Power Programming with Java

Session: 10

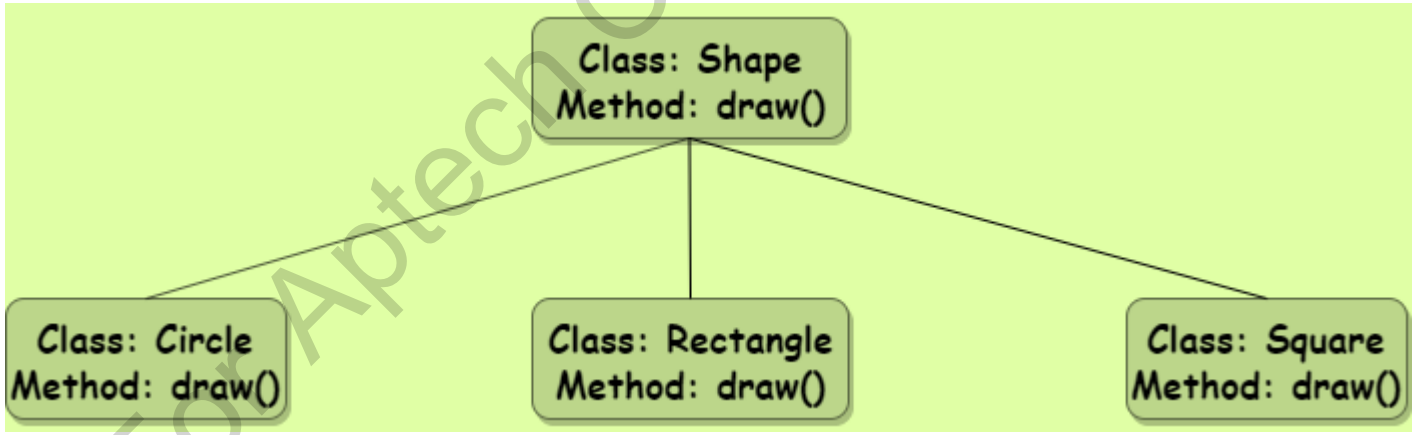
Design Patterns and Other Advanced Features

Objectives

- Explain design patterns
- Describe internationalization and localization
- Describe atomic operations with the new set of classes of the `java.util.concurrent.atomic` package
- Explain the new features of `ForkJoinPool`
- Describe parallel sorting of arrays
- Describe delegation, composition, and aggregation
- Explain the internationalization process
- Explain the enhancements of `java.util.concurrency` package
- Explain the `StampedLock` class to implement locks
- Define parallel streams
- Identify recursive actions of the fork/join framework

Introduction

- In the biological world, when certain organisms exhibit many different forms, it is termed as polymorphism.
- A similar concept can be applied to Java.
- Here, a subclass can have its own unique behavior.
- This is possible even while sharing some common functionalities with the parent class.



Design Patterns

- A design pattern is a clearly defined solution to problems that occur frequently.
- It can be considered as a template or a best practice suggested by expert programmers.
- It is great for learning good details about software design.
- Its proper usage results in enhanced code maintainability.
- A design pattern is neither an implementation not a framework.
- It cannot be installed using code.

Types of Patterns:

Creational Patterns

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

Structural Patterns

- Adapter Pattern
- Composite Pattern
- Proxy Pattern
- Flyweight Pattern
- Facade Pattern
- Bridge Pattern
- Decorator Pattern

Behavioral Patterns

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

Singleton Pattern [1-2]

- Certain class implementations can be instantiated only once.
- Singleton design pattern provides complete information on such class implementations.
- A static field is created representing the class to implement this.
- The object that the static field references can be created at the time when the class is initialized or the first time the `getInstance()` method is invoked.
- The constructor of a class using the singleton pattern is declared as private.
- To execute a singleton design pattern, perform following steps:

Step 1: Use a static reference to point to the single instance.

Step 2: Then, add a single private constructor to the singleton class.

Step 3: Next, a public factory method is declared static to access the static field declared in Step 1. Similar to the concept of a factory that manufactures products, the job of the factory method is to manufacture objects.

Step 4: Use the static `getInstance()` method to get the instance of the singleton.

Singleton Pattern [2-2]

Code Snippet

```
class SingletonExample {
  private static SingletonExample = null;
  private SingletonExample() {}
  public static SingletonExample getInstance() {
  if (singletonExample == null) {
     singletonExample = new SingletonExample();
  }
  return singletonExample;
  }
  public void display() {
    System.out.println("Welcome to Singleton Design Pattern");
  }
}
```

Code Snippet

```
public class SingletonTest {
   /**
   * @param args the command line arguments
   */
   public static void main(String[] args) {
   SingletonExample singletonExample = SingletonExample.getInstance();
   singletonExample.display();
   }
}
```

Interfaces in Design Patterns

- Java uses interfaces to define type abstraction.
- Outlining abstract types is a powerful feature of Java.

Benefits of Abstraction:

- Vendor-specific Implementation
- Tandem Development
- Easy Maintenance

Code Snippet shows an interface declaration:

Code Snippet

```
public interface IAircraft {
public int passengerCapacity = 400;
// method signatures void fly();
......
// more method signatures
}
```

- Only constant fields are allowed in an interface.
- A field is implicitly `public`, `static`, and `final` when an interface is declared.
- It is recommended to distribute constant values of an application across many classes and interface.

Difference between Class Inheritance and Interface Inheritance

A class can extend a single parent class whereas, it can implement multiple interfaces.

When a class extends a parent class, only certain functionalities can be overridden in the inherited class.

When a class inherits an interface, it implements all functionalities of the interfaces.

A class is extended because certain classes require detailed implementation based on the superclass.

