|  |  |  |
| --- | --- | --- |
| **1** | Design Patterns in Java | |
| This Cheat Sheet provides a quick reference to the original 23 **Gang of Four** (**GoF**) design patterns, as listed in the book Design Patterns: Elements of Reusable Object-Oriented Software. Each pattern includes class diagrams, explanation, usage, information, and a real world example. | | |
| |  |  | | --- | --- | |  | Creational Patterns Used to construct object such that they can be decoupled from their implementing system | |  | Structural Patterns Used to form large object structures between many disparate objects. | |  | Behavioral Patterns Used to manage algorithms, relationships, and responsibilities between objects. |  |  |  | | --- | --- | | Object Scope | Deals with object relationships that can be changed at runtime. | | Class Scope | Deals with class relationships that can be changed at compile time. | | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Abstract Factory |  | Decorator |  | Prototype | |  | Adapter |  | Façade |  | Proxy | |  | Bridge |  | Factory Method |  | Observer | |  | Builder |  | Flyweight |  | Singleton | |  | Chain of Responsibility |  | Iterator |  | Strategy | |  | Template Method |  | Mediator |  | Command | |  | Composite |  | Memento |  | Visitor | |
| |  |  |  |  | | --- | --- | --- | --- | | ABSTRACT FACTORY | | | Object Creational | | AZERTY | | | | | WHAT ? | azerty | | | | WHEN ? | azerty | | | | EXAMPLE | | azerty | |  |  |  |  |  | | --- | --- | --- | --- | | ABSTRACT FACTORY | | | Object Behavioral | | AZERTY | | | | | WHAT ? | azerty | | | | WHEN ? | azerty | | | | EXAMPLE | | azerty | | | | |  |  |  |  | | --- | --- | --- | --- | | ABSTRACT FACTORY | | | Object Structural | | AZERTY | | | | | WHAT ? | azerty | | | | WHEN ? | azerty | | | | EXAMPLE | | azerty | | |
| Creational Patterns | | |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | SINGLETON | | **Object Creational** | |  | | | | | | WHAT ? | | | Only one instance of a class is allowed within a system. | | | WHEN ? | | | * Exactly one instance of a class is required. * Controlled access to single object is necessary. * There are two forms of Singleton Instantiation:   + **Early** : object creation takes place at the load time.   + **Lazy** : object creation done according to requirement   + **Static Block Initialization**: class loading sequence.   + **Bill Pugh solution**: for java memory model changes.   + **Enum**: implementation employs the use of Enum.   + **readResolve()**: to avoid multiple instance problem. | | | EXAMPLE | | | We can use this single object repeatedly as per the requirements. This is the reason why the multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings, and much more. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it. | | | RESUME | | | Some argue that it violates the single responsibility principle. Singletons can also make testing more difficult. Some consider it to be a non-desirable coding practice since they introduce a global state to the program. However, Singletons are useful for thread-safety, serialization or to faster performance, this pattern can improve the overall readability and elegance of your code. This design pattern should not be avoided since the it is, after all, a classic Gang of Four design pattern. | | | | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | PROTOTYPE | | **Object Creational** | |  | | | | | | WHAT ? | | | Exposes a method for creating objects & allowing subclasses to control the actual creation process. | | | WHEN ? | | | * A class will not know what classes it will be required to create. * Subclasses may specify what objects should be created. * Parent classes wish to defer creation to their subclasses. * Participants are :   + **Prototype**: This is the prototype of actual object as discussed above.   + **Prototype registry**: This is used as registry service to have all prototypes accessible using simple string parameters.   + **Client**: will be responsible for using registry service to access porotype instances. * For objects cloning there is:   + **Shallow Copy** : we use it when we have immutable objects (whose state cannot change, once created).   + **Deep Copy** : when state changes (mutable objects). | | | Benefits  &  Inconvenient | | | * Create the instances with the appropriate state beforehand and then clone them whenever we want. * Eliminate subclasses that only differ in their state by creating prototypes with the initial state and then cloning them. * Process get complex when there are many classes. * Difficult to clone classes that have circular references. | | |
| Creational Patterns | | |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | SINGLETON | | **Object Creational** | |  | | | | | | WHAT ? | | | Only one instance of a class is allowed within a system. | | | WHEN ? | | | * Exactly one instance of a class is required. * Controlled access to single object is necessary. * There are two forms of Singleton Instantiation:   + **Early** : object creation takes place at the load time.   + **Lazy** : object creation done according to requirement   + **Static Block Initialization**: class loading sequence.   + **Bill Pugh solution**: for java memory model changes.   + **Enum**: implementation employs the use of Enum.   + **readResolve()**: to avoid multiple instance problem. | | | EXAMPLE | | | We can use this single object repeatedly as per the requirements. This is the reason why the multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings, and much more. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it. | | | RESUME | | | Some argue that it violates the single responsibility principle. Singletons can also make testing more difficult. Some consider it to be a non-desirable coding practice since they introduce a global state to the program. However, Singletons are useful for thread-safety, serialization or to faster performance, this pattern can improve the overall readability and elegance of your code. This design pattern should not be avoided since the it is, after all, a classic Gang of Four design pattern. | | | | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | PROTOTYPE | | **Object Creational** | |  | | | | | | WHAT ? | | | Exposes a method for creating objects & allowing subclasses to control the actual creation process. | | | WHEN ? | | | * A class will not know what classes it will be required to create. * Subclasses may specify what objects should be created. * Parent classes wish to defer creation to their subclasses. * Participants are :   + **Prototype**: This is the prototype of actual object as discussed above.   + **Prototype registry**: This is used as registry service to have all prototypes accessible using simple string parameters.   + **Client**: will be responsible for using registry service to access porotype instances. * For objects cloning there is:   + **Shallow Copy** : we use it when we have immutable objects (whose state cannot change, once created).   + **Deep Copy** : when state changes (mutable objects). | | | Benefits  &  Inconvenient | | | * Create the instances with the appropriate state beforehand and then clone them whenever we want. * Eliminate subclasses that only differ in their state by creating prototypes with the initial state and then cloning them. * Process get complex when there are many classes. * Difficult to clone classes that have circular references. | | |