**Lazy Loading Pattern**

**Problem Definition**

  We want to delay the loading of object until the point where we need it.

**Pattern Idea**

* Lazy loading is just a fancy name given to the process of initializing a class when it’s actually needed.
* In simple words, Lazy loading is a software design pattern where the initialization of an object occurs only when it is actually needed and not before to preserve simplicity of usage and improve performance.
* Lazy loading is essential when the cost of object creation is very high and the use of the object is very rare. So this is the scenario where it’s worth implementing lazy loading.The fundamental idea of lazy loading is to load object/data when needed.

**For Example**, Suppose You are creating an application in which there is a Company object and this object contains a list of employees of the company in a ContactList object. There could be thousands of employees in a company. Loading the Company object from the database along with the list of all its employees in the ContactList object could be very time consuming. In some cases you don’t even require the list of the employees, but you are forced to wait until the company and its list of employees loaded into the memory.  
One way to save time and memory is to avoid loading of the employee objects until required and this is done using the **Lazy Loading Design Pattern**.

**There are four common implementations of Lazy Loading pattern :**

1. Virtual proxy
2. Lazy initialization
3. Ghost
4. Value holder

**Virtual proxy**

The Virtual Proxy pattern is a memory saving technique that recommends postponing an object creation until it is needed. It is used when creating an object the is expensive in terms of memory usage or processing involved.

implementation

import java.util.List;

import java.util.ArrayList;

interface ContactList

{

public List<Employee> getEmployeeList();

}

class Company {

String companyName;

String companyAddress;

String companyContactNo;

ContactList contactList;

public Company(String companyName, String companyAddress,

String companyContactNo, ContactList contactList)

{

this.companyName = companyName;

this.companyAddress = companyAddress;

this.companyContactNo = companyContactNo;

this.contactList = contactList;

}

public String getCompanyName()

{

return companyName;

}

public String getCompanyAddress()

{

return companyAddress;

}

public String getCompanyContactNo()

{

return companyContactNo;

}

public ContactList getContactList()

{

return contactList;

}

}

class ContactListImpl implements ContactList {

public List<Employee> getEmployeeList()

{

return getEmpList();

}

private static List<Employee> getEmpList()

{

List<Employee> empList = new ArrayList<Employee>(5);

empList.add(new Employee("Lokesh", 2565.55, "SE"));

empList.add(new Employee("Kushagra", 22574, "Manager"));

empList.add(new Employee("Susmit", 3256.77, "G4"));

empList.add(new Employee("Vikram", 4875.54, "SSE"));

empList.add(new Employee("Achint", 2847.01, "SE"));

return empList;

}

}

class ContactListProxyImpl implements ContactList {

private ContactList contactList;

public List<Employee> getEmployeeList()

{

if (contactList == null) {

System.out.println("Fetching list of employees");

contactList = new ContactListImpl();

}

return contactList.getEmployeeList();

}

}

class Employee {

private String employeeName;

private double employeeSalary;

private String employeeDesignation;

public Employee(String employeeName,

double employeeSalary, String employeeDesignation)

{

this.employeeName = employeeName;

this.employeeSalary = employeeSalary;

this.employeeDesignation = employeeDesignation;

}

public String getEmployeeName()

{

return employeeName;

}

public double getEmployeeSalary()

{

return employeeSalary;

}

public String getEmployeeDesignation()

{

return employeeDesignation;

}

public String toString()

{

return "Employee Name: " + employeeName + ",

EmployeeDesignation : " + employeeDesignation + ",

Employee Salary : " + employeeSalary;

}

}

class LazyLoading {

public static void main(String[] args)

{

ContactList contactList = new ContactListProxyImpl();

Company company = new Company

("Geeksforgeeks", "India", "+91-011-28458965", contactList);

System.out.println("Company Name: " + company.getCompanyName());

System.out.println("Company Address: " + company.getCompanyAddress());

System.out.println("Company Contact No.: " + company.getCompanyContactNo());

System.out.println("Requesting for contact list");

contactList = company.getContactList();

List<Employee> empList = contactList.getEmployeeList();

for (Employee emp : empList) {

System.out.println(emp);

}

}

}

**Lazy Initialization**

The Lazy Initialization technique consists of checking the value of a class field when it’s being used. If that value equals to null then that field gets loaded with the proper value before it is returned.

implementation

// Java program to illustrate

// Lazy Initialization in

// Lazy Loading Design Pattern

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

enum CarType {

none,

Audi,

BMW,

}

class Car {

private static Map<CarType, Car> types = new HashMap<>();

private Car(CarType type) {}

public static Car getCarByTypeName(CarType type)

{

Car Car;

if (!types.containsKey(type)) {

// Lazy initialisation

Car = new Car(type);

types.put(type, Car);

} else {

// It's available currently

Car = types.get(type);

}

return Car;

}

public static Car getCarByTypeNameHighConcurrentVersion(CarType type)

{

if (!types.containsKey(type)) {

synchronized(types)

{

// Check again, after having acquired the lock to make sure

// the instance was not created meanwhile by another thread

if (!types.containsKey(type)) {

// Lazy initialisation

types.put(type, new Car(type));

}

}

}

return types.get(type);

}

public static void showAll()

{

if (types.size() > 0) {

System.out.println("Number of instances made = " + types.size());

for (Entry<CarType, Car> entry : types.entrySet()) {

String Car = entry.getKey().toString();

Car = Character.toUpperCase(Car.charAt(0)) + Car.substring(1);

System.out.println(Car);

}

System.out.println();

}

}

}

class Program {

public static void main(String[] args)

{

Car.getCarByTypeName(CarType.BMW);

Car.showAll();

Car.getCarByTypeName(CarType.Audi);

Car.showAll();

Car.getCarByTypeName(CarType.BMW);

Car.showAll();

}

}

**Value Holder**

Basically, A value holder is a generic object that handles the lazy loading behavior and appears in place of the object’s data fields.When the user needs to access it, they simply ask the value holder for its value by calling the GetValue method. At that time (and only then), the value gets loaded from a database or from a service.(this is not always needed).

Implementation

// Java function to illustrate

// Lazy Initialization in

// Lazy Loading Design Pattern

public class ValueHolder<T> {

private T value;

private readonly Func<object, T> valueRetrieval;

// Constructor

public ValueHolder(Func<object, T> valueRetrieval)

{

valueRetrieval = this.valueRetrieval;

}

// We'll use the signature "GetValue" for convention

public T GetValue(object parameter)

{

if (value == null)

value = valueRetrieval(parameter);

return value;

}

}

**Ghost**

A ghost is the object that is to be loaded in a partial state. It corresponds to the real object but not in its full state. It may be empty or it may contain just some fields (such as the ID). When the user tries to access some fields that haven’t been loaded yet, the ghost object fully initializes itself (this is not always needed).

**Drawbacks**

The code becomes complicated as we need to check if loading is needed or not. So this may cause slower in the performance.