**Design Patterns**

A design patterns are **well-proved solution** for solving the specific problem/task.

Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.

**Problem Given:**  
Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

**Solution:**  
**Singleton design pattern** is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

## Advantage of design pattern:

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don?t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

### When should we use the design patterns?

We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Creational design patterns

Creational design patterns are concerned with**the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

But everyone knows an object is created by using new keyword in java. For example:

**StudentRecord s1=new StudentRecord();**

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

Structural design patterns

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

1. Adapter Pattern
2. Composite Pattern
3. Proxy Pattern
4. Flyweight Pattern
5. Facade Pattern
6. Bridge Pattern
7. Decorator Pattern

Behavioral Design Patterns

Behavioral design patterns are concerned with **the interaction and responsibility of objects.**

In these design patterns, **the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

1. Template Method Pattern
2. Mediator Pattern
3. Chain of Responsibility Pattern
4. Observer Pattern
5. Strategy Pattern
6. Command Pattern
7. State Pattern
8. Visitor Pattern
9. Iterator Pattern
10. Memento Pattern

**Singleton Pattern**

A singleton is a class that is instantiated only once. This is typically accomplished by creating a static field in the class representing the class. A static method exists on the class to obtain the instance of the class and is typically named something such as getInstance(). The creation of the object referenced by the static field can be done either when the class is initialized or the first time that getInstance() is called. The singleton class typically has a private constructor to prevent the singleton class from being instantiated via a constructor. Rather, the instance of the singleton is obtained via the static getInstance() method.

### SingletonExample.java

**package** com.cakes;

**public** **class** SingletonExample {

**private** **static** SingletonExample singletonExample = null;

**private** SingletonExample() {

}

**public** **static** SingletonExample getInstance() {

**if** (singletonExample == null) {

singletonExample = **new** SingletonExample();

}

**return** singletonExample;

}

**public** **void** sayHello() {

System.out.println(**"Hello"**);

}

}

The Demo class obtains a SingletonExample singleton class via the call to the static SingletonExample.getInstance(). We call the sayHello() method on the singleton class. Executing the Demo class outputs "Hello" to standard output.

### Demo.java

**package** com.cakes;

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

SingletonExample singletonExample = SingletonExample.getInstance();

singletonExample.sayHello();

}

}

Singleton classes are a useful way of concentrating access to particular resources into a single class instance.

**Factory Pattern**

The factory pattern (also known as the factory method pattern) is a creational design pattern. A factory is a Java[S](http://java.sun.com/)W class that is used to encapsulate object creation code. A factory class instantiates and returns a particular type of object based on data passed to the factory. The different types of objects that are returned from a factory typically are subclasses of a common parent class.

The data passed from the calling code to the factory can be passed either when the factory is created or when the method on the factory is called to create an object. This creational method is often called something such as *getInstance*or *getClass*

### [Animal.java](https://www.avajava.com/tutorials/design-patterns/factory-pattern/Animal.java)

**package** com.cakes;

**public** **abstract** **class** Animal {

**public** **abstract** String makeSound();

}

The Dog class is a subclass of Animal. It implements makeSound() to return "Woof".

### [Dog.java](https://www.avajava.com/tutorials/design-patterns/factory-pattern/Dog.java)

**package** com.cakes;

**public** **class** Dog **extends** Animal {

@Override

**public** String makeSound() {

**return** **"Woof"**;

}

}

The Cat class is a subclass of Animal. It implements makeSound() to return "Meow".

### [Cat.java](https://www.avajava.com/tutorials/design-patterns/factory-pattern/Cat.java)

**package** com.cakes;

**public** **class** Cat **extends** Animal {

@Override

**public** String makeSound() {

**return** **"Meow"**;

}

}

Now, let's implement our factory. We will call our factory's object creation method getAnimal. This method takes a String as a parameter. If the String is "canine", it returns a Dog object. Otherwise, it returns a Cat object.

### [AnimalFactory.java](https://www.avajava.com/tutorials/design-patterns/factory-pattern/AnimalFactory.java)

**package** com.cakes;

**public** **class** AnimalFactory {

**public** Animal getAnimal(String type) {

**if** (**"canine"**.equals(type)) {

**return** **new** Dog();

} **else** {

**return** **new** Cat();

}

}

}

The Demo class demonstrates the use of our factory. It creates an AnimalFactory factory. The factory creates an Animal object and then another Animal object. The first object is a Cat and the second object is a Dog. The output of each object's makeSound() method is displayed.

### [Demo.java](https://www.avajava.com/tutorials/design-patterns/factory-pattern/Demo.java)

**package** com.cakes;

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

AnimalFactory animalFactory = **new** AnimalFactory();

Animal a1 = animalFactory.getAnimal(**"feline"**);

System.out.println(**"a1 sound: "** + a1.makeSound());

Animal a2 = animalFactory.getAnimal(**"canine"**);

System.out.println(**"a2 sound: "** + a2.makeSound());

}

}

The console output is shown here.

### [Console Output](https://www.avajava.com/tutorials/design-patterns/factory-pattern/console.txt)

a1 sound: Meow

a2 sound: Woof