Coding Bootcamp

Core Java

TECHNOLOGY

Packages and Multithreading

simplilearn

Learning Objectives

By the end of this lesson, you will be able to:

- Compare and contrast interfaces and classes in Java to highlight their structural differences and appropriate applications
- Demonstrate extending interfaces using the extends keyword to illustrate hierarchical interface inheritance in Java
- Utilize packages to organize Java code hierarchically and demonstrate how to import them for efficient project structuring

Learning Objectives

By the end of this lesson, you will be able to:

- Evaluate and select the right Java access modifiers for specific scenarios to ensure optimal data protection and visibility
- Design and implement multithreaded Java programs using Runnable interfaces, synchronized blocks, and advanced concurrency constructs like Futures and Promises to enhance application performance and manageability

TECHNOLOGY

Java Interfaces

Java Interfaces

An interface is a collection of abstract methods that defines a set of behaviors a class can implement.

- They look like classes but are different.
- They do not have instance variables or state.
- Methods in interfaces are declared without any body and end with a semicolon.
- They cannot be instantiated and thus cannot have constructors.
- They are implemented by a class and can implement multiple interfaces.

Syntax

```
accessModifier interface name {
 type final_var1 = value;
 type final_var2 = value;
 ----
 ----
 returnType method_name1(param_list);
 returnType method_name2(param_list);
 ----
 ----
}
```

Java Interfaces

Top-level interfaces are public by default and can only have public or default access modifiers. Nested interfaces can have public, private, or protected access modifiers.

Example

```
public class class_name implements abc {
    @Override
    public Integer method_1() {
        System.out.println("in method 1 "
+ a);
        return null;
    }

    @Override
    public void method_2(String Id) {
        System.out.println("in method
2");
    }
```


Interface variables are public, final, and static and must be initialized. Interface methods are abstract and public and must be implemented by a class.

Java Interfaces: Implements Keyword

A class uses the **implements** keyword to implement an interface.

Example

```
public class class_name implements abc {
    @Override
    public Integer method_1() {
        System.out.println("in method 1 "
+ a);
        return null;
    }

    @Override
    public void method_2(String Id) {
        System.out.println("in method
2");
    }
```

A class can implement more than one interface, which is separated by a comma.

Example

```
public static void main(String[] args) {
        // Instantiate class_name and
call its methods to see outputs.
        class_name instance = new
class_name();
        instance.method_1();
        instance.method_2("someID");
    }
}
```

Java Interfaces: Rules

- Methods defined in an interface and implemented by a class must be declared public.
- The signature of these methods should remain unchanged during implementation.
- Variables initialized in an interface are constant and cannot be altered.
- A class can extend only one class but can implement multiple interfaces.
- An interface can extend another interface.

Java Interfaces: Extend Keyword

Interfaces can extend other interfaces with the **extends** keyword, but a class can only extend one class.

```
public interface abc{
 int a = 3;
 Number method1();
 void method2(String Id);
}
// extending interface
interface B extends abc{
 void method3();
}
// class implements all methods of abc and
B
public class MyClass implements B {
```

```
public Integer method1() {
  System.out.println("in method 1" + i);
  return null;
 }
public void method2(String Id) {
  System.out.println("in method 2");
 }
 public void method3() {
  System.out.println("in method 3");
 }
 public static void main(String[] args) {
  }}
```

If an interface extends another interface, any class that implements it must provide implementations for all methods.

Java Interfaces: Abstract Class

If a class does not implement all the interface methods, it should be declared abstract.

```
public interface abc {
 void method1();
 String method2(String Id);
}
```

```
public abstract class AbstractClassDemo
implements abc {
 public static void main(String[] args) {
  System.out.println();
 }
}
```