Following table provides a comparison between an abstract class and an interface:

Interface	Abstract Class
Inheritance of several interfaces by a class is supported.	Inheritance of only one abstract class by a class is supported.
Requires more time to find the actual method in the corresponding child classes.	Requires less time to find the actual method in the corresponding child classes.
Best used when various implementations only share method signatures.	Best used when various implementations use common behavior or status.

Extending Interfaces

- It is recommended to specify all uses of the interface right from the beginning.
- This is not possible always. In such an event, more interfaces can be created later.
- Interfaces can be used to extend interfaces.
- Code Snippet creates an interface called `IVehicle`:

Code Snippet

```
public interface IVehicle {
int getMileage(String s);
. . .
}
```

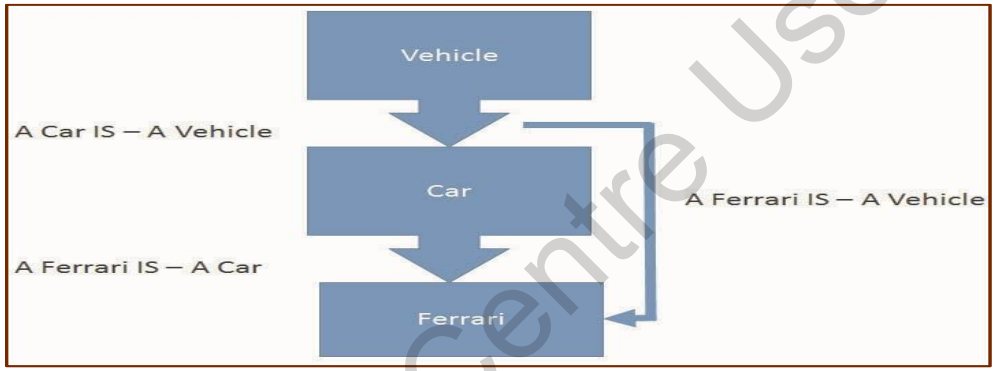
- Code Snippet shows how a new interface can be created that extends `IVehicle`:

Code Snippet

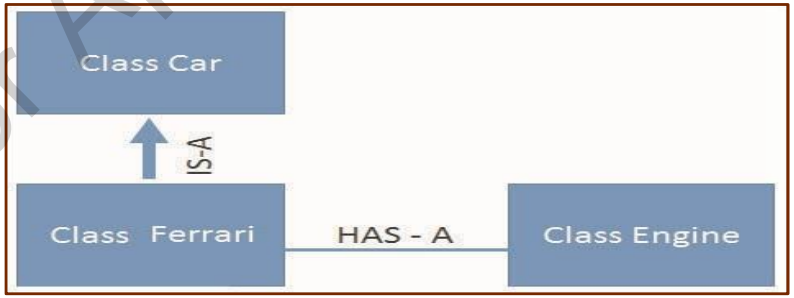
```
public interface IAutomobile extends IVehicle {
boolean accelerate(int i, double x, String s);
}
```

Implementation of IS-A and a HAS-A Relationships

- It is a concept based on class inheritance or interface implementation.
- An IS-A relationship displays class hierarchy in case of class inheritance.
- For example, if the class Ferrari extends the class ***Car***, the statement Ferrari IS-A ***Car*** is true.
- Following figure illustrates an IS-A relationship:



- The IS-A relationship is also used for interface implementation.
- This is done using keyword implements or extends.
- An HAS-A relationship uses instance variables that are references to other objects, such as a Ferrari includes an Engine.
- Following figure illustrates a HAS-A relationship:



The Data Access Object (DAO) Design Pattern [1-5]

The DAO pattern is used when an application is created that must persist its data.

It involves a technique for separating the business logic from persistence logic.

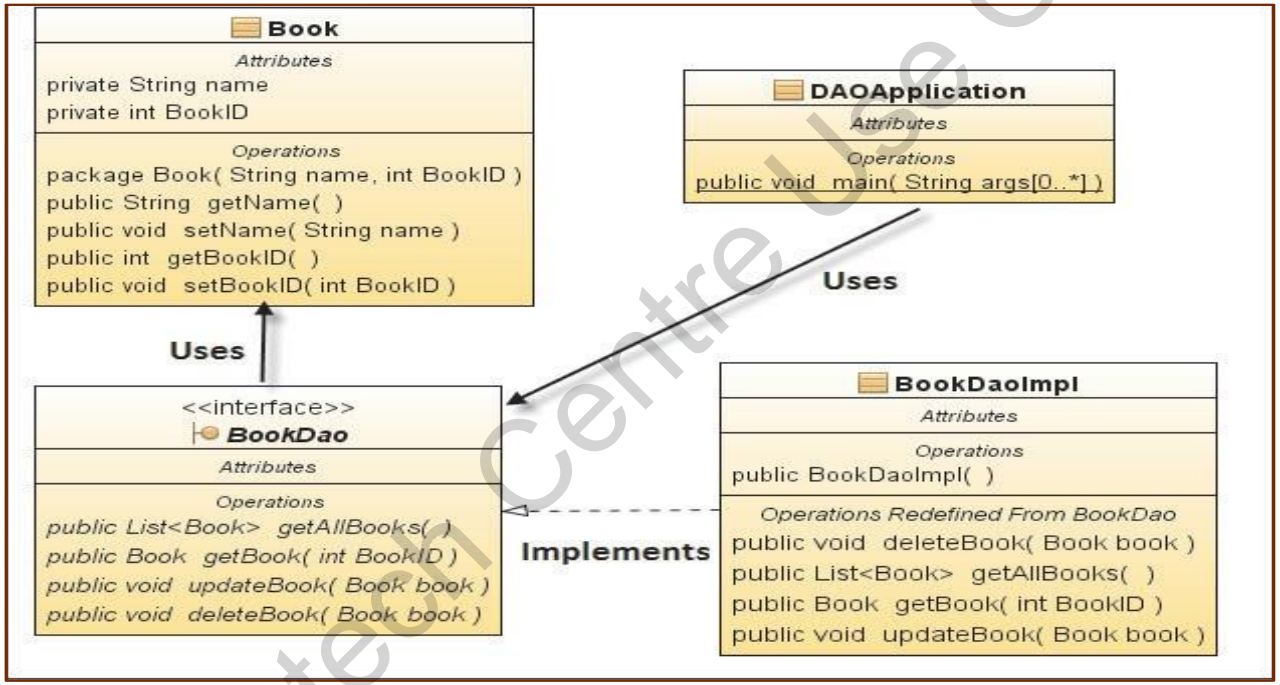
This makes it flexible to change the persistence mechanism of the application without changing the entire application.

