the actual consumer. Similarly if he ordered Eliachi chai then he should go the Eliachi factory.  
   In Programming World: Treat different factory classes the same way through a common interface or abstract base, and let runtime binding decide which factory is used. Because we want out actual **client code** to be **decoupled from knowing which factory is being used**, and **delegate the choice to simulator.**It will go to the **Creator Class** and **Concrete Creator** Classes for doing the same.  
     
   **A screenshot of a computer program

   AI-generated content may be incorrect.**  
     
   **A screenshot of a computer

   AI-generated content may be incorrect.  
     
   c.** **Product:** It’s the common type of object that the factory method will return. Think of this as a **contract**. All subclasses need to follow this.  
   For example:  
   In Layman language: Chai is the actual product which is order by the customer in the step a and all other adrak or eliachi chaiwalas (**Concrete Product**) need to give the chai to the end user although the way they make the chai would be different but the end product should be the chai only means they need to follow the same contract that is chai.  
     
   **A white rectangular object with a black border

   AI-generated content may be incorrect.**  
     
   **A white rectangular object with a black border

   AI-generated content may be incorrect.**  
    **d. Concrete Products:** It’s the **object type that the factory creates and returns and** the actual implementation of the product interface, as each chaiwala has its own implementation but all follow the IChai contract in it.  
     
   **A computer code on a white background

   AI-generated content may be incorrect.**  
     
     
   **A screenshot of a computer program

   AI-generated content may be incorrect.**  
     
   **e. Creator**: The **Creator** class declares the factory method that returns new product objects. It’s important that the return type of this method matches the product interface. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes.  
     
   A screenshot of a computer

   AI-generated content may be incorrect.  
     
   **f. Concrete Creators**: **Concrete Creators** override the base factory method so it returns a different type of product. It doesn’t have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.
7. **Simple Factory (centralized logic using switch):** It has only one factory class, **Easier** to maintain for small apps, **Violates Open/Closed Principle**: if you add a new vehicle, you must modify this factory. We can think this in a way that One chaiwala who asks what you want, then makes that chai from a recipe book.  
     
   **A computer screen shot of a computer code

   AI-generated content may be incorrect.**
8. **Factory Method Pattern (Concrete Creators):**Each concrete class knows how to instantiate its product with its specifications, Adheres better to **SOLID** principles that are SRP and OCP, Easy to extend: just add a new factory class. It’s good to use when object creation logic differs significantly. We can think this in a way that - You go to a different chaiwala for adrak chai and elaichi chai — each one has their own secret masala and method.  
     
   **A computer screen shot of a program

   AI-generated content may be incorrect.**
9. **Correct Conceptual Flow:  
     
   A screenshot of a computer program

   AI-generated content may be incorrect.  
     
   A screenshot of a computer

   AI-generated content may be incorrect.**
10. **Final Correct Flow (Vertical Visual)  
      
    A screenshot of a computer

    AI-generated content may be incorrect.**