Java Interfaces: Nested Interface

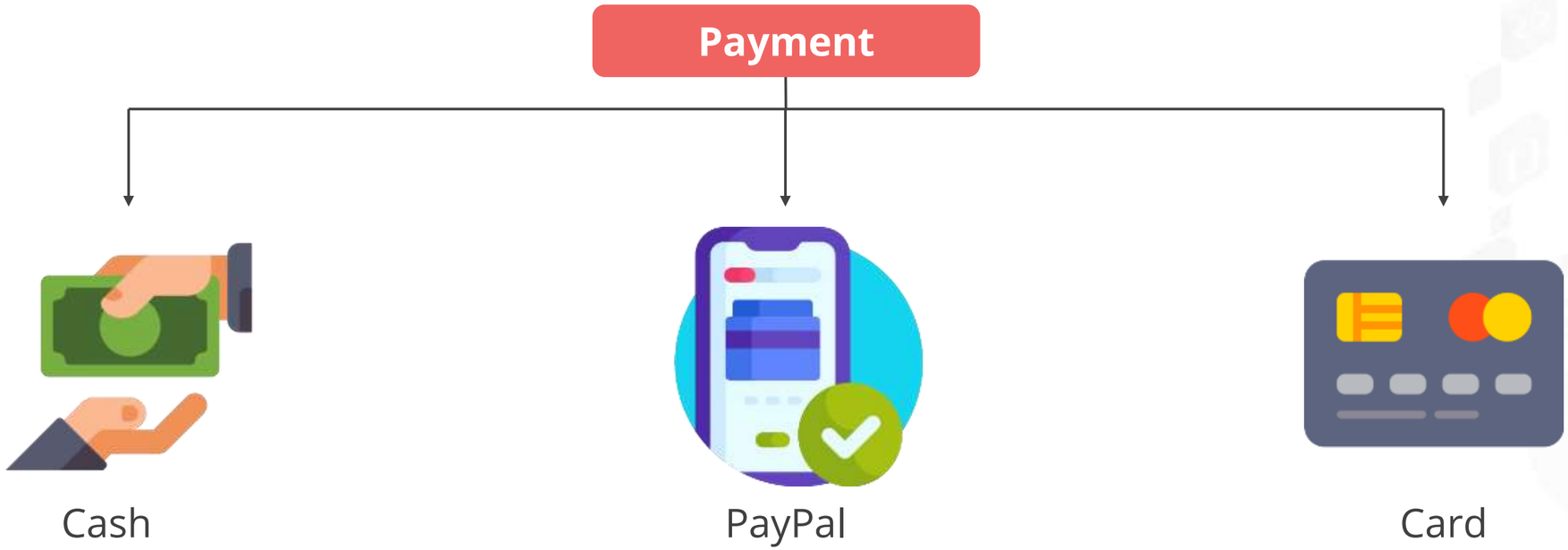
An interface can have another interface, also called a nested interface. It can be declared public, private, or protected.

```
class A{
 public interface TestInterface{
  void displayValue(String value);
 }
}
// class implementing the nested interface
class B implements A.TestInterface{
 public void displayValue(String value) {
  System.out.println("Value is " + value);
 } }
public class MyClass{
 public static void main(String[] args) {
  // reference of class B assigned to nested interface
  A.TestInterface obRef = new B();
  obRef.displayValue("hello");
 } }
```

Java Interfaces: Example

Interfaces cannot be instantiated as objects but can create object references. They allow for runtime polymorphism using a superclass reference and improve functionality by holding references to subclasses.

Example:



Here, an application handles payments through various modes, such as cash, PayPal, and card. Each has specific functionalities to be handled accordingly.

Java Interfaces: Example

The use of interfaces and inheritance for various types of method implementation is shown below:

```
// Super Class
public interface PaymentInt {
 public void payment(double amount);
}
// Cash Payment implementation of Payment interface
class CashPayment implements PaymentInt{
 // method implementation according to cash payment
functionality
 public void payment(double amount) {
  System.out.println("Cash Payment of amount " +
amount);
 } }
//PayPal Payment implementation of Payment interface
class PayPalPayment implements PaymentInt{
 // method implementation according to PayPal
payment functionality
 public void payment(double amount) {
  System.out.println("PayPal Payment of amount " +
amount);
 } }
// Child class
class Circle extends Shape{
```

```
//Card Payment implementation of Payment interface
class CardPayment implements PaymentInt{
 // method implementation according to card payment
functionality
 public void payment(double amount) {
  System.out.println("Card Payment of amount " +
amount);
 } }
public class PaymentDemo {
 public static void main(String[] args) {
  // Payment interface reference holding the
CashPayment obj
  PaymentInt paymentInt = new CashPayment();
  paymentInt.payment(…);
  // Payment interface reference holding the
CardPayment obj
  paymentInt = new CardPayment();
  paymentInt.payment(…);
  // Payment interface reference holding the
PayPalPayment obj
  paymentInt = new PayPalPayment();
  paymentInt.payment(…;
 } }
```

Creating Interfaces and Multiple Implementation

Problem Statement:

You have been asked to create multiple Interfaces.