DAO pattern uses:

- DAO Interface
- DAO Concrete Class
- Model Object or Value Object
- Memory based DAOs
- File based DAOs
- JDBC based DAOs
- Java persistence API based DAOs

The Data Access Object (DAO) Design Pattern [2-5]

- Following figure shows structure of a DAO design pattern that will be created:



- First, the Model or Value object is created which will store the book information.

The Data Access Object (DAO) Design Pattern [3-5]

Code Snippet illustrates this by defining a `Book` class having a constructor and get/see methods.

Code Snippet

```
class Book {
private String name;
private int BookID;
Book(String name, int BookID) {
this.name = name;
this.BookID = BookID;
}
public String getName() { return name; }
public void setName(String name) { this.name = name;}
public int getBookID() { return BookID;}
public void setBookID(int BookID) { this.BookID = BookID;}
}
```

Code Snippet

- Code Snippet creates a DAO interface that makes use of the `Book` class:

```
interface BookDao {
public java.util.List<Book> getAllBooks();
public Book getBook(int BookID);
public void updateBook(Book book);
public void deleteBook(Book book);
}
```

The Data Access Object (DAO) Design Pattern [4-5]

- Code Snippet creates a class that implements the interface:

Code Snippet

```
class BookDaoImpl implements BookDao {
// Here, list is working as a database
java.util.List<Book> booksList;
public BookDaoImpl(){
booksList = new java.util.ArrayList<Book>();
Book bookObj1 = new Book("Anna",1);
Book bookObj2 = new Book("John",2);
booksList.add(bookObj1); booksList.add(bookObj2);
}
@Override
public void deleteBook(Book book) { booksList.remove(book.getBookID());
System.out.println("Book: Book ID " + book.getBookID()
+", deleted from database");
}
// retrieve list of booksList from the database
@Override
public java.util.List<Book> getAllBooks() { return booksList;
}
@Override
public Book getBook(int BookID) { return booksList.get(BookID);}
@Override
public void updateBook(Book book) { booksList.get(book.getBookID()).setName(book.getName());
System.out.println("Book: Book ID " + book.getBookID() +", updated in the database");
}
}
```

The Data Access Object (DAO) Design Pattern [5-5]

- Code Snippet displays the use of the DAO pattern:

Code Snippet

```
public class DAOPatternApplication {
public static void main(String[] args) {
BookDao bookDao = new BookDaoImpl();
System.out.println("Book List:");
//print all booksList
for (Book book : bookDao.getAllBooks()) {
System.out.println("\n BookID : " + book.getBookID() + ", Name :" + book.getName() + " ");
}
//update book
Book book = bookDao.getAllBooks().get(0);
book.setName("Harry Long");
bookDao.updateBook(book);
//get the book
bookDao.getBook(0);
System.out.println("Book: [BookID : " + book.getBookID() + ", Name:" + book.getName() + "
]");
}
}
```

- Following figure displays the output of DAO design pattern:

```
Book List:

BookID : 1, Name : Anna

BookID : 2, Name : John
Book: Book ID 1, updated in the database
Book: [BookID : 1, Name : Harry Long ]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Factory Design Pattern [1-3]

- Factory pattern is one of the commonly used design patterns in Java.
- It belongs to the creational pattern category.
- It provides many ways to create an object.
- It prevents the application from being tightly coupled to a specific DAO implementation.
- This pattern does not perform direct constructor calls when invoking a method.
- Code Snippet creates a common interface for implementing the factory pattern:

Code Snippet

```
interface Vehicle {
void move();
}
```

- Code Snippet creates a class that implements the interface:

Code Snippet

```
class Car implements Vehicle {
@Override
public void move() {
System.out.println("Inside Car::move() method.");
}
}
```

Factory Design Pattern [2-3]

- Code Snippet creates another class that implements the interface:

Code Snippet

```
class Truck implements Vehicle {
@Override
public void move() {
    System.out.println("Inside Truck::move() method.");
   }
}
```

- Code Snippet creates a factory that creates object of concrete class based on information provided:

Code Snippet

```
class VehicleFactory {
   //use getVehicle method to get object of type Vehicle
   public Vehicle getVehicle(String vehicleType){
   if(vehicleType == null){
         return null;
       }
    if(vehicleType.equalsIgnoreCase("Car")){
         return new Car();
    } else if(vehicleType.equalsIgnoreCase("Truck")){
        return new Truck();
    }
    return null;
   }
}
```

Factory Design Pattern [3-3]

- Code Snippet uses the factory pattern to get objects of concrete classes by passing the `Vehicle` type information:

Code Snippet