Outcome:

By creating multiple interfaces in Java, you will learn how to define contracts for your classes, ensuring they implement specific methods. This task will enhance your ability to design flexible and scalable systems through decoupled components.

Note: Refer to the demo document for detailed steps: 01_Creating_Interfaces_and_Multiple_Implementation

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Implement the concept of interfaces
2. Comprehend how interfaces work
3. Write polymorphic statements using interfaces
4. Implement how multiple inheritances work in interfaces
5. Link interfaces to classes and methods

Implementing Anonymous Classes

Problem Statement:

You have been asked to demonstrate the implementation of anonymous classes.

Outcome:

By demonstrating the implementation of anonymous classes in Java, you will learn how to create and use unnamed classes for instant use. This skill is particularly useful for simplifying code when implementing interfaces or extending classes without creating a separate named class.

Note: Refer to the demo document for detailed steps: 02_Implementing_Anonymous_Classes

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the IDE and create a new project
2. Write an interface and execute the code with example data
3. Execute the code and print the message accordingly
4. Override the methods called on success and failure
5. Use anonymous classes

Implementing Abstraction with Interfaces

Problem Statement:

You have been asked to implement abstractions using interfaces.

Outcome:

By implementing abstractions using interfaces in Java, you will learn how to define methods that must be implemented by any class that uses the interface. This practice helps create a clear and consistent design, promoting flexible and scalable software development.

Note: Refer to the demo document for detailed steps: 03_Implementing_Abstraction_with_Interfaces

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the IDE and create a new project
2. Consider a class marked as abstract and two methods, each for failure and success
3. Execute the code with example data
4. Make the abstract class as an interface
5. Understand the polymorphic statement and execute the code

TECHNOLOGY

Java Packages

Java Packages

A package in Java is a group of similar classes, interfaces, and sub-packages.

Advantages of packages include:

- Organized structure for related classes and interfaces
- Access and visibility control
- Prevent naming collisions

It provides access controls and helps manage classes as a single unit, like folders in an OS. Non-visible class members can be defined inside a package.

Java Packages: Categories

In Java, there are two categories of packages:

Built-in packages

Provided by the Java API and contains classes grouped according to functionality

User-defined packages

Created by the users to group their classes and interfaces according to their desired functionality

Java Packages: Package Keyword

Syntax

```
package <package_name>;
class class_name
{
……..
……..
}
```

Example

```
package p;
class example
{
    public void method()
    {
        System.out.println("Hello
world!");
    }
}
```

Java Packages: Example

Command to compile a package:

```
Java -d directory JavaFileName
```

Java packages have a hierarchy where sub-packages are nested inside parent packages, with package names separated by a '.'.

Example:

```
package org.prog;
public class Example{
 public static void main(String[]
args) {
  Example E = new Example();
 }}
```

The file will be placed under the folder structure **\org\prog** in the Windows environment.

How to Import Packages in Java?

To import classes or interfaces from one package into another, use the following steps:

Use the fully qualified name, including the package and class or interface names.

Use the **import** keyword to create a shorter alias for the fully qualified name.

Note:

Import statements should be written directly after the package statement.

How to Import Packages in Java?

In the user's class, use the **List** interface from **Java.util**. The user can import the full package:

```
import java.util.*;

List<Integer> ls = new ArrayList<>();
```

Importing classes from the package requires the use of import statements.

```
import java.util.ArrayList;
import java.util.List;

List<Integer> ls = new ArrayList<>();
```

Access Modifier

Access Modifiers

They control the visibility of class members and determine if other classes can use fields, methods, or create objects of that class.

Access Modifier	Within Class	Within Package	Outside package by subclass only	Outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Access Modifiers: Private

Members declared as private are only accessible within the same class.