```
public class FactoryPatternExample {
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
VehicleFactory vehicleFactory = new VehicleFactory();
//get an object of Car and call its move method.
Vehicle carObj = vehicleFactory.getVehicle("Car");
//call move method of Car carObj.move();
//get an object of Truck and call its move method.
Vehicle truckObj = vehicleFactory.getVehicle("Truck");
//call move method of truck truckObj.move();
}
}
```

Delegation

- Besides the standard design patterns, users can also use the delegation design pattern.
- In Java, delegation means using an object of another class as an instance variable.
- It also involves forwarding messages to the instance.
- Delegation is a relationship between objects.
- Delegation is different from inheritance.
- Delegation supports code reusability and provides runtime flexibility.
- Code Snippet displays the use of delegation using a real-world scenario:

Code Snippet

```
interface Employee {
public Result sendMail();
}
public class Secretary implements Employee {
public Result sendMail() {
Result myResult = new Result(); return myResult;
}
}
public class Manager implements Employee {
private Secretary ;
public Result sendMail() {
return secretary.sendMail();
}
}
```

Composition and Aggregation [1-2]

- Composition is the process of composing a class from references to other objects.
- References to the main objects are fields of the containing object.
- Object composition can be used to create more complex objects.
- In aggregation, one class owns another class.
- In composition, when the owning object is destroyed, so are the objects within it
- Composition and aggregation are design concepts and not actual patterns.
- Code Snippet demonstrates a minimal example of composition.

Code Snippet

```
// Composition class House
{
// House has door.
// Door is built when House is built,
// it is destroyed when House is destroyed.
private Door dr;
};
```

Composition and Aggregation [2-2]

- Considering an example of a student attending a course, the composition for the `Course` and `Student` classes is depicted in the Code Snippets.

Code Snippet

```
package composition;
public class Course {
private String title;
private long score;
private int id;
public String getTitle() {
return title;
}
public void setTitle(String title) {
this.title = title;
}
public long getScore() { return score;
}
public void setScore(long score) {
this.score = score;
}
public int getId() { return id;
}
public void setId(int id) { this.id =
id;
}
}
```

The Course class is then used in the Student class in the code given in the right block.

Code Snippet

```
package composition;
public class Student {
//composition has-a relationship
private Course course;
public Student(){
this.course=new Course();
course.setScore(1000);
}
public long getScore() {
return course.getScore();
}
/**
* @param args the command line
arguments
*/
public static void main(String[] args)
{
Student p = new Student();
System.out.println(p.getScore());
}
}
```

Internationalization and Localization

Internationalization:

- Internationalization helps make an application accessible to the international market.
- It ensures that the input and output operations are specific to different locations and user preferences.
- The process occurs without any engineering changes.
- Java includes built-in support to internationalize applications
- This is called Java internationalization.

Localization:

- Localization deals with a specific region or language.
- In localization, an application is adapted to a specific region or language.
- Locale-specific components are added.
- Text is translated.
- The user interface elements and documentation are translated.
- Changes related to dates, currency, and so on are also taken care of.
- Culturally sensitive data, such as images, are localized as well.

Internationalization Process [1-4]

- For internationalization process, steps to be followed are:

Creating the Properties Files:

- A properties file stores information about the characteristics of a program or environment.
- A properties file is in plain-text format.
- It can be created with any text editor.

Defining the Locale:

- `Locale` object identifies a particular language and country.
- A `Locale` is an identifier for a particular combination of language and region.
- A `java.util.Locale` class object represents a specific geographical, political, or cultural region.
- Any operation that requires a locale to perform its task is said to be locale-sensitive.
- These operations use the `Locale` object to tailor information for the user.
- A `Locale` object is created using following constructors:
  - `public language, String country)`
  - `public Locale(StLocale(String ring language)`

Internationalization Process [2-4]

- Code Snippet creates `Locale` objects for the French language for the countries Canada and France.

Code Snippet

```
caLocale = new Locale("fr","CA");
frLocale = new Locale("fr","FR");
```

- Code Snippet demonstrates how to accept the language and country code from command line.

Code Snippet

```
String language = new String(args[0]);
String country = new String(args[1]);
currentLocale = new Locale(language, country);
```

- Following are certain important methods of the `Locale` class:
  - `public static Locale getDefault()`
  - `public final String getDisplayCountry()`
  - `public final String getDisplayLanguage()`

Internationalization Process [3-4]

Creating a `ResourceBundle`:

- `ResourceBundle` objects contain locale-specific objects used to isolate locale-sensitive data, such as translatable text.
- The `ResourceBundle` class is used to retrieve locale-specific information from the properties file.
- This class has a static and final method `getBundle()` that helps to retrieve a `ResourceBundle` instance.
- `ResourceBundle getBundle(String, Locale)` method helps to retrieve locale-specific information from a given properties file.
- For creating the `ResourceBundle`, consider following statement:

```
messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
```

- To retrieve the locale-specific data from the properties file, the `ResourceBundle` class object should first be created and then following methods should be invoked:
  - `MessagesBundle_en_US.properties`
  - `MessagesBundle_fr_FR.properties`
  - `MessagesBundle_de_DE.properties`
  - `MessagesBundle_ja_JP.properties`
- To retrieve locale-specific data from properties file, the `ResourceBundle` class object should first be created and then, following methods should be invoked:
  - `public final String getString(String key)`
  - `public abstract Enumeration<String>getKeys()`

Internationalization Process [4-4]

Fetching the Text from the `ResourceBundle` Class:

- The properties files contain key-value pairs.
- The values consist of the translated text that the program will display.
- The keys are specified when fetching the translated messages from the `ResourceBundle` with the `getString()` method.
- Following statement illustrates how to retrieve value from the key-value pair using the `getString()` method:

```
String msg1 = messages.getString("greetings");
```

- The statement uses key greetings because it reflects the content of the message.

Internationalization Elements [1-9]

- There are various elements that vary with language, region, and culture.
- All such elements must be internationalized.

Component Captions:

- These refer to the GUI component captions such as numerals, date, and text.
- These captions must be localized because their usage varies with region, culture, and language.
- It also ensures that the look and feel of the application is in a locale-sensitive manner.
- The code that displays the GUI is locale-independent.

Numbers, Currencies, and Percentages:

- The format of numbers, currencies, and percentages vary with culture, region, and language.
- It is necessary to format them before displaying.
- Similarly, the currency symbols and methods of displaying the percentage factor also vary with region and language.
- Formatting is required to make an internationalized application.
- This is with regards to percentage representation, thousands-separators, and decimal-point.
- NumberFormat class is used to create locale-specific formats for:
  - Numbers, Currencies, and Percentages

Number:

- `NumberFormat` class has a static method `getNumberInstance()`.
- This method returns an instance of a `NumberFormat` class.
- The `format()` method of the `NumberFormat` class should be invoked next.

Internationalization Elements [2-9]

- Syntaxes for some of the methods that are used for formatting numbers are:
  - `public static final NumberFormat getNumberInstance()`
  - `public final String format(double number)`
  - `public static NumberFormat getNumberInstance(Locale inLocale)`
- Code Snippet shows how to create locale-specific format of number for Japan.

Code Snippet

```
import java.text.NumberFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {
static public void printValue(Locale currentLocale) {
Integer value = new Integer(123456);
Double amt = new Double(345987.246);
NumberFormat numFormatObj;
String valueDisplay;
String amtDisplay;
numFormatObj = NumberFormat.getNumberInstance(currentLocale);
valueDisplay = numFormatObj.format(value);
amtDisplay = numFormatObj.format(amt);
System.out.println(valueDisplay + " " + currentLocale.toString());
System.out.println(amtDisplay + " " + currentLocale.toString());
}
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
```

Internationalization Elements [3-9]

```
String language;
String country;
if (args.length != 2) {
language = new String("en");
country = new String("US");
}
else {
language = new String(args[0]);
country = new String(args[1]);
}
Locale currentLocale; ResourceBundle messages;
currentLocale = new Locale(language, country);
messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
printValue(currentLocale);
}
}
```

Currencies:

- `NumberFormat` class has a static method `getCurrencyInstance()`.
- It takes an instance of Locale class as an argument.
- `getCurrencyInstance()` method returns an instance of a `NumberFormat` class initialized for the specified locale.
- Syntaxes for some of the methods to format currencies are:
  - `public final String format(double currency)`
  - `public static final NumberFormat getCurrencyInstance()`
  - `public static NumberFormat getCurrencyInstance(Locale inLocale)`

Internationalization Elements [4-9]

Currencies:

- `NumberFormat` class has a static method `getCurrencyInstance()`.
- It takes an instance of Locale class as an argument.
- `getCurrencyInstance()` method returns an instance of a `NumberFormat` class initialized for the specified locale.
- Syntaxes for some of the methods to format currencies are:
  - `public final String format(double currency)`
  - `public static final NumberFormat getCurrencyInstance()`
  - `public static NumberFormat getCurrencyInstance(Locale inLocale)`
- Code Snippet shows how to create locale-specific format of currency for the country, France:

Code Snippet

```
NumberFormat currencyFormatter;
String strCurrency;
// Creates a Locale object with language as French and country as France
Locale locale = new Locale("fr", "FR");
// Creates an object of a wrapper class Double
Double currency = new Double(123456.78);
// Retrieves the CurrencyFormatter instance
currencyFormatter = NumberFormat.getCurrencyInstance(locale);
// Formats the currency
strCurrency = currencyFormatter.format(currency);
```

Internationalization Elements [5-9]

Percentages:

- This class has a static method `getPercentInstance()`.
- It takes an instance of the `Locale` class as an argument.
- This method returns an instance of the `NumberFormat` class initialized to the specified locale.
- Syntaxes for some of the methods to format percentages are:
  - `public final String format(double percent)`
  - `public static final NumberFormat getPercentInstance()`
  - `public static NumberFormat getPercentInstance(Locale inLocale)`
- Code Snippet shows how to create locale-specific format of percentages for the country, France:

Code Snippet

```
NumberFormat percentFormatter;
String strPercent;
// Creates a Localeobject with language as French and country
// as France
Locale locale = new Locale("fr", "FR");
// Creates an object of a wrapper class Double
double percent = 25f / 100f;
// Retrieves the percentFormatter instance
percentFormatter = NumberFormat.getPercentInstance(locale);
// Formats the percent figure
strPercent = percentFormatter.format(percent);
System.out.println(strPercent);
```

Internationalization Elements [6-9]

Date and Times:

- Date and time format must conform to the end user's locale conventions.
- Date and time format varies with culture, region, and language.
- Java provides the `java.text.DateFormat` and `java.text.SimpleDateFormat` classes to format date and time.
- The `DateFormat` class is used to create locale-specific formats for date.
- The `format()` method of the `NumberFormat` class is also invoked.
- The `DateFormat getDateInstance(style, locale)` method returns an instance of the class `DateFormat`.
- Following is the syntax:

```
public static final DateFormatgetDateInstance(int style, Locale locale)
```

- Code Snippet demonstrates how to retrieve a `DateFormat` object and display the date in Japanese format.

Code Snippet

```
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
public class DateInternationalApplication {
public static void main(String[] args) {
Date today;
```

Internationalization Elements [7-9]