Example

```
class X{
private int data=10;
private void
msg(){System.out.println("Hello
World");}
}
 public class Y{
 public static void main(String args[]){
   X obj = new X();
   System.out.println(obj.data);
//Compile Time Error
   obj.msg();
   }  }
```

There are two classes: X and [Y], where [X] has private methods and data members. Accessing these private members from outside the class results in a compile-time error.

Example

```
class X{
private X(){} //private constructor
void msg(){System.out.println("Hello
World");}
}
public class Y{
 public static void main(String args[]){
   X obj=new X(); //Compile Time Error
 }
}
```

An instance of the class constructor declared private cannot be created.

Access Modifiers: Default

Members declared with no access modifier (or package-private) are accessible within the same package.

Example

```
package X;
class Notification{
  void msg(){System.out.println("Hello
World");}
}
//save by InstaNotification.Java
package Y;
import X.*;
class InstaNotification{
  public static void main(String args[]){
  Notification obj = new Notification();
//Compile Time Error
   obj.msg(); //Compile Time Error
  }  }
```

Class A is not declared public; it is only accessible within the same package.

In this example, the class and its method msg() are default, so they cannot be accessed outside the package.

Access Modifiers: Protected

Members declared protected are accessible within the same package and by subclasses in different packages.

Example

```
package X;
public class Notification{
protected void
msg(){System.out.println("Hello");}
}
//save by InstaNotification.Java
package Y;
import X.*;
  class InstaNotification extends
Notification{
  public static void main(String args[]){
  InstaNotification obj = new
InstaNotification();
```

Class A in package X is public, allowing external access. The msg method in package X is protected and can only be accessed externally through inheritance.

Method msg() of class A is declared default, meaning it can only be accessed within the same package and not from outside.

Access Modifiers: Public

Members declared as public are accessible from anywhere, both within and outside of the package.

Example

```
package X;
public class Notification{
public void msg(){System.out.println("Hello");}
}
//save by InstaNotification.Java
package Y;
import X.*;  // Importing all classes from
package X

public class InstaNotification {
    public static void main(String[] args) {
        Notification obj = new Notification();
// Create an instance of Notification
        obj.msg();
// Call the public method msg
    }
}
```

Class A is public and can be accessed from outside the package.

Access Modifiers: Example

The method override declared in a subclass should not have a more restrictive access level than the method it overrides in the superclass.

Example

```
class Notification {
    protected void msg() {
        System.out.println("Hello World");
    }
}

public class InstaNotification extends Notification {
    @Override
    protected void msg() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        InstaNotification obj = new InstaNotification();
        obj.msg();
    }
}
```

There is a compile-time error because the protected method is less restrictive than the default modifier.

Creating and Using Design Patterns Factory and State

Problem Statement:

You have been asked to implement the Factory and State design patterns.

Outcome:

By implementing the Factory and State design patterns in Java, you will learn to create objects without exposing the instantiation logic and to alter an object's behavior when its state changes. This knowledge enhances your ability to write modular, maintainable, and scalable code, crucial for complex software development.

Note: Refer to the demo document for detailed steps: 04_Creating_and_Using_Design_Patterns_Factory_and_State

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Implement and create a Factory Design Pattern class and Plan interface
2. Create the class Plan3G
3. Create the classes Plan4G and Plan5G
4. Create the class Plan Factory
5. Implement the methods of the interface and classes
6. Create State Design Patterns

Creating Packages and Access Modifiers

Problem Statement:

You have been asked to implement packages and access modifiers in Java.

Outcome:

By implementing packages and access modifiers in Java, you will learn to organize classes and interfaces into namespaces and control their accessibility within other parts of your program. This skill is essential for managing large codebases and maintaining encapsulation, which is fundamental to robust software architecture.

Note: Refer to the demo document for detailed steps: 05_Creating_Packages_And_Access_Modifiers

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a Java package and class
2. Implement the use of access modifiers in class declarations
3. Implement the use of access modifiers inside the class
4. Implement the use of access modifiers outside the class
5. Implement the difference between protected and default access modifiers

TECHNOLOGY

Exception Handling

Exception Handling

It handles program errors. When an exceptional condition occurs, a message is displayed to the user, and the program flow may be interrupted or terminated.

Code that may throw an exception is enclosed in a try block.

Exception handlers are provided in catch blocks to handle exceptions thrown in the try block.