```
String strDate;
DateFormatdateFormatter;
Locale locale = new Locale("ja", "JP");
dateFormatter = DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
today = new Date();
strDate = dateFormatter.format(today);
System.out.println(strDate);
}
}
```

Messages:

- Displaying messages are an integral part of any software.
- If the messages are predefined, they can be easily translated into various languages.
- If the messages contain variable data, it is difficult to create grammatically correct translations for all languages.
- The position of verbs and the variable data varies in different languages.

Internationalization Elements [8-9]

- The approach of concatenation works fine in English.
- However, it does not work for languages in which the verb appears at the end of the sentence.
- The solution is to use the `MessageFormat` class.
- To use the `MessageFormat` class, perform following steps:

Step 1
- Identify the variables in the message.
- write down the message, and identify all the variable parts of the message.

Step 2
- Create a template. It contains the variable parts and the fixed part of the message. Variable parts are encoded in {} with an argument number.
- Each argument number should match with an index of an element in an `Object` array containing argument values.

Step 3
- Create an `Object` array for variable arguments.
- If an element in the array requires translation, it should be fetched from the Resource Bundle with the `getString()` method.

Step 4
- Create a `MessageFormat` instance and set the desired locale.

Step 5
- Apply and format the pattern.
- Fetch the pattern string from the Resource Bundle with the `getString()` method.
- Once the pattern is applied to the `MessageFormat` instance, invoke the `format()` method.

Internationalization Elements [9-9]

- Code Snippet will display the message in German using `MessageFormatter` when executed.

Code Snippet

```
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;
import java.util.ResourceBundle;
public class MessageFormatterInternationalApplication {
    /**
    * @param args the command line arguments
    */
    public static void main(String[] args) {
    String template = "At {2,time,short} on {2,date,long}, we detected
    {1,number,integer} virus on the disk {0}";
    MessageFormat formatter = new MessageFormat("");
    String language, country;
    if (args.length != 2) {
    language = new String("en");
    country = new String("US");
    }
    else {
    language = new String(args[0]);
    country = new String(args[1]);
    }
    Locale currentLocale;
    currentLocale = new Locale(language, country);
    formatter.setLocale(currentLocale);
    ResourceBundle messages = ResourceBundle.getBundle (
    "MessageFormatBundle", currentLocale);
    Object[] messageArguments = {messages.getString("disk"), new
    Integer(7), new Date()};
    formatter.applyPattern(messages.getString("template"));
    String output = formatter.format(messageArguments);
    System.out.println(output);
    }
}
```

Parallelization

Java supports parallelization.

Parallelization can make applications run faster by efficiently using multi-core processors.

Parallelizing any type of workload comes with challenges particularly in the partitioning step.

Such challenges are addressed by work stealing, which is a scheduling strategy for multi-threaded programs.

The Fork-Join framework introduced in Java 7 meets the requirements of work stealing.

This is through recursive job partitioning and double-ended queue (deque) structure for holding the tasks.

The `ForkJoinTask.join()` method allows a thread to avoid blocking itself.

Instead, it asks the pool for any work it should do.

Enhancements in the java.util.concurrent Package [1-7]

- Java has made several enhancements in the `java.util.concurrent` package from version 8 onwards.
- `CompletableFuture.AsynchronousCompletionTask` is an interface in the `java.util.concurrent` package.
- It is useful to monitor, debug, and track asynchronous operations.
- Another new interface is `CompletionStage<T>`.
- Java also has an exception, `CompletionException` in the package.
- A `CompletionException` is thrown when an error or other exception is encountered in the course of completing a result or task.
- In addition, Java comes with following classes in the `java.util.concurrent` package:
  - `CompletableFuture`
  - `CountedCompleter`
  - `ConcurrentHashMap.KeySetView`

Enhancements in the java.util.concurrent Package [2-7]

CompletableFuture Class:

- CompletableFuture class implements the CompletionStage and the Future interface.
- These interfaces simplify coordination of asynchronous operations.
- get() method of Future returns the result of a computation when:
  - The computation completes
  - It is canceled
  - Or an exception gets thrown
- Methods of the CompletableFuture class can run asynchronously.
- These do not stop program execution.
- Following are the important methods available in the CompletableFuture class:
  - supplyAsync()
  - thenApply()
  - join()
  - thenAccept()
  - whenComplete()
  - getNow()

Enhancements in the java.util.concurrent Package [3-7]

CountedCompleter Class:

- `CountedCompleter` class extends `ForkJoinTask`.
- It represents a completion action performed when triggered, provided there are no pending actions.
- `compute()` method of the `CountedCompleter` class does the main computation.
- It typically invokes the `tryComplete()` method once before returning.
- `tryComplete()` method checks if the pending count is nonzero, and if so, decrements the count.
- Otherwise, `tryComplete()` method invokes the method `onCompletion(CountedCompleter)`.
- It attempts to complete this task's completer, and if successful, marks this task as complete.
- Optionally, the `CountedCompleter` class may override following methods:
  - `onCompletion(CountedCompleter)`
  - `onExceptionalCompletion(Throwable, CountedCompleter)`

Enhancements in the java.util.concurrent Package [4-7]

- Code Snippet shows the implementation of the `CountedCompleter` class:

Code Snippet

```
import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.CountedCompleter;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
public class CountedCompleterDemo {
static class NumberComputator extends CountedCompleter<Void> {
      final ConcurrentLinkedQueue<String> concurrentLinkedQueue;
      final int start;
      final int end;
      NumberComputator(ConcurrentLinkedQueue<String> concurrentLinkedQueue, int start, int
      end) {
           this(concurrentLinkedQueue, start, end, null);
      }
     NumberComputator(ConcurrentLinkedQueue<String> concurrentLinkedQueue, int start, int
     end, NumberComputator parent) {
       super(parent);
       this.concurrentLinkedQueue = concurrentLinkedQueue; this.start = start;
       this.end = end;
        }
   @Override
   public void compute() { if (end - start < 5) {
   String s = Thread.currentThread().getName();
   for (int i = start; i < end; i++) {
            concurrentLinkedQueue.add(String.format("Iteration number: {%d} performed by
            thread {%s}", i, s));
   }
```

Enhancements in the java.util.concurrent Package [5-7]

```
    propagateCompletion();
  }
   else {
        int mid = (end + start) / 2;
        NumberComputator subTaskA = new NumberComputator (concurrentLinkedQueue,
        start, mid, this);
        NumberComputator subTaskB = new NumberComputator (concurrentLinkedQueue,
        mid, end, this);
        setPendingCount(1); subTaskA.fork(); subTaskB.compute();
   }
   }
   }
   public static void main(String[] args) throws ExecutionException,
   InterruptedException {
    ConcurrentLinkedQueue<String> linkedQueue = new ConcurrentLinkedQueue<>();
    NumberComputator numberComputator = new NumberComputator(linkedQueue, 10,
    100);
    ForkJoinPool.commonPool().invoke(numberComputator);
    ArrayList<String> list = new ArrayList<>(linkedQueue);
    for (String listItem : list) {
    System.out.println(" " + listItem);
    }
    }
}
```

Enhancements in the java.util.concurrent Package [6-7]

ConcurrentHashMap.KeySetViews Class:

- `ConcurrentHashMap.KeySetView` provides a convenient view of the keys contained in a `ConcurrentHashMap`.
- `ConcurrentHashMap.KeySetView` implements the `Set` interface,
- Therefore, you access the `ConcurrentHashMap` keys as a `Set` object.
- `Set` object and `ConcurrentHashMap` object maintains a bi-directional relationship.
- Code Snippet shows creation of a `KeySetViewDemo` class to access the keys of a `ConcurrentHashMap` as a Set.

Code Snippet

```
import java.util.Map; import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
public class KeySetViewDemo {
    public static void main(String[] args) {
        Map<String,String> map = new ConcurrentHashMap<>();
        map.put("Java", "Java");
        map.put("Java EE", "Java EE");
        map.put("Spring", "Spring");
        Set keySet = map.keySet(); System.out.println(keySet);
    }
}
```

Enhancements in the `java.util.concurrent` Package [7-7]

More Recent Enhancements:

- Java 16 consists of several enhancements in the `java.util.concurrent` package.
- Utility classes are generally useful in concurrent programming.
- Specific standardized and expandable frameworks are present in the package.
- Following are the major modules:
  - Executor Interfaces
  - Implementation Modules
  - Queues
  - Synchronizers
  - Timing
  - Concurrent Collections
  - Memory Consistency Properties

Atomic Operations and Locks [1-5]

- One challenge is to maintain a single count or sum that is simultaneously updated by multiple threads.
- In Java 8, the challenge was addressed through a small set of new classes in the `java.util.concurrent.atomic` package.
- New classes that were introduced in Java 8 are:
  - `LongAccumulator`
  - `LongAdder`
  - `DoubleAccumulator`
  - `DoubleAdder`

Atomic Operations and Locks [2-5]

Code Snippet

```
import java.util.concurrent.atomic.DoubleAdder;
import java.util.concurrent.atomic.LongAdder;
public class AtomicOperationClassDemo {
   private final LongAdder longAdder;
   private final DoubleAdder doubleAdder;
   public AtomicOperationClassDemo(LongAdder , DoubleAdder
   doubleAdder) {
   this.longAdder = longAdder;
   this.doubleAdder = doubleAdder;}
   }
   public long getLongCounter() {
   return longAdder.longValue();
   }
   public void addDouble(int doubleValue) {
   doubleAdder.add(doubleValue);
   }
   public double getSumAsDouble() { return doubleAdder.doubleValue();
   }
   public static void main(String[] args) {
   AtomicOperationClassDemo atomicOperationClassDemo1 = new
   AtomicOperationClassDemo(new LongAdder(), new DoubleAdder());
   System.out.println("-----Long Counter-----");
   for (int i = 0; i < 10; i++) { atomicOperationClassDemo1.incrementLong();
   System.out.println("Long Counter " +
   atomicOperationClassDemo1.getLongCounter());
   }
   System.out.println("-----Double Sum-----");
   for (int j = 0; j < 10; j++) {
   atomicOperationClassDemo1.addDouble(j);
   System.out.println("Double Sum " +
   atomicOperationClassDemo1.getSumAsDouble());
   }
   }
}
```

Atomic Operations and Locks [3-5]

- Java includes the `StampedLock` class of the `java.util.concurrent.locks` package.
- `StampedLock` class implements lock with three modes for controlling read/write access.
- These modes are:
  - Writing
  - Reading
  - Optimistic Reading
- Code Snippet shows the use of the `StampedLock` class in the reading, writing, and optimistic reading mode.

Code Snippet

```
import java.util.concurrent.locks.StampedLock;
public class StampedLockDemo {
   private final StampedLock stampedLock = new StampedLock();
   private double balance;
   public StampedLockDemo(double balance) {
       this.balance = balance;
       System.out.println("Available balance: "+balance);
   }
   public void deposit(double amount) {
       System.out.println("\nAbout to deposit $: "+amount);
       long stamp = stampedLock.writeLock();
       System.out.println("Applied read lock");
       try {
       balance += amount;
```

Atomic Operations and Locks [4-5]

```
System.out.println("Available balance: "+balance);

} finally {
   stampedLock.unlockWrite(stamp);
   System.out.println("Unlocked write lock");
}
}
public void withdraw(double amount) {
     System.out.println("\n About to withdraw $: "+amount);
     long stamp = stampedLock.writeLock();
     System.out.println("Applied write lock");
     try {
     balance -= amount;
     System.out.println("Available balance: "+balance);
     }
     finally {
     stampedLock.unlockWrite(stamp);
     System.out.println("Unlocked write lock");
     }
}
public double checkBalance() {
   System.out.println("\nAbout to check balance");
   long stamp = stampedLock.readLock();
   System.out.println("Applied read lock");
   try {
   System.out.println("Available balance: "+balance);
   return balance;
   }
   finally {
   stampedLock.unlockRead(stamp);
   System.out.println("Unlocked read lock");
   }
}
```

Atomic Operations and Locks [5-5]