Finally, the block is executed after the try block, regardless of whether an exception is thrown.

Syntax for exception handling block:

```
try {
  // block of code
}
catch (Exception_type1 exp_obj) {
  // exception handler for Exception_type1}
catch (Exception_type2 exp_obj) {
  // exception handler for Exception_type2
}

// ...
finally {
  // block of code to be executed after try block
ends
}
```

Exception Handling: Issues

Errors and exceptions are the two types of issues that can arise during the execution of a program. They are both members of the throwable class.

Exceptions

A type of throwable that indicates a problem occurred during the execution of a program and can be handled by the program code using try-catch blocks or other mechanisms.

Errors

A type of throwable that represents serious problems that a program, such as system failures or out-of-memory errors, cannot handle.

Exception Handling: Try-Catch

A try block encloses code that may throw an exception. It should be used within a method. If an exception is thrown, the remaining code within the try block will not execute.

Syntax for try-catch

```
try {
    // Code that might throw an
exception
} catch (ExceptionType name) {
    // Code to handle the exception
}
```

Syntax for try-finally block

```
try {
    // Code that might throw an
exception
} finally {
    // Code that has to execute
after the try block, regardless
of the outcome
}
```

In Java, it is necessary to have either a catch or a final block after a try block.

Try-Catch: Example

Example for try-catch block

```
//code
public class Exception_Example {
 public static void main(String[] args) {
  int b = 0;
  List <Integer> numList = new
ArrayList<Integer>();
  // Putting values in a list
  numList.add(2);
  numList.add(3);
  numList.add(0);// putting zero
  numList.add(6);
  numList.add(8);
  // looping the list and dividing 24 by
each
  // integer retrieved from the list
  for(Integer i: numList){
  try{
   System.out.println("Dividing by " + i);
```

Example for try-catch block

```
// Division by zero will throw exception
   b = 24 / i;
  }catch (ArithmeticException aExp) {
     System.out.println("Division by
zero.");
     // Setting value to zero in case
     // of exception
     b = 0;
     aExp.printStackTrace();
  }
  System.out.println("Value of b " + b);
  }
  System.out.println("After for loop");
 }
}
```

Try-Catch: Example

Output

```
Dividing by 2
Value of b 12
Dividing by 3
Value of b 8
Dividing by 0
Division by zero.
Java.lang.ArithmeticException: / by zero
Value of b 0 at
org.netjs.examples.impl.Exception_Example.main(Exception_
Example.Java:23)
Dividing by 6
Value of b 4
Dividing by 8
Value of b 3
After for loop
```

Exception Handling: Throw and Throws

Throw statement

A throw statement throws an exception in Java.

Syntax:

```
throw throwableObject;
```

Throws keyword

A throws keyword handles any exception thrown by the method by itself.

Syntax:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Note

There are two ways to obtain a throwable object: by accessing the exception parameter of the catch block and by creating a new object using the new operator.

Throw and Throws: Example

Example for throw keyword

```
public class ThrowExample {
 public static void main(String[] args) {
  ThrowExample ThrowExample = new
ThrowExample();
  try{
    ThrowExample.displayValue();
  }catch(NullPointerException nExp){
     System.out.println("Exception caught
in catch block of main");
     nExp.printStackTrace();;
  }
 }
 public void displayValue(){
  try{
    throw new NullPointerException();
  }catch(NullPointerException nExp){
     System.out.println("Exception caught
in catch block of displayValue");
     throw nExp;
  }}}
```

Implementing Exception Handling

Problem Statement:

You have been asked to implement exception handling in Java.

Outcome:

By implementing exception handling in Java, you will learn to manage and respond to runtime errors in a controlled way. This skill is crucial for building reliable applications that can handle unexpected issues without crashing, ensuring a smoother user experience.

Note: Refer to the demo document for detailed steps: 06_Implementing_Exception_Handling

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the IDE and create a new project
2. Create an array and take inputs from the user
3. Execute the code with sample data
4. Implement the method print stack trace
5. Use Try and catch functionality and execute the code

Implementing Custom Exceptions

Problem Statement:

You have been asked to implement custom exceptions in Java.

Outcome:

By implementing custom exceptions in Java, you will learn to create specific error types that can provide clearer and more detailed information about issues that occur within your applications. This practice enhances error management by allowing more precise handling based on distinct exception classes.

Note: Refer to the demo document for detailed steps: 07_Implementing_Custom_Exceptions

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the IDE and create a new project
2. Implement exceptions in Java
3. Create the custom exception objects with example data
4. Compare checked and unchecked exceptions

Implementing throw and throws in a Banking Application

Problem Statement:

You have been asked to implement throw and throws in a banking application in Java.

Outcome:

By implementing throw and throws in a banking application in Java, you will learn to explicitly signal the occurrence of exceptions and declare the potential for exceptions in method signatures. This enables robust error handling strategies, essential for maintaining the integrity and security of financial transactions within your application.

Note: Refer to the demo document for detailed steps: 08_Implementing_throw_and_throws_in_a_Banking_Application

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the IDE and create a new project
2. Create a class with an executable method
3. Use a reference variable with the default constructor
4. Execute the code with example data
5. Add the code in the try and catch

Multithreading

Multithreading

It involves running multiple threads simultaneously. Threads are lightweight subprocesses and the smallest unit of processing.

- Reduces inter-thread communication cost
- Minimizes overhead compared to process-based multitasking
- Shares the same address space to save on memory requirements
- Lowers the cost of context switching

Multithreading: Mechanism

In Java, there are two mechanisms for creating a new thread:

Extending the thread class

Create a new class that extends Thread, override the **run()** method, and call **start()** to begin thread execution.

Implementing the Runnable interface

Create a new class that implements Runnable, define behavior in the **run()** method, pass an instance to a Thread constructor, and call **start()** to initiate thread execution.

Multithreading: Syntax

Following is the syntax to implement a **runnable** method:

```
public class Example implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread has
terminated");
    }
    public static void main(String[] args){
        Example example = new Example();
        Thread t1= new Thread(example);
        t1.start();
        System.out.println("Hello");
    }  }
```

Multithreading: Example

Thread creation by implementing runnable interface:

Example

```
class UseThread implements Runnable{
  @Override
  public void run() {
    System.out.println("In run method of
UseThread- "
      + Thread.currentThread().getName());
  }
}
public class ThreadLearn {
  public static void main(String[] args) {
    System.out.println("In main method- " +
Thread.currentThread().getName());
    // Passing runnable instance
    Thread thread = new Thread(new
UseThread(), "UseThread");
    // Calling start method
    thread.start();
  } }
```

Thread creation by extending thread class:

Example

```
class UseThread extends Thread{
  @Override
  public void run() {
    System.out.println("In run method of
UseThread- "
      + Thread.currentThread().getName());
  }
}

public class ThreadDemo {
  public static void main(String[] args) {
    System.out.println("In main method- " +
Thread.currentThread().getName());
    // Calling start method
    new UseThread().start();
  }}
```

Output

```
In main method- main
In run method of UseThread- UseThread
```

Multithreading: Thread Categories

In multithreading, there are two distinct categories of threads:

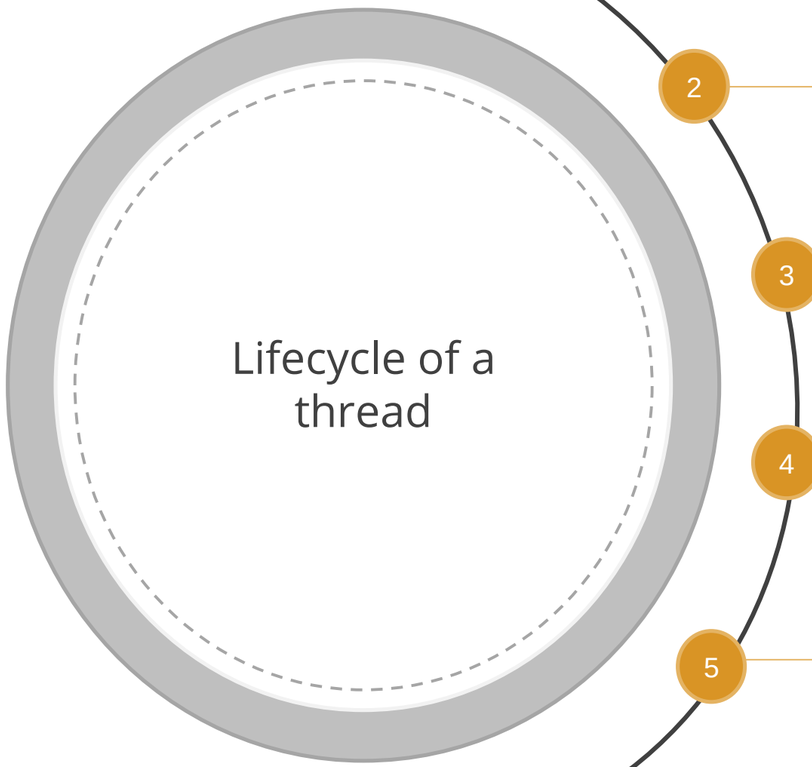
User threads

Perform tasks associated with the application

Daemon threads

Perform tasks for the program when it is running

Multithreading: Thread Lifecycle



1. **New state**
When a thread is created with the runnable interface

2. **Runnable state**
When the start() method is called, and the thread is ready to execute

3. **Blocked state**
When a thread is waiting to acquire a lock

4. **Waiting state**
When a thread is waiting indefinitely for another thread to perform a specific action

5. **Timed waiting**
When a thread is waiting for another thread to perform an action for a specified time

6. **Terminated**
When a thread completes its execution

Implementing Multithreading

Problem Statement:

You have been asked to implement multithreading in Java.

Outcome:

By implementing multithreading in Java, you will learn to run multiple threads concurrently, enhancing the performance and responsiveness of your applications. This skill is crucial for developing efficient programs that can handle multiple tasks simultaneously.

Note: Refer to the demo document for detailed steps: 09_Implementing_Multithreading

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new class
2. Write a basic for loop and execute the code
3. Use the paste operation to write a print task
4. Implement a use case with sample data and execute the code
5. Implement multithreading and override a method
6. Write a polymorphic statement
7. Print out the names of threads
8. Implement the concept of priority and Max Priority and execute the code
9. Access the state
10. Mark the thread as a daemon method

Synchronized Block

Synchronized Block

A synchronized block allows for synchronizing specific resources within a method's code, functioning similarly to a synchronized method.

Syntax

```
synchronized (object reference expression) {
  //code block
}
```

Synchronized Block: Example

```
class Table
{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 } //end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}         }
```

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table table= new Table(); //only one object
MyThread1 t1=new MyThread1(table);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Executing the Synchronization of Threads in Java

Problem Statement:

You have been asked to demonstrate the concept of synchronization of threads in Java.

Outcome:

By demonstrating the concept of synchronization of threads in Java, you will learn to control access to shared resources, preventing data inconsistency and ensuring thread-safe operations. This is essential for developing reliable and concurrent applications.

Note: Refer to the demo document for detailed steps: 10_Executing_the_Synchronization_of_Threads

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new class
2. Write a for loop to print the document n number of times
3. Write a thread dot sleep with the try and catch
4. Create an object and execute the code
5. Create two threads and work on the same object
6. Execute the code with sample data
7. Implement the concept of synchronization

Future

Future

A future represents the outcome of an asynchronous computation and is returned when an asynchronous task is created, serving as a reference to the task's result.