```
    public double checkBalanceOptimisticRead() {
         System.out.println("\nAbout to check balance with optimistic read lock");
         long stamp = stampedLock.tryOptimisticRead();
         System.out.println("Applied non-blocking optimistic read lock");
         double balance = this.balance;
         if (!stampedLock.validate(stamp)) {
         System.out.println("Stamp have changed. Applying full-blown read lock.");
         stamp = stampedLock.readLock();
         try { balance = this.balance;}
         finally {
         stampedLock.unlockRead(stamp);
         System.out.println("Unlocked read lock");
         }
         }
         System.out.println("Available balance: "+balance);
         return balance;
    }
     public static void main(String[] args){
         StampedLockDemo stampedLockDemo=new StampedLockDemo(4000.00);
         stampedLockDemo.withdraw(1000.00);
         stampedLockDemo.deposit(5000.00);
         stampedLockDemo.checkBalance();
         stampedLockDemo.checkBalanceOptimisticRead();
       }
}
```

More Features of the Fork-Join Framework

- Some additional features of the Fork-Join framework include features added to the `ForkJoinPool` class in Java 8, stream parallelization, and parallelization of array sorting.

New `ForkJoinPool` Features

- Java 8 introduced the common thread pool feature.
- It allows any `ForkJoinTask` that to use a common thread.
- Given that it is not explicitly submitted to a specified thread pool.
- This helps applications to reduce resource usage.
- New methods of the `ForkJoinPool` class are:
  - `commonPool()`
  - `getCommonPoolParallelism()`

Stream Parallelization

- Stream API in Java operates on collections and arrays.
- It helps to produce pipelined data used to perform operations.
- These include iterating, sorting, and filtering through data.
- With parallel computation, streams can now be accessed faster without the risk of threading issues.
- Code Snippet shows the use of parallel stream to iterate through the elements of an `ArrayList`:

Code Snippet

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
public class ParallelStreamDemo {
   public static void main(String[] args) {
   List<String> items = new ArrayList<String>();
   items.add("one");
   items.add("two");
   items.add("three");
   items.add("four");
   Stream parallelStream = items.parallelStream();
   parallelStream.forEach(System.out::println);
   }
}
```

Arrays Sort Parallelism

- Java 8 introduces a new `parallelSort()` method in the Arrays class.
- It allows parallel sorting of array elements.
- Code Snippet shows how to sort arrays in parallel assuming that relevant import statements are added.

Code Snippet

```
import java.util.Arrays;
public class ParallelArraySortDemo {
public static void main(String[] args) {
int[] intArray = new int[100];
for(int i = 0; i < intArray.length; i++) {
intArray[i] = (int)(Math.random() * 100);
}
Arrays.parallelSort(intArray);
System.out.println(Arrays.toString(intArray));
}
}
```

Recursive Action [1-2]

- `ForkJoinPool` class extends `AbstractExecutorService` class.
- It implements main work-stealing algorithm to execute `ForkJoinTask` processes.
- `RecursiveTask` and `RecursiveAction` are similar in functioning.
- Both extend `ForkJoinTask` to represent tasks that run within a `ForkJoinPool`.
- Code Snippet shows the use of Fork-Join functionality with `RecursiveAction`:

Code Snippet

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
public class RecursiveActionDemo extends RecursiveAction {
private long assignedWork = 0;
public RecursiveActionDemo(long assignedWork) {
this.assignedWork = assignedWork;}
private List<RecursiveActionDemo> createSubtasks() {
List<RecursiveActionDemo> subtaskList = new ArrayList<>();
RecursiveActionDemo subtask1 = new RecursiveActionDemo( this.assignedWork / 2);
RecursiveActionDemo subtask2 = new RecursiveActionDemo ( this.assignedWork / 2);
subtaskList.add(subtask1);
subtaskList.add(subtask2);
```

Recursive Action [2-2]

```
    return subtaskList;
    }
    @Override
    protected void compute() {
    if (this.assignedWork > 50) {
    System.out.println("Splitting assignedWork : " + Thread. currentThread() + "
    computing: : " + this.assignedWork);
    List<RecursiveActionDemo> subtaskList = new ArrayList<>();
    subtaskList.addAll(createSubtasks());
    for (RecursiveAction subtask : subtaskList) {
      subtask.fork();
    }
    }
    else {
    System.out.println("Main thread " + Thread.currentThread() + " computing: : "
    + this.assignedWork);
    }
    }
    public static void main(String[] args) { RecursiveActionDemo
    recursiveActionDemo = new RecursiveActionDemo(500);
    final ForkJoinPool forkJoinPool = new ForkJoinPool(4);
    forkJoinPool.invoke(recursiveActionDemo);
    }
}
```

Summary

- A design pattern is a clearly defined solution to problems that occur frequently.
- It can be considered as a template or a best practice suggested by expert programmers.
- The singleton design pattern provides complete information on class implementations that can be instantiated only once.
- Data Access Object (DAO) Design Pattern is used when an application is created that must persist its data.
- Factory pattern is one of the commonly used design patterns in Java and belongs to the creational pattern category and provides many ways to create an object.
- The Observer pattern helps to observe the behavior of objects such as change in state or change in property.
- In the internationalization process, the input and output operations of an application are specific to different locations and user preferences. Improvements as compared to atomic variables.
- `java.util.concurrent` includes many enhancements in the form of interfaces and classes to support concurrency and parallelism.