Syntax:

```
public interface Future<V> {
    boolean cancel(boolean
mayInterruptIfRunning)
    void  get();
    void  get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

Here, **V** is the returnType. If get() is called before an asynchronous task is completed, it will block until the result is available.

Note

After task completion, results can be accessed via returned future objects.

Implementing Callable and Future

Problem Statement:

You have been asked to demonstrate the usage of callable interfaces and futures.

Outcome:

By demonstrating the usage of Callable interfaces and Futures in Java, you will learn to execute tasks that can return results and handle asynchronous computations. This enhances your ability to write efficient and responsive applications by managing concurrent tasks effectively.

Note: Refer to the demo document for detailed steps: 11_Implementing_Callable_and_Future

ASSISTED PRACTICE

Assisted Practice: Guidelines

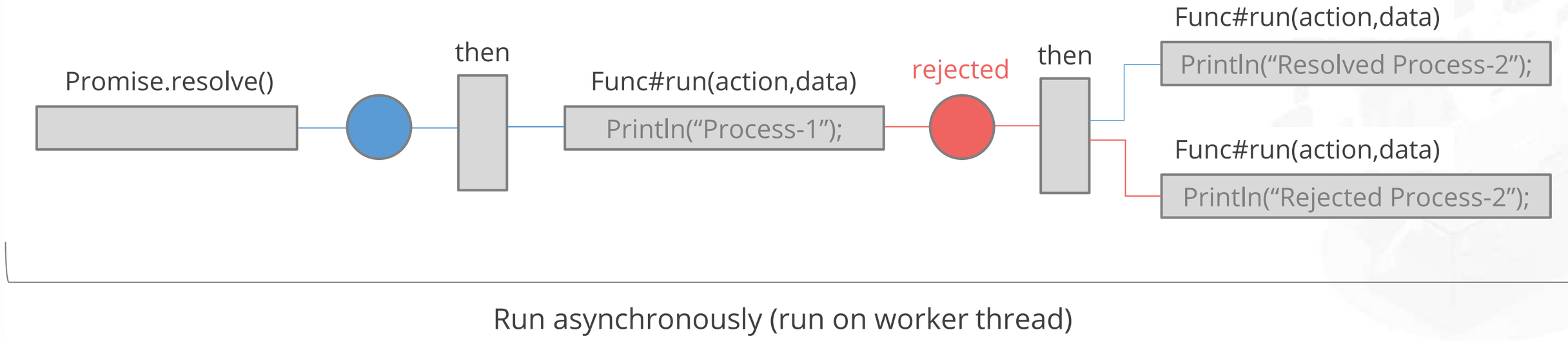
Steps to be followed are:

1. Implement a thread using Callable and Future Interfaces along with the suitable scenarios

Promise

Promise

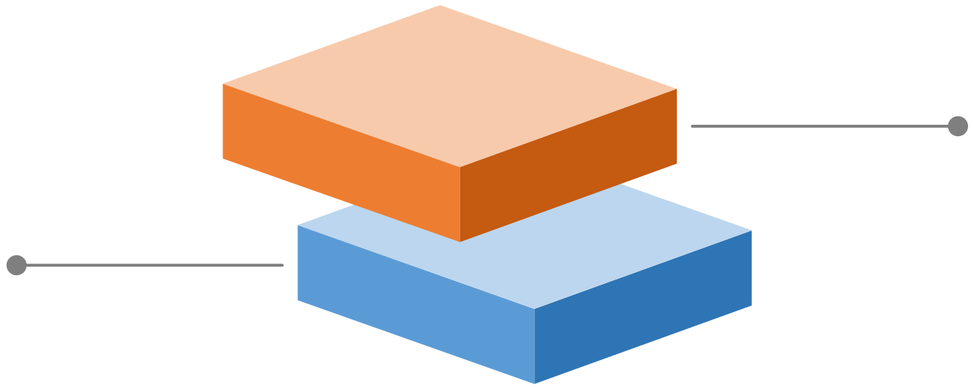
A promise handles asynchronous tasks that have yet to be finished. It acts as a placeholder for the task until it is done.



Promise

It can manage asynchronous tasks that have not been finished yet but will do so in the future.
It acts as a placeholder for the task until it's done.

Advantages of Promises over callbacks are:



It helps in function composition and error handling.

It prevents callback hell and gives callback aggregation.

Key Takeaways

- An interface cannot be instantiated, indicating that the interfaces cannot have constructors.
- The initialized variable is constant in the interface. The value cannot be changed in the implementing class.
- It is crucial to note that if a class does not implement all the interface methods, it should be declared abstract.
- A package is a collection of related classes, interfaces, and sub-packages organized hierarchically.
- Access modifiers control the visibility of the class members (fields and methods) or the class.

Key Takeaways

- When an exceptional condition occurs within a method, the method creates an exception object and throws it.
- The advantages of multithreading include less overhead, saving memory requirements, inexpensive context switching, and inter-thread communication.
- A Java future depicts the results of an asynchronous computation.
- Synchronized blocks enable synchronization on specific resources within a method. Any code within these blocks behaves similarly to that in a synchronized method.
- A promise is a way of writing async code that still appears to be executed synchronously.

TECHNOLOGY

Thank You

simplilearn